HB0087
Handbook
CorePCIF v4.1

Downloaded from Arrow.com.

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Fax: +1 (949) 215-4996
Email: sales.support@microsemi.com
www.microsemi.com

© 2017 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.

Microsemi makes no warranty, representation, or guarantee regarding the information contained herein or the suitability of its products and services for any particular purpose, nor does Microsemi assume any liability whatsoever arising out of the application or use of any product or circuit. The products sold hereunder and any other products sold by Microsemi have been subject to limited testing and should not be used in conjunction with mission-critical equipment or applications. Any performance specifications are believed to be reliable but are not verified, and Buyer must conduct and complete all performance and other testing of the products, alone and together with, or installed in, any end-products. Buyer shall not rely on any data and performance specifications or parameters provided by Microsemi. It is the Buyer's responsibility to independently determine suitability of any products and to test and verify the same. The information provided by Microsemi hereunder is provided “as is, where is” and with all faults, and the entire risk associated with such information is entirely with the Buyer. Microsemi does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other IP rights, whether with regard to such information itself or anything described by such information. Information provided in this document is proprietary to Microsemi, and Microsemi reserves the right to make any changes to the information in this document or to any products and services at any time without notice.

About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, California, and has approximately 4,800 employees globally. Learn more at www.microsemi.com.

50200087. 8.0 11/17

Downloaded from Arrow.com.

Contents

1 Revision History 1
- 1.1 Revision 8.0 1
- 1.2 Revision 7.0 1
- 1.3 Revision 6.0 1
- 1.4 Revision 5.0 1
- 1.5 Revision 4.0 1
- 1.6 Revision 3.0 1
- 1.7 Revision 2.0 1
- 1.8 Revision 1.0 1

2 Introduction 2
- 2.1 Core Versions 2
- 2.2 Supported Families 3
- 2.3 CorePCIF Device Requirements 3
- 2.4 Utilization Statistics 4
- 2.5 Performance Statistics 7
- 2.6 I/O Requirements 8
- 2.7 Electrical Requirements 9

3 Functional Description 10
- 3.1 Target Controller 10
- 3.2 CorePCIF – Master Controller 10
- 3.3 CorePCIF – Datapath 11
- 3.4 CorePCIF – Internal Data Storage 11
- 3.5 CorePCIF Target Function 11
  - 3.5.1 Supported Target Commands 11
  - 3.5.2 I/O Read (0010) and Write (0011) 12
  - 3.5.3 Memory Read (0110) and Write (0111) 12
  - 3.5.4 Memory Read Multiple (1100) and Memory Read Line (1110) 12
  - 3.5.5 Memory Write and Invalidate (1111) 12
  - 3.5.6 Configuration Read (1010) and Write (1011) 12
  - 3.5.7 Disconnects and Retries 12
- 3.6 CorePCIF Master Function 13
  - 3.6.1 Backend Interface 13
  - 3.6.2 Supported Master Commands 13
  - 3.6.3 DMA Master Registers 13
  - 3.6.4 Master Transfers 14
  - 3.6.5 Master Byte Commands 14
- 3.7 CardBus Support 14
- 3.8 CompactPCI Hot-Swap Support 14
- 3.9 CorePCIF Backend Dataflow 15
  - 3.9.1 Burst Transfers 15
  - 3.9.2 Byte-Controlled Transfers 15
  - 3.9.3 Dataflow Control 15
- 3.10 FIFO Recovery Logic 15
- 3.11 Example System Implementation 16

4 Core Structure 18

Downloaded from Arrow.com.

5 Tool Flows . . . 21
5.1 SmartDesign . . . 21
5.2 Synthesis in Libero . . . 23
5.3 Place-and-Route in Libero . . . 24

6 Configuring Parameters . . . 26
6.1 General Configuration Parameters . . . 26
6.2 PCI Configuration Space Parameters . . . 28
6.3 BAR Parameters . . . 28
6.4 Master/DMA Parameters . . . 30
6.5 Default Core Parameter Settings . . . 30

7 Core Interfaces . . . 34
7.1 PCI Bus Signals . . . 34
7.2 Backend System-Level Signals . . . 35
7.3 Backend Target and Master Dataflow Signals . . . 36
7.4 Backend Target Dataflow Signals . . . 38
7.5 Backend Master Dataflow Signals . . . 38
7.6 Backend Master DMA Register Access Signals . . . 39
7.7 Hot-Swap Interface . . . 40

8 Timing Diagrams . . . 41
8.1 Single-Cycle Read and Write . . . 41
8.2 Burst Transfer at Maximum Transfer Rate . . . 44
8.3 Burst Transfer with a Slow PCI Master . . . 46
8.4 Burst Transfer with a Slow Backend . . . 48
8.5 Burst Transfer with FIFO Recovery Enabled . . . 50
8.6 Byte-Controlled Transfers . . . 52
8.7 64-Bit Burst Transfer . . . 55
8.7.1 Operating Note . . . 57
8.8 Slow Read Transfers . . . 57
8.9 Backend-Terminated (BUSY) Cycle at Transfer Start (Target) . . . 58
8.10 Backend-Terminated (ERROR) Cycle at Transfer Start (Target) . . . 60
8.11 Backend-Terminated (BUSY) Cycle during Data Burst (Target) . . . 60
8.12 PCI Configuration Cycle . . . 63
8.13 PCI Interrupt Generation . . . 64
8.14 Simple DMA Transfer . . . 65
8.15 DMA Operation with a FIFO Backend . . . 68
8.16 STOP_MASTER Assertion during Data Burst . . . 69
8.17 RD_BUSY_MASTER and WR_BUSY_MASTER Operation . . . 73
8.18 STALL_MASTER Operation . . . 75
8.19 DMA Register Access from the Backend . . . 79
8.20 Direct DMA Transfers . . . 81
8.21 Hot-Swap Sequence . . . 83

9 PCI Configuration Space . . . 86
9.1 Target Configuration Space . . . 86
9.1.1 Read-Only Configuration Registers . . . 88
9.1.2 Read/Write Configuration Registers . . . 89

10 Testbench Operation . . . 96

Downloaded from Arrow.com.

10.1 Verification Testbench 96
10.1.1 Customizing the Verification Testbench 98
10.1.2 Files Used in the Verification Testbench 98
10.2 User Testbench 99
10.2.1 Files Used in the User Testbenches 100
10.2.2 Testbench Operation 101
10.2.3 Customizing the User Testbenches 103

11 Implementation Hints 104
11.1 Clocking 104
11.1.1 Example: Clocking in SmartFusion2 104
11.2 Clock and Reset Networks 106
11.3 Assigning Pin Layout Constraints 106
11.4 Pin Assignments 106
11.4.1 SX-A and RTSX-S Families 107
11.4.2 ProASICPLUS Family 107
11.4.3 Axcelerator and RTAX-S Families 107
11.4.4 Fusion, IGLOO/e, ProASIC3L, and ProASIC3/E Families 107
11.4.5 SmartFusion2 108
11.4.6 RTG4 109
11.4.7 PolarFire 110
11.4.8 All Families 111
11.4.9 Meeting PCI Hold Requirements 111

12 PCI Pinout 112

13 Synthesis Timing Constraints 113

14 Place-and-Route Timing Constraints 114

15 Verification Testbench Tests 115

16 VHDL User Testbench Procedures 117

17 Verilog User Testbench Procedures 119

18 Ordering Information 121
18.1 Ordering Codes 121

Downloaded from Arrow.com.

Figures

Figure 1 CorePCIF System Block Diagram . . . 2
Figure 2 CorePCIF Block Diagram . . . 10
Figure 3 External FIFO Connection (Target mode) . . . 16
Figure 4 External FIFO Connection (Master mode) . . . 16
Figure 5 Simple Target Implementation . . . 16
Figure 6 Master and Target Implementation . . . 17
Figure 7 CorePCIF Structure . . . 18
Figure 8 CorePCIF Full I/O View . . . 21
Figure 9 CorePCIF Configurator . . . 22
Figure 10 CorePCIF Configurator (Continued) . . . 23
Figure 11 Backend Read Cycle (RD_SYNC = 0) . . . 42
Figure 12 Backend Read Cycle (RD_SYNC = 1) . . . 43
Figure 13 Backend Write Cycle . . . 44
Figure 14 Backend Burst Read Cycle (RD_SYNC = 0) . . . 45
Figure 15 Backend Burst Read Cycle (RD_SYNC = 1) . . . 45
Figure 16 Backend Burst Write Cycle . . . 46
Figure 17 Backend Read Cycle with Slow PCI Master (RD_SYNC = 0) . . . 47
Figure 18 Backend Burst Read Cycle with Slow PCI Master (RD_SYNC = 1) . . . 47
Figure 19 Backend Burst Write Cycle with Slow PCI Master . . . 48
Figure 20 Backend Burst Read Cycle with Slow Backend (RD_SYNC = 0) . . . 49
Figure 21 Backend Burst Read Cycle with Slow Backend (RD_SYNC = 1) . . . 49
Figure 22 Backend Burst Write Cycle with Slow Backend . . . 50
Figure 23 Backend Burst Write Cycle with Additional Writes after Ready Removed . . . 50
Figure 24 FIFO Recovery Operation (RD_SYNC = 0) . . . 51
Figure 25 FIFO Recovery Operation (RD_SYNC = 1) . . . 52
Figure 26 Backend Byte Read Cycle (RD_SYNC = 0) . . . 53
Figure 27 Byte Burst Read Cycle (RD_SYNC = 1) . . . 54
Figure 28 Byte Burst Write Cycle . . . 55
Figure 29 64-Bit Burst Read Cycle (RD_SYNC = 0) . . . 56
Figure 30 64-Bit Burst Read Cycle (RD_SYNC = 1) . . . 56
Figure 31 64-Bit Burst Write Cycle . . . 57
Figure 32 Slow Read Transfer (RD_SYNC = 0) . . . 58
Figure 33 Slow Read Transfer (RD_SYNC = 1) . . . 58
Figure 34 Backend-Terminated (BUSY) Cycle at Transfer Start . . . 59
Figure 35 Backend Fails to Assert RD_STB_IN . . . 59
Figure 36 Backend-Terminated (ERROR) Cycle at Transfer Start . . . 60
Figure 37 Backend Burst Read Cycle Terminated by BUSY (RD_SYNC = 0) . . . 61
Figure 38 Backend Burst Read Cycle Terminated by BUSY (RD_SYNC = 1) . . . 62
Figure 39 Backend Burst Write Cycle Terminated by BUSY . . . 62
Figure 40 Configuration Read Cycle . . . 63
Figure 41 Configuration Write Cycle . . . 64
Figure 42 PCI Interrupt Generation and Acknowledge Sequence . . . 64
Figure 43 DMA Burst Read Cycle Including DMA Start Sequence . . . 65
Figure 44 DMA Burst Read Cycle (RD_SYNC = 0) . . . 66
Figure 45 DMA Burst Read Cycle (RD_SYNC = 1) . . . 67
Figure 46 DMA Burst Write Cycle . . . 67
Figure 47 DMA Cycle with Grant Removal during Startup . . . 68
Figure 48 DMA Cycle with a FIFO Backend . . . 69
Figure 49 STOP_MASTER Assertion during DMA Burst Read Cycle (RD_SYNC = 0) . . . 70
Figure 50 STOP_MASTER Assertion during DMA Burst Read Cycle (RD_SYNC = 1) . . . 71
Figure 51 STOP_MASTER Assertion during DMA Burst Write Cycle . . . 72
Figure 52 STOP_MASTER Held Asserted during DMA Burst Read . . . 73
Figure 53 RD_BUSY_MASTER Operation . . . 74
Figure 54 WR_BUSY_MASTER Operation . . . 75

Downloaded from Arrow.com.

Figure 55 STALL_MASTER Assertion DMA Read Cycle (RD_SYNC = 0) 76
Figure 56 STALL_MASTER Assertion DMA Read Cycle (RD_SYNC = 1) 77
Figure 57 STALL_MASTER Assertion DMA Write Cycle 78
Figure 58 STALL_MASTER Assertion and Cycle Aborted due to Loss of GNTN 79
Figure 59 DMA Register Single Write Cycle 79
Figure 60 DMA Register Single Read Cycle 80
Figure 61 DMA Register Burst Write Cycle 80
Figure 62 DMA Register Burst Read Cycle 80
Figure 63 DMA Register Access and DMA Startup 81
Figure 64 Direct DMA Write to the PCI Bus 82
Figure 65 Direct DMA Read from the PCI Bus 83
Figure 66 Hot-Swap Insertion Sequence 84
Figure 67 Hot-Swap Extraction Sequence 85
Figure 68 The Verification Testbench 96
Figure 69 User Testbench 100
Figure 70 User Testbench Startup Sequence 102
Figure 71 Clock Generation 104
Figure 72 Clocking in SmartFusion2 with No CCC 105
Figure 73 Clocking in SmartFusion2 with CCC 105
Figure 74 FCCC Configuration in SmartFusion2 106
Figure 75 SmartFusion2 M2S050T Device 108
Figure 76 RTG4 RT4G150-CG1657 Device 110
Figure 77 PolarFire MPF300-FCG1152 Device 111
Figure 78 Recommended PCI Pin Ordering 112

Downloaded from Arrow.com.

Tables

Table 1 Example Implementations . . . 3
Table 2 CorePCIF Utilization . . . 4
Table 3 32-Bit CorePCIF Device Utilization . . . 5
Table 4 64-Bit CorePCIF Device Utilization . . . 6
Table 5 Device Speed Grade Requirements . . . 7
Table 6 PCI Bus Timing . . . 8
Table 7 CorePCIF I/O Requirements . . . 8
Table 8 Supported Electrical Environments . . . 9
Table 9 Supported PCI Target Commands . . . 11
Table 10 DMA Register Addresses . . . 13
Table 11 CorePCIF Common Source Files . . . 18
Table 12 Technology-Specific Source Files . . . 19
Table 13 CorePCIF Miscellaneous Source Files . . . 19
Table 14 Designer Compile Options . . . 24
Table 15 General Parameters . . . 26
Table 16 PCI Configuration Space Parameters . . . 28
Table 17 BAR Parameters . . . 28
Table 18 Master/DMA Parameters . . . 30
Table 19 Default Build Parameters[1] . . . 30
Table 20 PCI Bus Interface Signals . . . 34
Table 21 System-Level Signals . . . 35
Table 22 Dataflow Interface Signals . . . 36
Table 23 Target Mode Control Signals . . . 38
Table 24 Master Mode Signals . . . 38
Table 25 Backend DMA Register Access Signals . . . 39
Table 26 Hot-Swap Interface Signals . . . 40
Table 27 Example Waveforms . . . 41
Table 28 Backend Initial Access Time Limits—
Delay Allowed from DP_START to RD_STB_IN or WR_BE_RDY (clock cycles) . . . 43
Table 29 PCI Configuration Space . . . 86
Table 30 PCI Configuration Space . . . 86
Table 31 Capability Structure (Target-only cores with hot-swap) . . . 87
Table 32 Capability Structure (Master cores with hot-swap) . . . 87
Table 33 Capability Structure (Target-only cores with hot-swap and FIFO status) . . . 88
Table 34 Capability Structure (Master cores with hot-swap and FIFO status) . . . 88
Table 35 Command Register 04 Hex . . . 89
Table 36 Status Register 06 Hex . . . 90
Table 37 Base Address Registers (memory) 10 Hex to 24 Hex . . . 90
Table 38 Base Address Registers (I/O) 10 Hex to 24 Hex . . . 91
Table 39 Expansion ROM Address Register 30 Hex . . . 91
Table 40 Capabilities Pointer 34 Hex . . . 91
Table 41 Interrupt Register 3C Hex . . . 91
Table 42 Hot-Swap Capability Register 40 Hex . . . 91
Table 43 Microsemi Capabilities Register 44, 48, or 4C Hex . . . 92
Table 44 Interrupt Control Register 48 Hex (MASTER = 0) . . . 92
Table 45 FIFO Status Register . . . 93
Table 46 PCI Address Register 50 Hex . . . 93
Table 47 Backend Address Register 54 Hex (ENABLE_DIRECTDMA = 0) . . . 93
Table 48 Backend Address and Data Register 54 Hex (ENABLE_DIRECTDMA = 1) . . . 93
Table 49 DMA Transfer Count 58 Hex . . . 94
Table 50 DMA Control Register 5C Hex . . . 94
Table 51 Verification Testbench Configurations . . . 96
Table 52 Verification Testbench Source Files . . . 98
Table 53 User Testbench Source Files . . . 100

Downloaded from Arrow.com.

Table 54 User Testbench Test Sequence 102
Table 55 Synthesis Timing Constraints 113
Table 56 Place-and-Route Timing Constraints 114
Table 57 Verification Testbench Tests 115
Table 58 Procedure Call Parameters 117
Table 59 Global Descriptions 119
Table 60 Parameter Descriptions 119
Table 61 Ordering Codes 121

Downloaded from Arrow.com.

1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

1.1 Revision 8.0

Updated changes related to CorePCIF v4.1.

1.2 Revision 7.0

Updated changes related to CorePCIF v4.0.

1.3 Revision 6.0

Updated changes related to CorePCIF v3.2.

1.4 Revision 5.0

Updated changes related to CorePCIF v3.1.

1.5 Revision 4.0

Updated changes related to CorePCIF v3.0.

1.6 Revision 3.0

Updated changes related to CorePCIF v2.1.

1.7 Revision 2.0

Updated changes related to CorePCIF v2.0.

1.8 Revision 1.0

Revision 1.0 was the first publication of this document. Created for CorePCIF v1.0

Downloaded from Arrow.com.

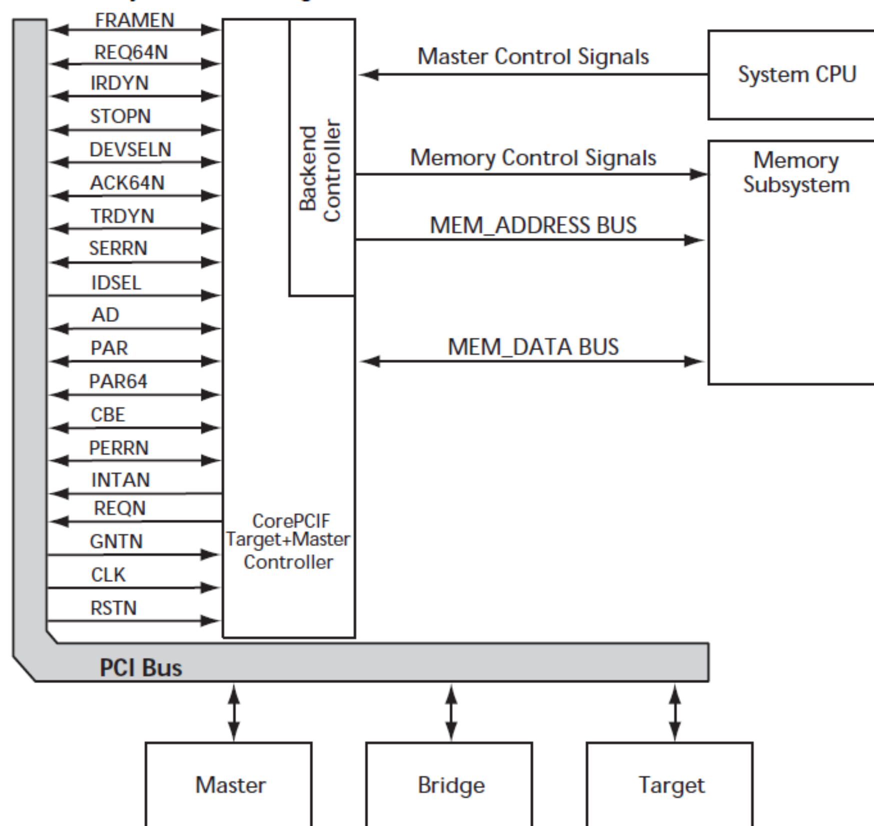
2 Introduction

CorePCIF connects memory, FIFO, and processor subsystem resources to the main system via the PCI bus. CorePCIF is intended for use with a wide variety of peripherals where high-performance data transactions are required. Figure 1 depicts typical system applications using the core. Though CorePCIF can handle any transfer rate, most applications will operate with zero wait states. When required, wait states can be inserted automatically by a slower peripheral.

CorePCIF can implement Target and/or Master functions. The Target function allows the PCI bus to access memory devices attached to the CorePCIF backend. The Master function allows CorePCIF to move data between the backend or internal registers and the PCI bus using the internal DMA engine. The DMA engine can be programmed either from the PCI bus or directly from the backend.

CorePCIF can be customized. A variety of parameters are provided to easily change features such as memory and I/O sizes along with the PCI vendor and device IDs. A single top-level core has parameters that enable and disable functions, allowing a minimal-size core to be implemented for the required functionality. The core consists of four basic units: the Target controller, the Master controller, the DMA controller, and the backend controller. The backend controller provides the necessary control for the I/O or memory subsystem, allowing external (to the core) memory and FIFOs to be directly connected to the core.

Figure 1 • **CorePCIF System Block Diagram**



2.1 Core Versions

This handbook applies to CorePCIF v4.1. The release notes provided with the core list known discrepancies between this handbook and the core release associated with the release notes.

Downloaded from Arrow.com.

2.2 Supported Families

- PolarFire®
- RTG4™
- SmartFusion®2
- IGLOO®2
- SmartFusion®
- IGLOO®
- IGLOO®e
- Fusion
- ProASIC®3
- ProASIC®3E
- ProASIC®3L
- ProASICPLUS®
- Axcelerator®
- RTAX-S
- SX-A
- RTSX-S

2.3 CorePCIF Device Requirements

CorePCIF includes Target and/or Master functions. The core also has an option for a built-in DMA controller.

There are nine implementations available for the core. The SMALL32 implementation is the smallest Target core possible but does not support zero-wait-state transfers; TARG32 does support zero-wait-state transfers. MAST32 is the smallest Master-only core possible. TARGDMA32 implements a typical Target and Master function. TARGMAST32 implements a fully configured core. The remaining four implementations are 64-bit versions of the 32-bit implementations. Table 1 describes example implementations.

Table 1 • **Example Implementations**

Implementation	Description
SMALL32	32-bit Target-only core with a single base address register (BAR). The slow read function is enabled. Interrupts, BAR overflow, and hot-swap features are disabled.
TARG32	32-bit Target-only core with a single 64 kB BAR. The FIFO recovery logic is not implemented. BAR overflow logic and hot-swap features are disabled.
MAST32	32-bit Master-only core with a single 64 kB BAR. The FIFO recovery logic is not implemented. Direct DMA is enabled.
TARGDMA32	32-bit Target and Master function with a single 64 kB BAR. DMA registers are accessible from the PCI side and are memory-mapped in the second BAR. The FIFO recovery logic is not implemented. BAR overflow logic and hot-swap features are disabled. Backend access to the DMA registers is not implemented. Direct DMA is disabled.
TARGMAST32	32-bit Target and Master function with five memory BARs that have variable sizes from 64 kB to 1 GB. The DMA registers are memory-mapped to the sixth BAR. All of the memory BARs include the FIFO recovery logic. The Expansion ROM address registers are also implemented. BAR overflow logic and hot-swap features are enabled. Backend access to the DMA registers is also implemented. Direct DMA is enabled.
TARG64	64-bit Target-only core with a single 64 kB BAR. The FIFO recovery logic is not implemented. BAR overflow logic and hot-swap features are disabled. Direct DMA is disabled.
MAST64	64-bit Master-only core with a single 64 kB BAR. The FIFO recovery logic is not implemented. Direct DMA is disabled.

Downloaded from Arrow.com.

Table 1 • **Example Implementations**

Implementation	Description
TARGDMA64	64-bit Target and Master function with a single 64 kB memory. DMA registers are accessible from the PCI side and are memory-mapped in the second BAR. The FIFO recovery logic is not implemented. BAR overflow logic and hot-swap features are disabled. Backend access to the DMA registers is not implemented. Direct DMA is disabled.
TARGMAST64	64-bit Target and Master function with five memory BARs that have variable sizes from 64 kB to 1 GB. The DMA registers are memory-mapped to the sixth BAR. All of the memory BARs include the FIFO recovery logic. The Expansion ROM address registers are also implemented. BAR overflow logic and hot-swap features are enabled. Backend access to the DMA registers is also implemented. Direct DMA is enabled.

2.4 Utilization Statistics

Table 3 and Table 4 give the CorePCIF device utilization for both 32-bit and 64-bit implementations. The numbers in these tables are typical and will vary based on the actual core configuration and the synthesis tools used.

CorePCIF device utilization and performance varies, depending on which features are implemented. The core has approximately 50 configuration parameters.

Table 2 • **CorePCIF Utilization**

Implementation	Family	Combinational (4LUT)	Sequential (DFF)	Total	Memory Blocks	Utilization %
SMALL32	SmartFusion2 (M2S050) / IGLOO2 (M2GL050)	384	231	615	0	0.54
TARG32		497	291	788	1	0.70
MAST32		947	448	1395	1	1.24
TARGDMA32		1092	459	1551	1	1.37
TARGMAST32		1740	707	2447	1	2.18
TARG64		706	468	1174	2	1.04
MAST64		1135	635	1770	2	1.57
TARGDMA64		1169	642	1811	2	1.60
TARGMAST64		2029	932	2961	2	2.62
SMALL32	RTG4 (RTG4150)	383	231	614	0	0.20
TARG32		559	315	874	1	0.29
MAST32		1052	451	1503	1	0.49
TARGDMA32		1040	467	1507	1	0.50
TARGMAST32		1916	714	2630	1	0.86
TARG64		678	468	1146	2	0.43
MAST64		1191	634	1824	2	0.60
TARGDMA64		1251	642	1893	2	0.62
TARGMAST64		2148	920	3068	2	1.01

Downloaded from Arrow.com.

Microsemi
Power Matters.

Table 2 • **CorePCIF Utilization**

Implementation	Family	Combinational (4LUT)	Sequential (DFF)	Total	Memory Blocks	Utilization %
SMALL32	PolarFire (MPF300T_ES)	361	231	592	0	0.10
TARG32		536	291	827	1	0.14
MAST32		893	449	1342	1	0.22
TARGDMA32		1083	459	1542	1	0.25
TARGMAST32		1722	706	2428	1	0.40
TARG64		735	468	1203	2	0.20
MAST64		1099	636	1735	2	0.29
TARGDMA64		1198	642	1840	2	0.30
TARGMAST64		1822	919	2741	2	0.46

Note: The data in Table 2 was achieved using Verilog RTL with typical synthesis and layout settings. Frequency (in MHz) was set to 100 and speed grade was STD. The data given in Table 2 is indicative only. The overall device utilization and performance of the core is system dependent.

The exact parameter settings are detailed in Table 5.

Table 3 • **32-Bit CorePCIF Device Utilization**

Implementation	Family	Cells or Tiles Combinatorial	Sequential	Total	Memory Blocks	Device	Utilization
SMALL32	Fusion® IGLOO®/e ProASIC®3/E ProASIC3L	544	177	721	0	AFS600	5%
TARG32		661	203	864	2	AGLE600	6%
MAST32		1,434	383	1,817	2	AGL600	13%
TARGDMA32		1,594	369	1,963	2	A3PE600	14%
TARGMAST32		2,698	658	3,356	2	A3P600	24%
SMALL32	ProASIC^PLUS®	658	215	873	0	APA150	14%
TARG32		716	208	924	4	APA150	15%
MAST32		1,479	422	1,901	4	APA150	31%
TARGDMA32		1,644	377	2,021	4	APA300	25%
TARGMAST32		3,020	697	3,717	4	APA300	45%
SMALL32	RTAX-S	350	178	528	0	RTAX250S	12%
TARG32		465	244	709	0	RTAX250S	17%
MAST32		799	422	1,221	0	RTAX250S	29%
TARGDMA32		867	414	1,281	0	RTAX250S	30%
TARGMAST32		2,562	2,206	4,768	0	RTAX1000S	26%
SMALL32	Axcelerator®	381	180	561	0	AX500	7%
TARG32		453	210	663	1	AX500	8%
MAST32		830	393	1,223	1	AX500	15%
TARGDMA32		874	380	1,254	1	AX500	16%
TARGMAST32		1,677	653	2,330	1	AX500	29%

Downloaded from Arrow.com.

Table 3 • **32-Bit CorePCIF Device Utilization *(continued)***

Implementation	Family	Cells or Tiles Combinatorial	Sequential	Total	Memory Blocks	Device	Utilization
SMALL32	RTSX-S	387	221	608	0	RT54SX32S	21%
TARG32		491	282	773	0	RT54SX32S	27%
MAST32		1,134	507	1,641	0	RT54SX32S	57%
TARGDMA32		966	465	1,431	0	RT54SX72S	24%
TARGMAST32		1,359	834	2,193	0	RT54SX72S	36%
SMALL32	SX-A	385	222	607	0	A54SX32A	21%
TARG32		494	285	779	0	A54SX32A	27%
MAST32		1,111	507	1,618	0	A54SX32A	56%
TARGDMA32		959	460	1,419	0	A54SX72A	24%
TARGMAST32		1,352	834	2,186	0	A54SX72A	36%

Table 4 • **64-Bit CorePCIF Device Utilization**

Implementation	Family	Cells or Tiles Combinatorial	Sequential	Total	Memory Blocks	Device	Utilization
TARG64	Fusion	930	315	1,245	4	AFS600	9%
MAST64	IGLOO/e ProASIC3/E ProASIC3L	1,686	498	2,184	4	AGLE600 AGL600 A3PE600 A3P600	16%
TARGDMA64		1,852	484	2,336	4		17%
TARGMAST64		2,989	772	3,761	4		27%
TARG64	ProASIC^PLUS	961	319	1,280	8	APA150	21%
MAST64		1,770	542	2,312	8	APA150	38%
TARGDMA64		1,962	500	2,462	8	APA150	40%
TARGMAST64		3,173	814	3,987	8	APA300	49%
TARG64	RTAX-S	634	387	1,021	0	RTAX250S	24%
MAST64		1,002	565	1,567	0	RTAX250S	37%
TARGDMA64		1,087	553	1,640	0	RTAX1000S	9%
TARGMAST64		3,524	3,858	7,382	0	RTAX1000S	41%
TARG64	Axcelerator	642	317	959	2	AX500	12%
MAST64		1,021	502	1,523	2	AX500	19%
TARGDMA64		1,087	493	1,580	2	AX500	20%
TARGMAST64		1,874	765	2,639	2	AX500	33%
TARG64	SX-A	693	456	1,149	0	A54SX32A	40%
MAST64		1,095	682	1,777	0	A54SX72A	29%
TARGDMA64		1,201	645	1,846	0	A54SX72A	31%
TARGMAST64		1,711	1,200	2,911	0	A54SX72A	48%

Downloaded from Arrow.com.

2.5 Performance Statistics

Table 5 and Table 8 give the device speed grades required to meet either 33 MHz or 66 MHz PCI operation for the 32-bit and 64-bit cores for the three operating environments supported by Microsemi. Not all families support 64-bit or 66 MHz operation.

Table 5 • **Device Speed Grade Requirements**

	Family	Commercial	Industrial	Military
33 MHz 32-bit	Fusion	STD	STD	N/A
	IGLOO/e	STD	STD	N/A
	ProASIC3/E/L	STD	STD	N/A
	ProASIC^{PLUS}	STD	STD	STD
	RTAX-S	N/A	N/A	STD
	Axcelerator	STD	STD	STD
	RTSX-S	N/A	N/A	–1
	SX-A	STD	STD	STD
33 MHz 64-bit	Fusion	STD	STD	N/A
	IGLOO/e	STD	STD	N/A
	ProASIC3/E/L	STD	STD	N/A
	ProASIC^{PLUS}	STD	STD	STD
	RTAX-S	N/A	N/A	STD
	Axcelerator	STD	STD	STD
	RTSX-S	N/A	N/A	N/A
	SX-A	STD	STD	STD
66 MHz 32-bit	Fusion	–2	–2	N/A
	IGLOO/e	N/A	N/A	N/A
	ProASIC3/E	–2	–2	N/A
	ProASIC^{PLUS}	N/A	N/A	N/A
	RTAX-S (RTAX250S)	N/A	N/A	–1
	RTAX-S (RTAX1000S to RTAX4000S)	N/A	N/A	N/A
	Axcelerator (AX125 to AX500)	–1	–1	–1
	Axcelerator (AX1000 to AX2000)	–2	–2	–2
	RTSX-S	N/A	N/A	N/A
	SX-A	N/A	N/A	N/A

Downloaded from Arrow.com.

Table 5 • Device Speed Grade Requirements (continued)

	Family	Commercial	Industrial	Military
66 MHz 64-bit	Fusion	–2	–2	N/A
	IGLOO/e	N/A	N/A	N/A
	ProASIC3/E	–2	–2	N/A
	ProASIC^{PLUS}	N/A	N/A	N/A
	RTAX-S (RTAX250S)	N/A	N/A	–1
	RTAX-S (RTAX1000S to RTAX4000S)	N/A	N/A	N/A
	Axcelerator (AX125 to AX500)	–1	–1	–1
	Axcelerator (AX1000 to AX2000)	–2	–2	–2
	RTSX-S	N/A	N/A	N/A
	SX-A	N/A	N/A	N/A

The PCI specification timing requirements are given in Table 6.

Table 6 • PCI Bus Timing

	Setup		Hold		Clock to Out	
Signals	**33 MHz**	**66 MHz**	**33 MHz**	**66 MHz**	**33 MHz**	**66 MHz**
Bussed Signals	7 ns	3 ns	0 ns	0 ns	11 ns	6 ns
Non-Bussed Signals (e.g., GNTN)	10 ns	5 ns	0 ns	0 ns	11 ns	6 ns

2.6 I/O Requirements

Table 7 gives the I/O requirements for CorePCIF. The number of device I/O pins required for the PCI interface varies, depending on the bus width as well as whether the core supports Target and/or Master functions. The number of backend device I/O pins that the core requires depends on the core interface. For instance, a device that implements a PCI-to-serial communication channel may only require a single device I/O pin, whereas a PCI-to-memory interface may require many I/O pins. Table 7 shows the maximum number of I/O pins, assuming all the core backend pins are connected to device I/O pins.

Table 7 • CorePCIF I/O Requirements

	I/O Count				
		Backend		Total	
Core	**PCI**	**Min.**	**Max.**	**Min.**	**Max.**
32-bit Target	48	1	146	49	194
64-bit Target	88	1	219	89	307
32-bit Master with backend interface	49	1	162	50	211
64-bit Master with backend interface	88	1	235	89	323
32-bit Target and Master	50	1	146	51	196
64-bit Target and Master	89	1	219	90	308
32-bit Target and Master with backend interface	50	1	162	51	212
64-bit Target and Master with backend interface	89	1	235	90	324

Downloaded from Arrow.com.

2.7 Electrical Requirements

CorePCIF supports both the 3.3 V and 5.0 V PCI specifications when operating at 33 MHz; at 66 MHz, the PCI bus must operate at 3.3 V. The SX-A and RTSX-S families have I/O buffers that directly support 5.0 V operation. Other families in 5.0 V PCI environments may require external voltage level translator devices, or may not be supported. See Table 8 for details.

Table 8 • **Supported Electrical Environments**

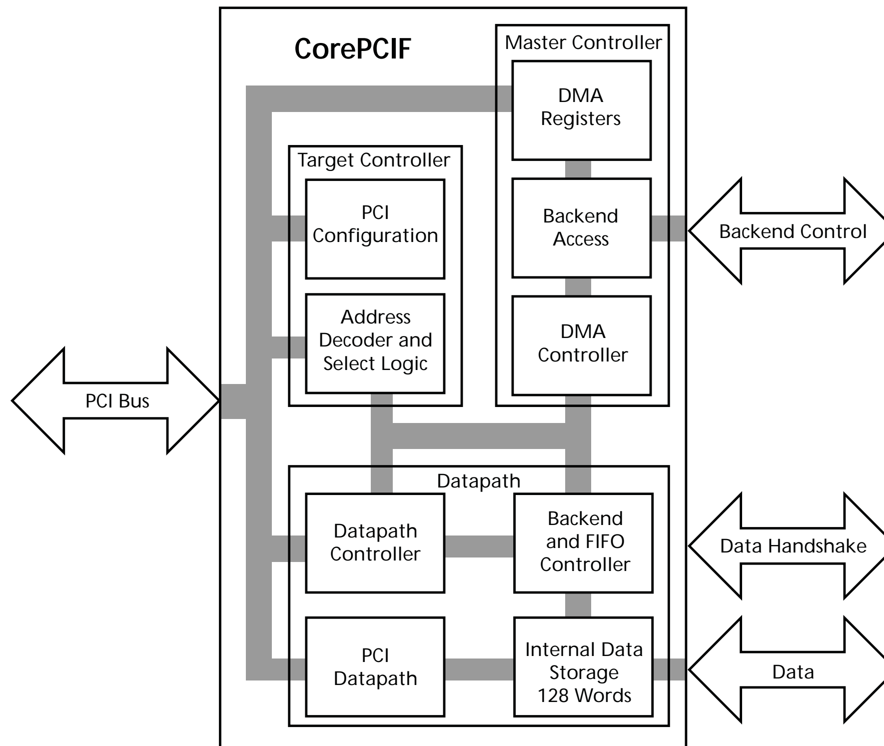
Clock Speed	Family	PCI Voltage with Direct FPGA Connection	PCI Voltage with Level Translators
33 MHz	Fusion	3.3	3.3 and 5.0
	IGLOO/e	3.3	3.3 and 5.0
	ProASIC3/E/L	3.3	3.3 and 5.0
	ProASICPLUS	3.3	3.3 and 5.0
	RTAX-S	3.3	3.3 and 5.0
	Axcelerator	3.3	3.3 and 5.0
	RTSX-S	3.3 and 5.0	3.3 and 5.0
	SX-A	3.3 and 5.0	3.3 and 5.0
	SmartFusion2	3.3 and 5.0	3.3 and 5.0
66 MHz	Fusion	3.3	3.3
	IGLOO/e	3.3	3.3
	ProASIC3/E	3.3	3.3
	ProASICPLUS	Not supported	Not supported
	RTAX-S	3.3	3.3
	Axcelerator	3.3	3.3
	RTSX-S	Not supported	Not supported
	SX-A	Not supported	Not supported
	SmartFusion2	3.3	3.3 and 5.0

Downloaded from Arrow.com.

3 Functional Description

CorePCIF consists of three major functional blocks, shown in Figure 2. These blocks are the Target Controller, Master Controller, and Datapath. With both a Target and Master, all three blocks are required. Otherwise, only the Datapath and either the Target or Master function are required.

Figure 2 • **CorePCIF Block Diagram**



3.1 Target Controller

The Target controller implements the PCI Target function. It contains two sub-blocks: the PCI configuration space and the address decoder logic. The configuration block implements a "type 0" PCI configuration space, supporting up to six base address registers and the Expansion ROM register.

The actual registers implemented are described in Table 29.

The address decoder block monitors the PCI bus for address cycles and compares the address with the base address registers configured in the configuration space. A match signals the datapath controller to start a PCI cycle.

3.2 CorePCIF – Master Controller

The Master controller implements the PCI Master function. It contains three sub-blocks: the DMA registers, DMA controller, and backend access logic. The DMA register block implements the four 32-bit registers that control the DMA controller. These registers can be programmed either from the PCI bus or from the backend.

The DMA controller implements a PCI-compliant Master function that can burst up to 2^{32} bytes of data without intervention. The controller will stop a DMA burst automatically if the backend runs out of data, and restart when data is available.

The backend access block allows a processor connected to the core backend to access the DMA registers and initiate a DMA transfer.

Downloaded from Arrow.com.

3.3 CorePCIF – Datapath

The datapath block provides the data control and storage path between the backend and the PCI bus. It contains four sub-blocks: the PCI datapath, the PCI datapath controller, the backend and FIFO controller, and the internal data storage memory.

The PCI datapath controller is responsible for controlling the PCI control signals and coordinating the data transfers with the backend controller for both Target and Master operations.

The PCI datapath block selects which data should be routed to the PCI bus. Data may come from the PCI configuration block, the DMA register block, or the internal data storage. The datapath block also generates and verifies the PCI parity signals.

The backend controller implements the FIFO control logic. This interfaces to the user's backend logic and moves data from the backend interface into the internal storage. It also includes logic that monitors how much data is actually transferred on the PCI bus. The backend controller can recover data that has not actually been transferred, such as when a Master transfer is terminated with a disconnect without data.

3.4 CorePCIF – Internal Data Storage

CorePCIF includes a 64-word internal memory block for 32-bit PCI data width or a 128-word internal memory block for 64-bit PCI data width that is used to store data being moved from the backend to the PCI bus. Data being transferred from the PCI bus to the backend is not stored internally in the core.

This data storage performs two functions. First, it implements a four-word FIFO that decouples the PCI data transfers from the backend data transfers, thereby increasing throughput. Second, it provides storage for the FIFO recovery logic used to prevent data loss when the backend is connected to a standard FIFO.

Each of the seven supported BARs (six BARs and the Expansion ROM) is allocated eight words of memory. BAR 0 is allocated locations 0–7, BAR 1 is allocated 8–15, and so on. The Expansion ROM is allocated locations 48–55, and the remaining eight locations are not used. Each word is 32 bits wide for 32-bit implementations and 64 bits wide for 64-bit implementations.

For the Axcelerator, ProASIC^{PLUS}, ProASIC3, and ProASIC3E families, the data storage is implemented using FPGA memory resources. For SX-A and RTSX-S families, the storage is implemented using FPGA logic resources. For the RTAX-S family, the storage can be implemented using FPGA logic resources or memory resources. Each BAR will require at least 256 logic modules to implement the storage. Storage is only required for the enabled BARs.

When the SLOW_READ parameter is set, the internal data storage is not implemented, eliminating the need for FPGA memory resources. Instead, the data throughput rate is reduced to prevent data loss.

3.5 CorePCIF Target Function

The CorePCIF Target function acts as a slave on the PCI bus. The Target controller monitors the bus and checks for hits to the configuration space or the address space defined in its BARs. When a hit is detected, the Target controller notifies the backend and then acts to control the flow of data between the PCI bus and the backend.

3.5.1 Supported Target Commands

Table 9 lists the PCI commands supported in the CorePCIF Target implementation.

Table 9 • **Supported PCI Target Commands**

CBEN[3:0]	Command Type	Supported
0000	Interrupt Acknowledge	No
0001	Special Cycle	No
0010	I/O Read	Yes

Downloaded from Arrow.com.

Table 9 • **Supported PCI Target Commands**

CBEN[3:0]	Command Type	Supported
0011	I/O Write	Yes
0100	Reserved	–
0101	Reserved	–
0110	Memory Read	Yes
0111	Memory Write	Yes
1000	Reserved	–
1001	Reserved	–
1010	Configuration Read	Yes
1011	Configuration Write	Yes
1100	Memory Read Multiple	Yes
1101	Dual Address Cycle	No
1110	Memory Read Line	Yes
1111	Memory Write and Invalidate	Yes

3.5.2 I/O Read (0010) and Write (0011)

The I/O Read command is used to read data mapped into I/O address space. The I/O Write command is used to write data mapped into I/O address space. In this case, the write is qualified by the byte enables.

3.5.3 Memory Read (0110) and Write (0111)

The Memory Read command is used to read data in memory-mapped address space. The Memory Write command is used to write data mapped into memory address space. In this case, the write is qualified by the byte enables.

3.5.4 Memory Read Multiple (1100) and Memory Read Line (1110)

The Memory Read Multiple and Memory Read Line commands are treated in the same manner as a Memory Read command. Typically, the bus master will use these commands when data is prefetchable.

3.5.5 Memory Write and Invalidate (1111)

The Memory Write and Invalidate command is treated in the same manner as a Memory Write command.

3.5.6 Configuration Read (1010) and Write (1011)

The Configuration Read command is used to read the configuration space of each device. The Configuration Write command is used to write information into the configuration space. The device is selected if its IDSEL signal is asserted and AD[1:0] are set to '00'. Additional address bits are defined as follows:

- AD[7:2] contain one of 64 DWORD addresses for the configuration registers.
- AD[10:8] indicate which device of a multi-function agent is addressed. The core does not support multi-function devices, and these bits should be '000'.
- AD[31:11] are ignored.

The core supports burst configuration read and write cycles.

3.5.7 Disconnects and Retries

The CorePCIF Target will perform either single-DWORD or burst transactions, depending on the request from the system Master. If the backend is unable to deliver data quickly enough, the Target will respond with either a PCI retry or disconnect, with or without data. If the system Master requests a transfer that the backend is not able to perform, a Target abort can be initiated by the backend.

Downloaded from Arrow.com.

3.6 CorePCIF Master Function

The Master function in CorePCIF is designed to do the following:

- Arbitrate for the PCI bus
- Initiate a PCI cycle
- Pass dataflow control to the Target controller
- End the transfer when the DMA count has been exhausted
- Allow the backend hardware to stop and start DMA cycles

Master transfers can be initiated directly from the backend interface, or another PCI device may program the DMA engine to initiate a PCI transfer.

3.6.1 Backend Interface

Through the backend interface (BE_REQ, BE_GNT, BE_ADDRESS, and so on), an external processor through the backend interface, the AHB interface can access the DMA Master control registers and initiate a Master transfer. This interface also allows the complete PCI configuration space to be accessed so the core can be self-configured by a backend processor. This is required when the core is used to implement the PCI device responsible for configuring the PCI bus. A hardware lock (BE_CFGLOCK) is provided for safety reasons to prevent the backend from changing the values in the PCI configuration space.

3.6.2 Supported Master Commands

The CorePCIF Master controller is capable of performing configuration, I/O, memory, and interrupt acknowledge cycles. Data transfers can be up to 2^{32} bytes.

The Master controller will attempt to complete the transfer using a maximum-length PCI burst unless the maximum burst length bits are set in the control register. If the addressed Target is unable to complete the transfer and performs a retry or disconnect, the Master control will restart the transfer and continue from the last known good transfer. If a Target does not respond (no DEVSELn asserted) or responds with a Target abort cycle, the Master controller will abort the current transaction and report it as an error in the control register.

3.6.3 DMA Master Registers

There are four 32-bit registers used to control the function of the CorePCIF Master. The first register is the PCI address register. The second register is the RAM or backend address register. These two registers provide the source/destination addressing for all data transfers. The third register contains the number of words to be transferred, and the final control register defines the type and status of a Master transfer. These registers are cleared on reset. They are defined in detail in Table 44 through Table 50.

The DMA registers can be accessed from either the PCI or the backend interface. The address locations for the DMA registers are listed in Table 10. When these registers are accessible from the PCI bus, they can be memory-, I/O-, or configuration-mapped. The DMA_REG_LOC, DMA_REG_BAR, and BACKEND parameters control access to these registers are accessible though BAR 1, a 256-byte memory-mapped BAR.

The complete configuration space can be read when BAR access to these registers is enabled, but writing can be done only to the four DMA control registers.

When the BACKEND parameter is set, the four registers and the complete PCI configuration space can be accessed through the backend (Table 10).

Table 10 • **DMA Register Addresses**

Register Name	Address
PCI address	50h
RAM address or data register	54h
DMA transfer length	58h
DMA control register	5Ch

Downloaded from Arrow.com.

3.6.4 Master Transfers

The CorePCIF Master function supports full DMA transfers to and from the backend interface and initiates direct PCI transfers.

When normal DMA transfers are used, CorePCIF writes each data word to or fetches it from memory through its backend interface. This allows data to be transferred directly from the PCI bus to or from backend memory blocks. In some circumstances, this is inefficient, especially if a processor connected to the backend simply wants to carry out a single-word PCI read or write. In this case, the processor writes the data word to a known location in its memory map. It then programs the DMA controller to perform a single-word DMA transfer. The DMA controller accesses the memory location to obtain the data value; this may require the processor to stop operating while the PCI core accesses the memory to complete the PCI transfer.

When direct DMA transfers are enabled, the processor simply writes the PCI address and data into the core and starts the transfer by writing to the control register, setting the DMA_BAR value to '111'. The core then fetches the data value or writes it to the internal register during the PCI transfer. Access to the backend memory is not required to complete the DMA transfer.

Direct DMA transfer supports only 32-bit transfers. When using 64-bit versions of the core, the 64-bit transfer mode select bit in the DMA control register should not be set if Direct DMA mode is enabled.

3.6.5 Master Byte Commands

CorePCIF can either transfer multiple whole DWORDs (QWORDs for 64-bit transfers) or perform a single DWORD or QWORD transfer with one or more byte enables active.

When multiple words are to be transferred—the DMA transfer length register is greater than four bytes (eight bytes for 64-bit)—the byte enable bits in the DMA control register should be programmed to all ones. All four or eight (64-bit) bytes will be transferred in each data cycle.

If a partial-word read or write is required, the DMA transfer length register should be programmed to four bytes (or eight for 64-bit) and the correct bits set in the byte enable bits in the DMA control register. The DMA engine will transfer a single word, setting the appropriate byte enable bits on both the backend and the PCI interface.

If a non-aligned DMA transfer is required, three separate DMA operations should be performed. The first DMA transfer should be configured to transfer a single DWORD with just the initial bytes enabled. The second DMA should transfer the remaining complete DWORDs with all bytes enabled. A third DMA transfer should transfer the final DWORD with just the remaining bytes enabled. For example, a transfer starting at address 3 and ending at address 12 would require three operations. The first DMA transfer would enable byte 3 only, the second transfer would transfer two DWORD addresses to bytes 4 through 11, and the third DMA transfer would enable byte 0 and transfer address 12.

3.7 CardBus Support

CorePCIF directly supports CardBus functional requirements. Two top-level parameters, CIS_UPPER and CIS_LOWER, specify the 32-bit configuration space setting for the CIS pointer. CIS_UPPER sets the upper 16 bits, and CIS_LOWER sets the lower 16 bits.

The CIS address space must be mapped to one of the BARs or the Expansion ROM. It may not be mapped to configuration space, which means the lower three bits of the CIS pointer (that is, the lower three bits of CIS_LOWER) must not be set to '000'. This allows the user to implement the CIS address space as one of the external backend BAR memory spaces.

When CardBus support is enabled, the IDSEL core input is disabled. CardBus does not require IDSEL to be active for configuration cycles.

3.8 CompactPCI Hot-Swap Support

CorePCIF supports the CompactPCI Hot-Swap PICMG 2.1 R2.0 standard; additional inputs and outputs are provided to support this standard. When enabled, the core includes the hot-swap capabilities register in the configuration space and a state machine that implements the hardware connection process

Downloaded from Arrow.com.

defined in the PICMG Hot-Swap specification. The insertion and extraction sequences are shown in Figure 66 and Figure 67.

3.9 CorePCIF Backend Dataflow

CorePCIF has a very flexible backend interface that supports various transfer rates as well as FIFOs. To decouple the backend data transfers from the PCI transfers, CorePCIF implements an eight-stage FIFO for each BAR. During normal operation, the FIFO stores up to four data words, the remaining four locations being used for the FIFO recovery mechanism. This is implemented using FPGA memory resources in all families except SX-A, RTSX-S, and RTAX-S.

3.9.1 Burst Transfers

CorePCIF is capable of bursting data from the PCI bus to the backend or vice versa. During transfers to the backend, the WR_BE_RDY and WR_BE_NOW signals are used to control the dataflow. When the backend asserts WR_BE_RDY, the core is allowed to write data to the backend by asserting WR_BE_NOW. A separate WR_BE_NOW signal is provided for each byte.

For transfers from the backend, RD_STB_IN and RD_STB_OUT control the dataflow. When both of these signals are active, data is transferred from the backend into the core.

3.9.2 Byte-Controlled Transfers

CorePCIF supports both write- and read-controlled byte transfers to the backend. When data is written to the backend, four (eight for 64-bit operations) write strobes (WR_BE_NOW) are provided, indicating which bytes should be written.

When data is read from the backend interface, the BYTE_ENN and BYTE_VALN signals can be used to control the byte reads. The backend should wait until BYTE_VALN is active (LOW) and then use the four (eight for 64-bit) BYTE_ENN signals (active low) to control the data read. Using the BYTE_VALN signal prevents the core from bursting data every clock cycle; in that case, data will be transferred once every four clock cycles at best.

3.9.3 Dataflow Control

CorePCIF allows the backend to stop data transfers in Master and Target mode, and to initiate transfers in Master mode. In Target mode, the BUSY signal can be used to terminate a data transfer so it will be retried. The ERROR signal can be used to simply terminate a transfer.

Likewise, in Master mode, the STOP_MASTER signal can be used to terminate a data transfer. The WR_BUSY_MASTER and RD_BUSY_MASTER signals can be used to delay a DMA transfer from starting. If STOP_MASTER and RD_BUSY_MASTER are connected to a FIFO empty signal, the DMA engine will automatically stop a DMA cycle when the FIFO becomes empty and restart it when the FIFO becomes non-empty. This allows the core to move data from a FIFO to PCI memory without any host intervention.

3.10 FIFO Recovery Logic

The CorePCIF backend interface directly supports the connection of external FIFOs using internal FPGA FIFO memories or external FIFO devices. To prevent data loss, CorePCIF includes optional FIFO recovery logic for each BAR. In normal burst operations, the core reads data from the backend at the same time as previous data is being transferred on the PCI bus. When the Master terminates the Target transfer, it is likely that data has been read from the FIFO and not transferred on the PCI bus (Figure 15). Without recovery logic, this data would be lost; however, if the FIFO recovery logic is enabled (Figure 24), the core stores this data until the next Target access to the same BAR. Data loss also potentially occurs when the core is operating in Master mode. In this case, the core also needs to recover data lost due to PCI cycles that are terminated with a disconnect without data cycle.

Figure 3 and Figure 4 show how to connect a FIFO to the backend interface, supporting Target and Master transfers. In Target mode, the FIFO empty signal is used to assert the BUSY input while the FIFO is empty and to assert RD_STB_IN when data is available.

In Master mode, the FIFO empty signal is used to assert the RD_BUSY_MASTER input while the FIFO is empty, preventing a DMA cycle from starting, and to assert RD_STB_IN when data is available. The

Downloaded from Arrow.com.

FIFO almost empty signal is used to assert STOP_MASTER, which will cause the current DMA cycle to be terminated as soon as possible. Additional data words may be read from the backend after STOP_MASTER has been asserted.

If both Master and Target transfers will be used, the connections in both Figure 3 and Figure 4 should be implemented.

Figure 3 • **External FIFO Connection (Target mode)**

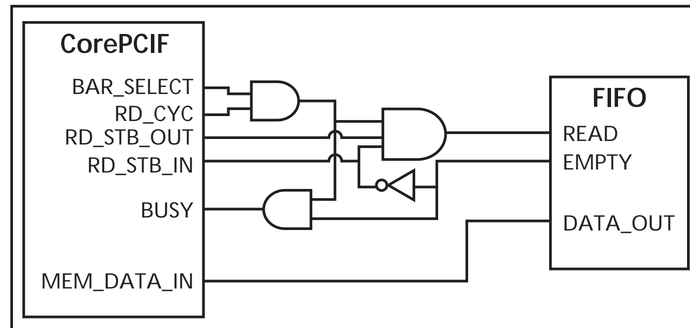
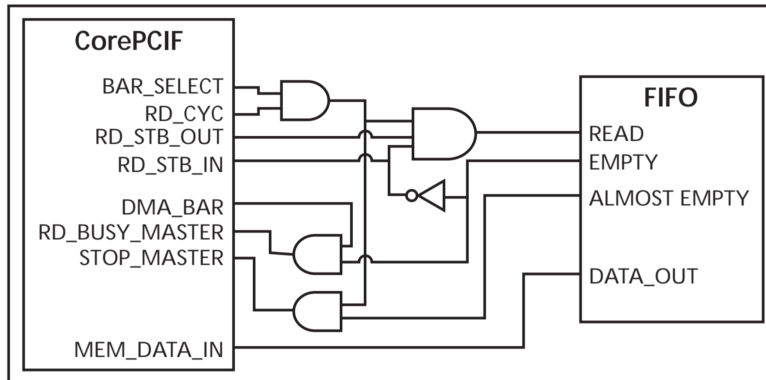


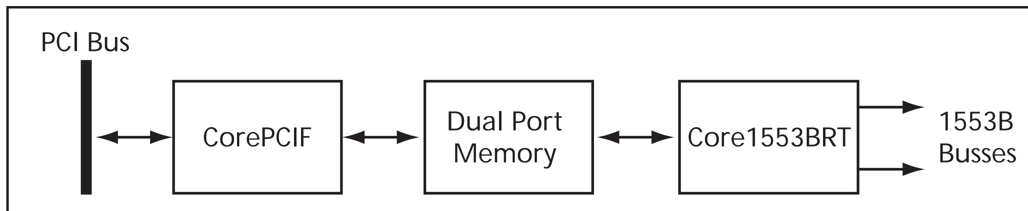
Figure 4 • **External FIFO Connection (Master mode)**



3.11 Example System Implementation

CorePCIF provides an extremely flexible PCI interface that can be configured in many ways. Figure 5 shows a PCI-to-1553 interface. In this example, CorePCIF is configured as a Target with a single memory BAR used to access the Core1553BRT memory.

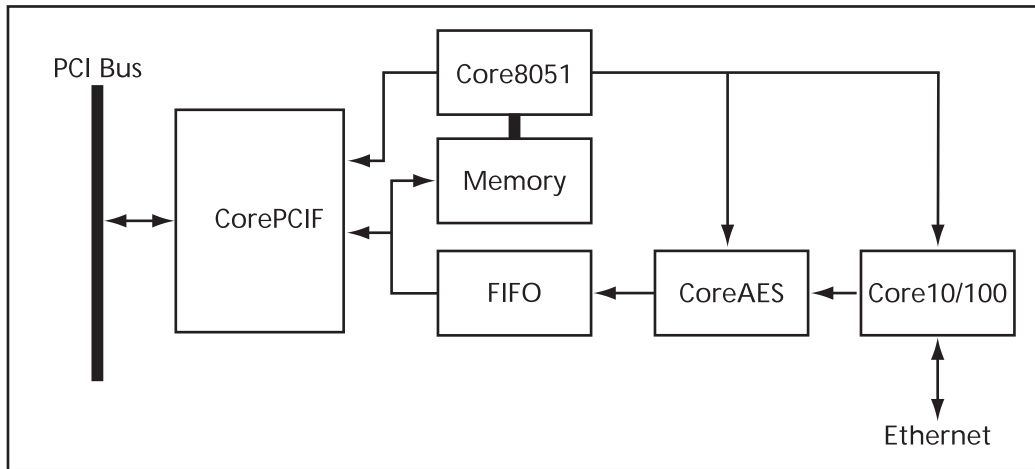
Figure 5 • **Simple Target Implementation**



A more complex system is shown in Figure 6. In this case, the core supports both Target and Master operation. Core8051 is connected to the backend interface, allowing it to initiate PCI cycles. Core8051 is used to control the AES encryption core and the Core10/100 Ethernet interface. CorePCIF has two memory BARs configured. The first allows the PCI interface to access the 8051 memory space, and the second reads data from the FIFO.

Downloaded from Arrow.com.

Figure 6 • **Master and Target Implementation**



Downloaded from Arrow.com.

4 Core Structure

This chapter describes the internal core structure and associated source files.

As shown in Figure 7, the unshaded modules are common to all FPGA families; the shaded modules are specific to each family. This is required to consistently meet the PCI timing constraints. The shaded modules are optimized specifically for each of the supported logic families. The low-level technology cells are used in the higher-level modules.

Figure 7 • **CorePCIF Structure**

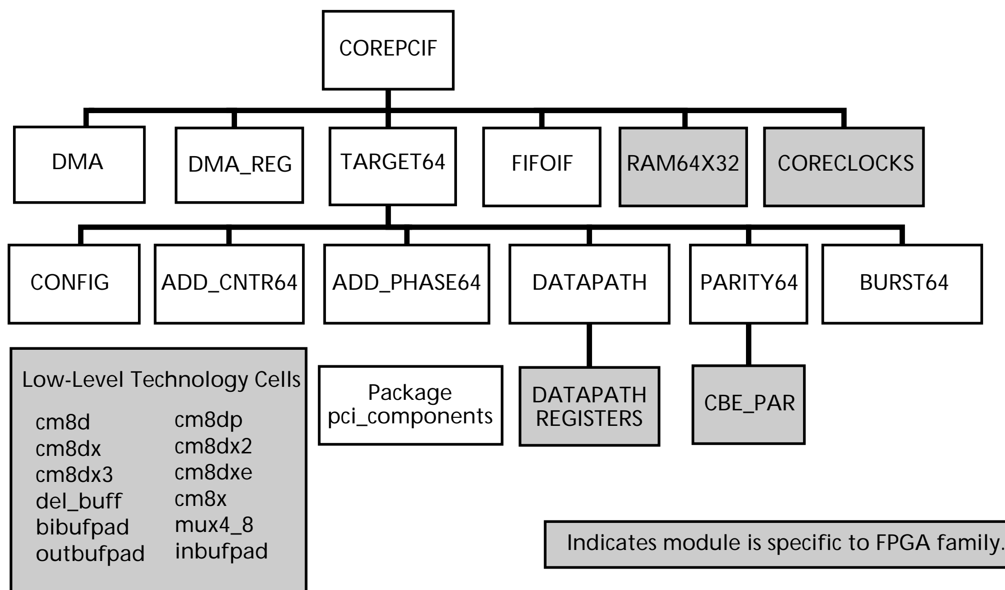


Table 11 provides details for each of the common source modules and the functions they implement. Table 12 provides the details of the FPGA technology files. Table 12 provides details of a few high-level source files not directly used by the core but provided to create test databases.

Table 11 • **CorePCIF Common Source Files**

Common Files	Description
corepcif	This is the top level of the core. It includes all the top-level ports. Parameters will enable and disable the Target, Master, and backend functions as well as switching between 32- and 64-bit operation. This is the CorePCIF top level upon which this core is built.
pci_components	This is the VHDL package that contains the component declarations used within the core along with some common type conversion functions.
fifoif	This is the control logic that manages internal data storage and performs FIFO recovery cycles.
target64	This is the top level of the main PCI Target function.
burst64	This is the main control logic used to handle the PCI protocol and manage data transfers to and from the PCI bus.
add_cntr64	This is the main address counter that tracks both the current PCI and backend addresses.
add_phase64	This block compares the PCI address during an address phase to detect whether the configuration space or one of the BARs on the Target is being addressed. It also contains the logic to detect BAR overflows so a Target disconnect can be triggered.

Downloaded from Arrow.com.

Table 11 • **CorePCIF Common Source Files**

Common Files	Description
config	This block contains the registers required to implement PCI configuration space.
datapath	This block routes the data between the backend interface or memory buffer and the PCI bus. It provides a data storage register used to recover when transferred data are stalled.
parity64	This block creates the top-level structure for parity generation and checking. The parity generation and checking is done using the cbe_par module.
dma	This is the main Master control logic. It contains state machines and counters that control the DMA engine and initiate PCI cycles.
dma_reg	This module contains the four DMA control registers and the main DMA transfer counters.

Table 12 • **Technology-Specific Source Files**

FPGA Specific Files	Description
bibufpad	This module contains a bidirectional I/O pad.
cbe_par	This block implements a PCI parity generator and checker. The 36-input parity tree is hand-optimized to obtain the most efficient implementation for each FPGA family.
cm8d	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with clear.
cm8dp	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with preset.
cm8dx	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with clear.
cm8dx2	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with clear, with some inputs tied or shared.
cm8dx3	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with clear, with some inputs tied or shared.
cm8dxe	This is a low-level FPGA technology cell implementing a four-input multiplexer and register with enable and clear.
cm8x	This is a low-level FPGA technology cell implementing a four-input multiplexer.
coreclocks	This module contains the global and clock buffers.
datapath_registers	This module implements the datapath registers used to interface to the PCI bus.
del_buff	This module contains a delay element used to insert delays to control the PCI hold times. The amount of inserted delay for all critical PCI input paths can be adjusted in this file.
family	This is a VHDL package or Verilog include file that sets the FPGA family to enable some family-specific optimizations.
inbufpad	This module contains an input I/O pad.
mux4_8	This is a low-level FPGA technology cell implementing eight four-input multiplexers.
outputpad	This module contains an output I/O pad.
ram64x32	This module contains a 64×32 RAM using the appropriate FPGA memory blocks.

Table 13 • **CorePCIF Miscellaneous Source Files**

Miscellaneous Files	Description
pcicoretest	This is a top-level wrapper module that creates a simple top-level design with just the PCI I/O pins used for creating the example layout databases in the release. It connects all PCI interface signals to top-level ports, and then all interface signals to the loopback module.

Downloaded from Arrow.com.

Table 13 • **CorePCIF Miscellaneous Source Files**

Miscellaneous Files	Description
loopback	This module is used in the example database designs. It connects core backend output signals to input signals. This removes the need for the backend signals to be connected to FPGA I/O pins when creating the example designs, allowing the core to be placed and routed in small pinout packages.
pcicore_components	This is a VHDL components package that contains the PCI core component declaration.

Downloaded from Arrow.com.

5 Tool Flows

5.1 SmartDesign

Note: CorePCIF is compatible with Libero Integrated Design Environment (IDE), Libero System-on-Chip (SoC) and Libero System-on-Chip (SoC) PolarFire. Unless specified otherwise, this document uses the name Libero to identify Libero IDE, Libero SoC and Libero SoC PolarFire.

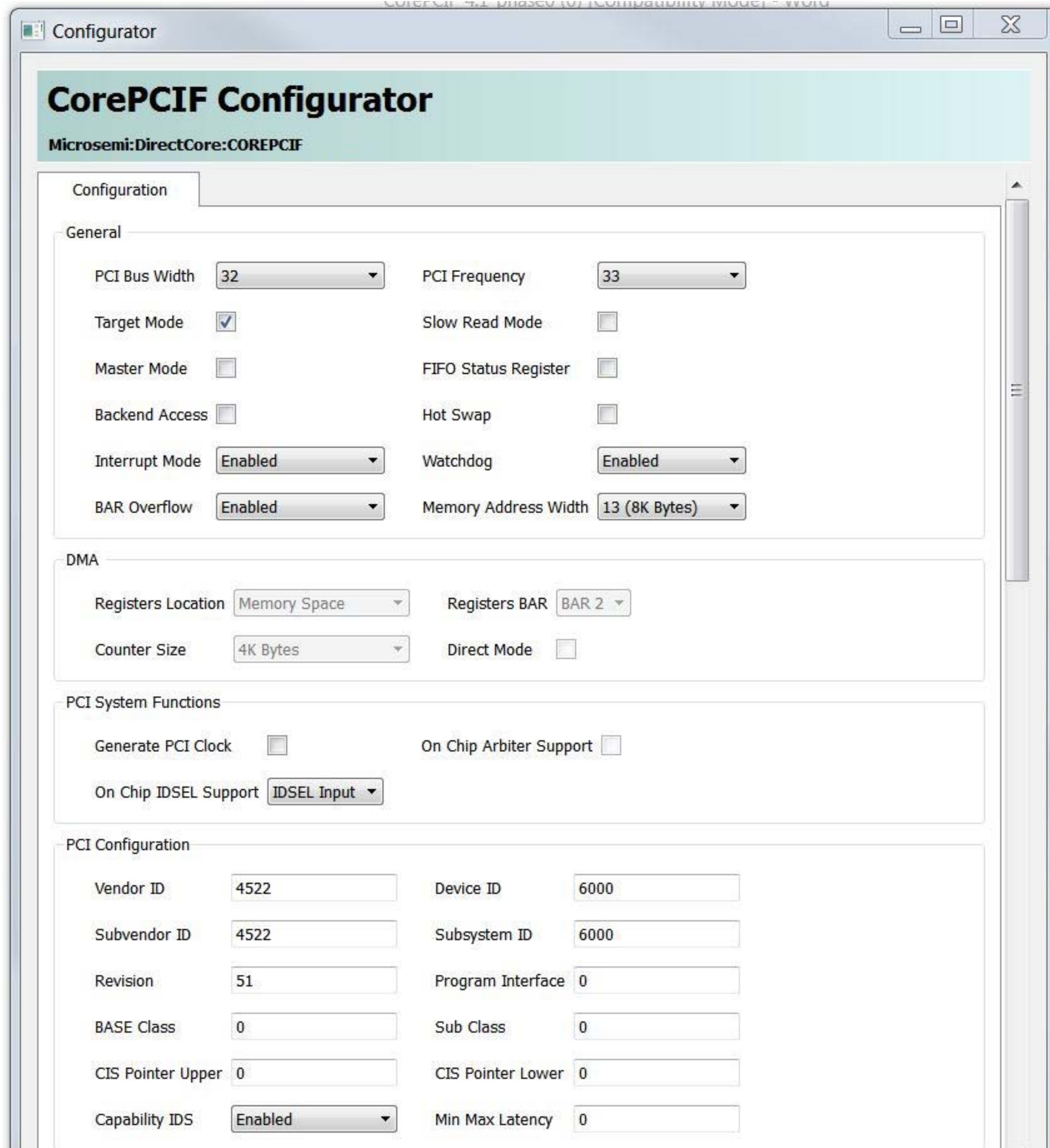
CorePCIF is pre-installed in the SmartDesign IP deployment design environment or downloaded from the online repository. Figure 8 shows an example instantiated.

Figure 8 • **CorePCIF Full I/O View**

The core can be configured using the configuration GUI within SmartDesign, as shown in Figure 9 and Figure 10.

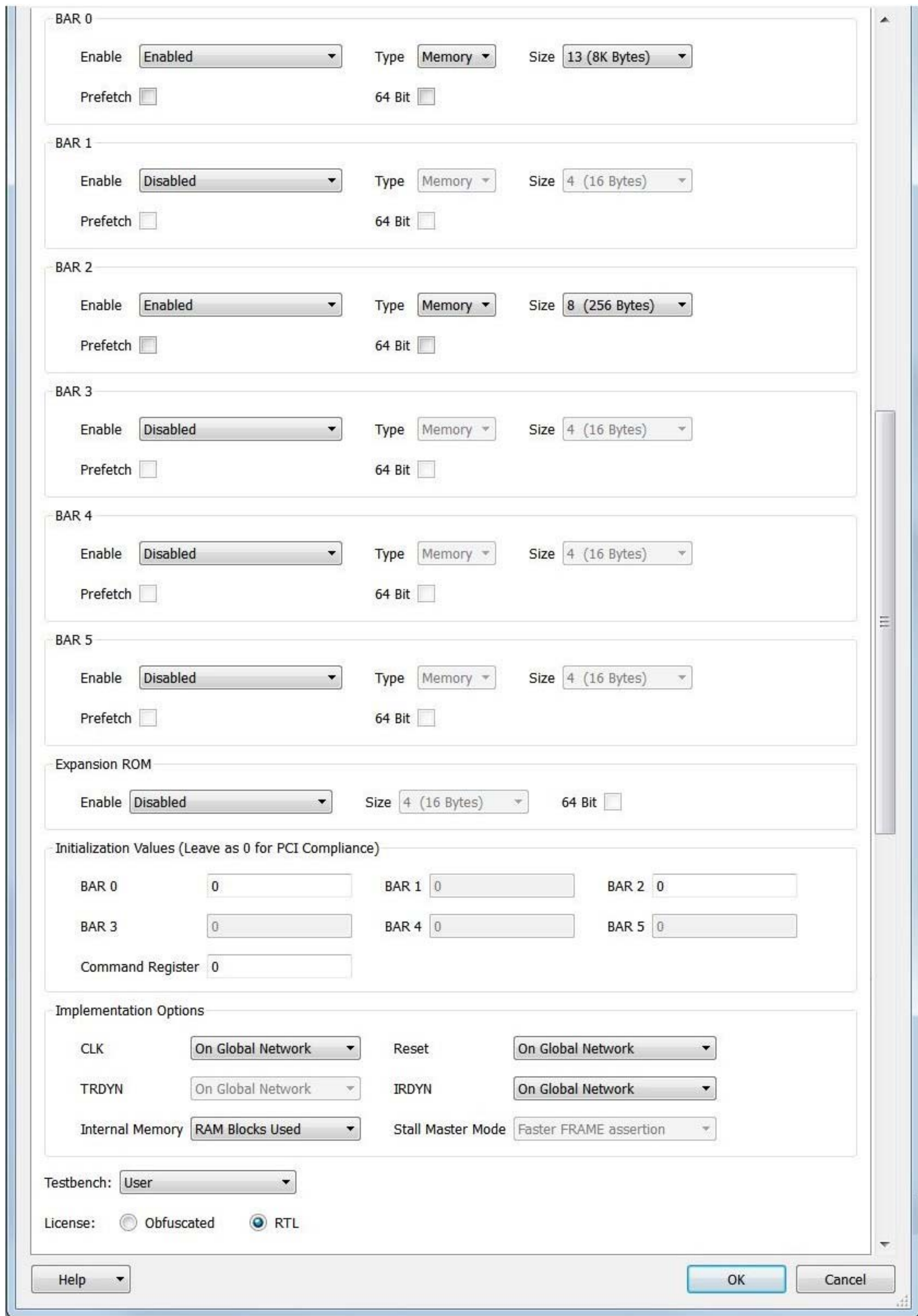
Downloaded from Arrow.com.

Figure 9 • **CorePCIF Configurator**



Downloaded from Arrow.com.

Figure 10 • **CorePCIF Configurator (Continued)**



5.2 Synthesis in Libero

To run Synthesis on the core, **set the design root to the top of the project**. This is a wrapper around the core that sets all the generics appropriately.

Downloaded from Arrow.com.

Note: Make sure the required timing constraints files are associated with the synthesis tool. Use sdc constraint files with suffix “_synplicity” on Libero 9.2 or before versions, and with suffix “_designer” on Libero versions later to 9.2.

Synthesis Timing Constraints details the timing constraints that are required.

Click the **Synthesis** icon in Libero. The synthesis window appears, displaying the Synplicity® project. To run Synthesis, click the **Run** icon.

To allow the core to be synthesized standalone within Libero, an additional top-level module, PCICORETEST, is also imported into Libero. The top level only includes the PCI bus signals; all the core backend signals are connected to a loopback module. This allows the core to synthesized and place-and-route performed in small-pin-count packages so that utilization and performance can be verified without user logic being connected to the backend interface. To use this top level, the design root must be set to **PCICORETEST** in Libero.

Additionally, the *PCISYSTEM* or *PCISYSTEM2* stimulus file can be moved to the HDL directory; these files add fully functioning memory and FIFO to the backend of the core (Axcelerator, IGLOO/e, ProASIC3/E/L, and Fusion FPGA families only), creating a single-chip PCI system (**see User Testbench**). To synthesize the PCISYSTEM design, move the *pcisystem*, *fifo*, *memory*, *ram2k8*, and *fifo512x32* files from the CorePCI stimulus directory to the HDL source files directory in Libero, and then set the design root to **PCISYSTEM**. For the PCISYSTEM2 design, move the *pcisystem2*, *memory*, and *ram2k8* files.

5.3 Place-and-Route in Libero

Make sure required timing and physical constraints files are associated with the place-and-route tool. There should be multiple timing and physical constraints files available, covering the PCI functions: 33/66 MHz and 32-/64-bit operation as well as device package combinations.

For Target-only cores, the *T_* files should be used; for Master-only cores, the *M_* files should be used; and for Target and Master operation, the *TM_* files should be used.

32/64 selects 32- or 64-bit PCI operation.

33/66 selects 33 or 66 MHz PCI operation.

The supplied timing constraints files assume a typical configuration of the core. Some configurations may cause the timing constraints files to cause an error when loaded by Designer. For example, if the ONCHIP_IDSEL function is enabled, the IDSEL input is not used and Synthesis will remove the IDSEL input. When this occurs, Designer will detect an error when it tries to set a timing constraint on the nonexistent IDSEL input. In this case, the timing constraints on the IDSEL input should be removed from the SDC files.

If there is not a good match for your selected device and package, you should create your own physical constraints files, following the rules in the **Implementation Hints**.

Having set the design root appropriately and run Synthesis, click the **Layout** icon in Libero to invoke Designer. CorePCIF requires no special place-and-route settings. Microsemi recommends you set the compile options given in Table 14.

Table 14 • **Designer Compile Options**

Device Family	Compile Option(s)
ProASICPLUS	No special requirements
Fusion IGLOO/e ProASIC3/E/L	Set pdc_abort_on_error “off” Set pdc_eco_display_unmatched_objects “off” Set demote_globals “off” -promote_globals “off” Set combine_register “on” -delete_buffer_tree “off” Set report_high_fanout_nets_limit 10
Axcelerator	Set combine_register = 1
RTAX-S	Set combine_register = 1

Downloaded from Arrow.com.

Table 14 • **Designer Compile Options**

Device Family	Compile Option(s)
SX-A	No special requirements
RTSX-S	No special requirements

Downloaded from Arrow.com.

6 Configuring Parameters

CorePCIF is a highly configurable core. The configuration is controlled by approximately 5025 top-level parameters. These are listed in Table 15 to Table 18.

6.1 General Configuration Parameters

Table 15 shows general configuration parameters for CorePCIF.

Table 15 • **General Parameters**

Name	Values	Description
FAMILY	-	Must be set to the required FPGA family: 8: SX-A 9: RTSX-S 11: Axcelerator 12: RTAX-S 14: ProASIC<u>PLUS</u> 15: ProASIC3 16: ProASIC3E 17: Fusion 18: SmartFusion 19: SmartFusion2 20: IGLOO 21: IGLOOe 22: ProASIC3L 24: IGLOO2 25: RTG4 26: PolarFire
MASTER	0 or 1	When 1, the PCI Master function with DMA controller is implemented.
TARGET	0 or 1	When 1, the PCI Target function is implemented.
PCI_FREQ	33 or 66	When 66, the 66 MHz bit in the PCI configuration space is set.
SLOW_READ	0 or 1	When 1, the core inserts either one or two wait states in all read transfersone wait state in all PCI read transfers, eliminating the requirement for internal data storage within the corethe two internal memories within CorePCIF. This parameter must not be set if the FIFO recovery option is enabled.
PCI_WIDTH	32 or 64	Sets 32- or 64-bit PCI implementation.
DISABLE_WDOG	0 or 1	When 1, the data transfer watchdog inside the core is disabled. The core normally includes a transfer watchdog that will terminate a PCI cycle if the backend logic fails to provide or accept data within the time limits defined by the PCI specification. This function can be disabled in embedded systems if longer access times are permitted.
DISABLE_BAROV	0 or 1	When 1, the core will not disconnect when a memory or I/O transfer overflows the BAR as required by the PCI specification. Instead, the core will wrap the address and jump to the beginning of the BAR space. Setting the parameter to 1 will reduce the number of logic elements in the core. When the BAR overflow logic is enabled, the core may disconnect burst transfers before they reach the upper limit of the BAR, depending on the transfer rate controlled by IRDY and TRDY. This may require the PCI Master to perform several separate PCI transfers before the top of memory is reached.

Downloaded from Arrow.com.

Table 15 • General Parameters (continued)

Name	Values	Description
REMOVE_CAP_ID	0 or 1	When 1, the capability pointer and capability values in the PCI configuration space are all held at 0. The interrupt and DMA control registers are still accessible at locations 48 and 50–5C hex. When 0, the capability IDs.
INTERRUPT_MODE	0 to 2	Configures the PCI interrupt: 0: The interrupt register is implemented. 1: The interrupt system is disabled. 2: The interrupt register (48h) is not implemented, and the EXT_INTn input directly drives INTAn. When Master functions are enabled (MASTER = 1), this parameter should be set to 0. When INTERRUPT_MODE is set to 0 or 2, the interrupt disable and status bits in the configuration space control and status registers are implemented and may be used to disable the interrupt.
ENABLE_FIFOSTAT	0 to 1	When 1, the FIFO status register is implemented.
USE_REGISTERS	0 or 1	When 1, the internal RAM blocks are replaced with a register-based implementation. For SX-A and RTSX-S, the internal RAM blocks are always replaced with registers.
MADDR_WIDTH	8 to 32	Specifies the width of the backend address bus. This should match the largest BAR address width. For example, if 64 kB of address space are configured, MADDR_WIDTH should be set to 16. If all BARs are less than 256 bytes, MADDR_WIDTH should be set to 8. Values below 8 are not permitted. Memory Size = 2^{MADDR_WIDTH}
GENERATE_PCICLK	0 or 1	Set to 1 when the core is required to generate the PCI clock.
USE_GLOBAL_CLK	0, 1, or 2	Controls the sort of clock buffer used. 0: No buffer is used. The synthesis tool will insert a buffer. 1: A standard clock buffer is used. 2: When using AX/RTAX-S families, uses a CLKBUF instead of a HCLKBUF cell Microsemi recommends setting this parameter to 1.
USE_GLOBAL_RESET	0 or 1	When 1, a global buffer is used to drive the internal reset network in the core. When 0, normal routing resources are used, and due to the high fanout of the reset network, a buffer tree will be created for it. Microsemi recommends that this parameter be set to 1.
USE_GLOBAL_TRDY	0 or 1	When 1 and MASTER = 1, a global buffer is used to drive the internal TRDY network in the core. When 0, normal routing resources are used. Microsemi recommends that this parameter be set to 1.
USE_GLOBAL_IRDY	0 or 1	When 1 and TARGET = 1, a global buffer is used to drive the internal IRDY network in the core. When 0, normal routing resources are used. Microsemi recommends that this parameter be set to 1.
ONCHIP_ARBITER	0 or 1	In some applications, the FPGA will be the system controller as well, and include the PCI arbiter. When 1, this removes the pads from the REQN and GNTN I/Os, allowing connection inside the FPGA and enabling the FRAMEN_OUT and IRDYN_OUT outputs.
ONCHIP_IDSEL	0 to 31	In some applications, the FPGA will be the system controller as well, and include the IDSEL decoding. When *value* is non-zero, the IDSEL input is directly driven by the AD *(value*) output. When 0, IDSEL is driven from the IDSEL input.

Downloaded from Arrow.com.

Table 15 • **General Parameters *(continued)***

Name	Values	Description
STALL_MODE	0 or 1	Controls the speed of FRAME and IRDY assertion when STALL_MASTER is used. 0: FRAME will be asserted followed by IRDY two clocks later. 1: FRAME assertion will be delayed by an additional clock cycle and IRDY asserted one clock later. IRDY assertion assumes that the data fetch from the backend has completed whilst STALL_MASTER was asserted.

6.2 PCI Configuration Space Parameters

Table 16 • **PCI Configuration Space Parameters**

Name	Values	Description
VENDOR_ID	0 to 65,535	Sets the user vendor ID value in the PCI configuration space. The Microsemi Vendor ID is 4522 (11AAh) and may be used with permission. Microsemi will allocate a device ID and sub-vendor ID on demand. Contact Technical Support to request this service.
DEVICE_ID	0 to 65,535	Sets the user device ID value in the PCI configuration space.
REVISION_ID	0 to 255	Sets the user revision ID value in the PCI configuration space.
BASE_CLASS	0 to 255	Sets the user base class value in the PCI configuration space.
SUB_CLASS	0 to 255	Sets the user subclass value in the PCI configuration space.
PROGRAM_IF	0 to 255	Sets the user program interface value in the PCI configuration space.
SUBVENDOR_ID	0 to 65,535	Sets the user subvendor ID value in the PCI configuration space.
SUBSYSTEM_ID	0 to 65,535	Sets the user subsystem ID value in the PCI configuration space.
CIS_UPPER	0 to 65,535	Sets the value of the upper 16 bits of the CardBus CIS pointer.
CIS_LOWER	0 to 65,535	Sets the value of the lower 16 bits of the CardBus CIS pointer.
ENABLE_HOT_SWAP	0 or 1	Enables the hot-swap register and functionality.
MINMAXLAT	0 to 65,535	Sets the minimum grant and maximum latency values at location 3Eh in the configuration space.
CMD_INITVAL	0 to 65,335	Sets the value of the PCI command register at reset. For PCI compliance, this should be set to zero.

6.3 BAR Parameters

CorePCIF supports up to six BARs and the Expansion ROM address register. Enabling all the BARs will have a significant effect on logic utilization. For the SX-A and RTSX-S families, only BAR 0 and BAR 1 can be used to access backend memory. BAR 2 can be used to access the DMA registers. BARs 3 to 5 and the Expansion ROM are not supported in the SX-A and RTSX-S families. Table 17 displays BAR parameters. The variable *i* can have a value from 0 to 5.

Table 17 • **BAR Parameters**

Name	Values	Description
BAR*i*_ENABLE	0 to 2	0: BAR *i* is disabled. 1: BAR *i* is enabled without FIFO recovery. 2: BAR *i* is enabled with FIFO recovery.

Downloaded from Arrow.com.

Table 17 • **BAR Parameters**

Name	Values	Description
BAR*i*_ADDR_WIDTH	4 to 32	Specifies the width of the BAR. A value of 8 would create a 256-byte address space. If the BAR is disabled, this should be set to 4. BAR_SIZE = $2^{BARi_ADDR_WIDTH}$
BAR*i*_IS_IO	0 or 1	0: BAR *i* is configured as memory space. 1: BAR *i* is configured as I/O space.
BAR*i*_PREFETCH	0 or 1	If BAR *i* is memory space, this bit controls the PREFETCH bit in the BAR. 0: Prefetch is disabled for BAR *i*. 1: Prefetch is enabled for BAR *i*. This should be set to zero when the FIFO recovery logic is enabled.
BAR*i*_64BIT	0 or 1	0: BAR *i* supports only 32-bit transfers. 1: BAR *i* supports 32- and 64-bit transfers (PCI_WIDTH must also be set to 64).
BAR*i*_INITVAL	0 to 268,435,455	Specifies the reset value of the upper 28 bits of the BAR at reset. For PCI compliance, this should be set to zero. If non-zero, BAR*i*_INITVAL allows the core to respond to PCI accesses without the BAR being programmed. If the BAR initialization value is required to be 0x80001000, the parameter should be set to the required value divided by 16 (that is, 0x08000100). The division is required because the value provided is used to set the upper 28 bits. The lowest 4 bits are set depending on the BAR*i*_IS_IO and BAR*i*_PREFETCH values.
EXPR_ENABLE	0 to 1	0: Expansion ROM is disabled. 1: Expansion ROM is enabled.
EXPR_ADDR_WIDTH	4 to 32	Specifies the width of the Expansion ROM register. A value of 8 would create a 256-byte address space. If the Expansion ROM is disabled, this should be set to 4.
EXPR_64BIT	0 or 1	0: Expansion ROM supports only 32-bit transfers. 1: Expansion ROM supports 32- and 64-bit transfers (PCI_WIDTH must also be set to 64).

If a memory or I/O BAR is used for the DMA registers (DMA_REG_LOC set to 2 or 3, Table 18), the BAR specified by DMA_REG_BAR (Table 18) should be configured as follows:

BAR*i*_ENABLE = 1

BAR*i*_ADDR_WIDTH = 8

BAR*i*_IS_IO = 0 (memory BAR) or 1 (I/O BAR)

BAR*i*_PREFETCH = 0

This BAR will be used solely to access the DMA control registers and configuration space. It cannot be used to access user logic connected to the backend of the core.

Downloaded from Arrow.com.

6.4 Master/DMA Parameters

Table 18 • **Master/DMA Parameters**

Name	Values	Description
BACKEND	0 or 1	When 1, the backend interface to the DMA control registers is enabled. When 0, the DMA registers can only be accessed from the PCI bus. If BACKEND = 1 and DMA_REG_LOC > 0, the DMA control registers can be accessed from both the PCI and backend interfaces.
DMA_REG_LOC	0 to 3	Configures how the DMA control registers are accessed from the PCI bus. 0: None – DMA registers can be read at locations 50-5F hex of the configuration space, but not written. Register read/writes are expected to be from the backend interface. (The BACKEND parameter must be set to 1 for backend access.) 1: Config – DMA registers are mapped to locations 50–5F hex of the configuration space. 2: MEM – DMA registers are mapped to configuration space and memory locations 50–5F hex of the BAR set by DMA_REG_BAR. 3: I/O – DMA registers are mapped to configuration space and I/O locations 50–5F hex of the BAR set by DMA_REG_BAR.
DMA_REG_BAR	0 to 5	Sets which BAR is used to access the DMA registers if DMA_REG_LOC is set to 2 or 3. The BAR parameters must be set up to configure a 256-byte BAR, either memory- or I/O-mapped with prefetch disabled. This BAR is in addition to other memory and I/O BARs being used. In other words, the BAR used for DMA registers may not be shared with other memory and I/O BARs used to access user logic connected to the core.
DMA_COUNT_WIDTH	8 to 32	Sets the width of the internal DMA counter. For example, if DMA_COUNT_WIDTH is set to 12, the DMA engine can transfer up to 4,096 bytes of data. Max Transfer Size = $2^{DMA_COUNT_WIDTH}$
ENABLE_DIRECTDMA	0 or 1	Enables core support for direct DMA operations. When 1, direct DMA mode is enabled, allowing the PCI data value to be read from and written to an internal register rather than the backend interface.

6.5 Default Core Parameter Settings

Table 19 details the parameter settings used to create the eight example builds in **Utilization Statistics**.

Table 19 • **Default Build Parameters**[1]

Parameter	SMALL32	TARG32	MAST32	TARG DMA32	TARG MAST32	TARG64	MAST64	TARG DMA64	TARG MAST64
TARGET	1	1	0	1	1	1	0	1	1
MASTER	0	0	1	1	1	0	1	1	1
BACKEND	0	0	1	0	1	0	1	0	1
SLOW_READ	1	0	0	0	0	0	0	0	0
PCI_WIDTH	32	32	32	32	32	64	64	64	64

Note: For SX-A and RTSX-S builds, only BAR 1 and BAR 2 are configured. All other BARs are disabled.

Note: For RTAX-S builds, USE_REGISTERS is set to 1.

Downloaded from Arrow.com.

Table 19 • **Default Build Parameters[1]** ***(continued)***

Parameter	SMALL32	TARG32	MAST32	TARG DMA32	TARG MAST32	TARG64	MAST64	TARG DMA64	TARG MAST64
PCI_FREQ	33 or 66	33 or 66	33 or 66	33 or 66	33 or 66	33 or 66	33 or 66	33 or 66	33 or 66
DISABLE_WDOG	1	0	1	0	0	0	1	0	0
DISABLE_BAROV	1	1	1	1	0	1	1	1	0
ENABLE_FIFOSTAT	0	0	0	1	0	0	0	0	1
ENABLE_HOT_SWAP	0	0	0	0	1	0	0	0	1
MADDR_WIDTH	16	16	16	16	20	16	16	16	20
USER_VENDOR_ID	4,522	4,522	4,522	4,522	4,522	4,522	4,522	4,522	4,522
USER_DEVICE_ID	A different ID value is used for each build and family.								
USER_REVISION_ID	30	30	30	30	30	30	30	30	30
USER_BASE_CLASS	255	255	255	255	255	255	255	255	255
USER_SUB_CLASS	0	0	0	0	0	0	0	0	0
USER_PROGRAM_IF	0	0	0	0	0	0	0	0	0
USER_SUBVENDOR_ID	A different ID value is used for each build and family.								
USER_SUBSYSTEM_ID	A different ID value is used for each build and family.								
CIS_UPPER	0	0	0	0	0	0	0	0	0
CIS_LOWER	0	0	0	0	0	0	0	0	0
BAR0_ENABLE	1	1	1	1	2	1	1	1	2
BAR0_ADDR_WIDTH	16	16	16	16	20	16	16	16	20
BAR0_ISIO	0	0	0	0	0	0	0	0	0
BAR0_PREFETCH	1	1	1	1	0	1	1	1	0
BAR0_64BIT	0	0	0	0	0	1	1	1	1
BAR1_ENABLE	0	0	0	0	2	0	0	0	2
BAR1_ADDR_WIDTH	4	4	4	4	16	4	4	4	16
BAR1_ISIO	0	0	0	0	0	0	0	0	0
BAR1_PREFETCH	0	0	0	0	0	0	0	0	0
BAR1_64BIT	0	0	0	0	0	1	1	1	1
BAR2_ENABLE	0	0	0	0	2	0	0	0	2
BAR2_ADDR_WIDTH	4	4	4	4	10	4	4	4	10
BAR2_ISIO	0	0	0	0	0	0	0	0	0
BAR2_PREFETCH	0	0	0	0	0	0	0	0	0
BAR2_64BIT	0	0	0	0	0	1	1	1	1
BAR3_ENABLE	0	0	0	0	2	0	0	0	2
BAR3_ADDR_WIDTH	4	4	4	4	10	4	4	4	10

Note: For SX-A and RTSX-S builds, only BAR 1 and BAR 2 are configured. All other BARs are disabled.

Note: For RTAX-S builds, USE_REGISTERS is set to 1.

Downloaded from Arrow.com.

Table 19 • **Default Build Parameters[1] *(continued)***

Parameter	SMALL32	TARG32	MAST32	TARG DMA32	TARG MAST32	TARG64	MAST64	TARG DMA64	TARG MAST64
BAR3_ISIO	0	0	0	0	0	0	0	0	0
BAR3_PREFETCH	0	0	0	0	0	0	0	0	0
BAR3_64BIT	0	0	0	0	0	1	1	1	1
BAR4_ENABLE	0	0	0	0	1	0	0	0	1
BAR4_ADDR_WIDTH	4	4	4	4	8	4	4	4	8
BAR4_ISIO	0	0	0	0	1	0	0	0	1
BAR4_PREFETCH	0	0	0	0	0	0	0	0	0
BAR4_64BIT	0	0	0	0	0	1	1	1	1
BAR5_ENABLE	0	0	0	0	1	0	0	0	1
BAR5_ADDR_WIDTH	4	4	4	4	8	4	4	4	8
BAR5_ISIO	0	0	0	0	0	0	0	0	0
BAR5_PREFETCH	0	0	0	0	0	0	0	0	0
BAR5_64BIT	0	0	0	0	0	1	1	1	1
EXPR_ENABLE	0	0	0	0	1	0	0	0	1
EXPR_ADDR_WIDTH	4	4	4	4	16	4	4	4	16
EXPR_64BIT	0	0	0	0	0	0	0	0	1
ENABLE_DIRECTDMA	0	0	1	0	1	0	1	0	1
DMA_REG_LOC	0	0	0	1	2	0	0	1	2
DMA_REG_BAR	0	0	0	0	5	0	0	0	5
DMA_COUNT_WIDTH	0	0	16	12	16	0	16	12	16
CMD_INITVAL	0	0	0	0	0	0	0	0	0
BAR0_INITVAL	0	0	0	0	0	0	0	0	0
BAR1_INITVAL	0	0	0	0	0	0	0	0	0
BAR2_INITVAL	0	0	0	0	0	0	0	0	0
BAR3_INITVAL	0	0	0	0	0	0	0	0	0
BAR4_INITVAL	0	0	0	0	0	0	0	0	0
BAR5_INITVAL	0	0	0	0	0	0	0	0	0
REMOVE_CAPID	1	0	0	0	0	0	0	0	0
INTERRUPT_MODE	1	0	0	0	0	0	0	0	0
USE_REGISTERS[2]	0	0	0	0	0	0	0	0	0
USE_GLOBAL_CLK	1	1	1	1	1	1	1	1	1
USE_GLOBAL_RESET	1	1	1	1	1	1	1	1	1
GENERATE_PCICLK	0	0	0	0	0	0	0	0	0

Note: For SX-A and RTSX-S builds, only BAR 1 and BAR 2 are configured. All other BARs are disabled.

Note: For RTAX-S builds, USE_REGISTERS is set to 1.

Downloaded from Arrow.com.

Table 19 • Default Build Parameters[1] *(continued)*

Parameter	SMALL32	TARG32	MAST32	TARG DMA32	TARG MAST32	TARG64	MAST64	TARG DMA64	TARG MAST64
ONCHIP_ARBITER	0	0	0	0	0	0	0	0	0
ONCHIP_IDSEL	0	0	0	0	0	0	0	0	0
STALL_MODE	0	0	0	0	0	0	0	0	0

Note: For SX-A and RTSX-S builds, only BAR 1 and BAR 2 are configured. All other BARs are disabled.

Note: For RTAX-S builds, USE_REGISTERS is set to 1.

Downloaded from Arrow.com.

7 Core Interfaces

7.1 PCI Bus Signals

Table 20 lists the signals used in the PCI interface. The "Used On" column indicates which core configurations use each signal. For example, REQN is only used on PCI Master cores (M), IDSEL is only used on PCI Target cores (T), and PAR64 is only used on 64-bit cores. Note that the "Type" column describes the characteristics of the signal in the PCI specification, not necessarily its usage in a particular core. For example, TRDYN is bidirectional for PCI Target/Master cores but only an input to PCI Target cores.

Table 20 • **PCI Bus Interface Signals**

Name	Type	Used On	Description
PCICLK	Bidirectional	All	33 MHz or 66 MHz clock input or output for the PCI core
PCIRSTN	Input	All	Active low asynchronous reset
IDSEL	Input	T	Active high Target select used during configuration read and write transactions
AD	Bidirectional	All	Multiplexed 32-bit or 64-bit address and data bus. Valid address is indicated by FRAMEN assertion.
CBEN	Bidirectional	All	Bus command and byte enable information. During the address phase, the lower four bits define the bus command. During the data phase, they define the byte enables (active high). This bus is 4 bits wide in 32-bit PCI systems and 8 bits wide in 64-bit systems.
PAR	Bidirectional	All	Parity signal. Parity is even across AD[31:0] and CBE[3:0].
FRAMEN	Bidirectional (STS)	All	Active low signal indicating the beginning and duration of an access. While FRAMEN is asserted, data transfers continue.
DEVSELN	Bidirectional (STS)	All	Active low output from the Target indicating that it is the Target of the current access
IRDYN	Bidirectional (STS)	All	Active low signal indicating that the bus Master is ready to complete the current dataphase transaction
TRDYN	Bidirectional (STS)	All	Active low signal indicating that the Target is ready to complete the current dataphase transaction
STOPN	Bidirectional (STS)	All	Active low signal from the Target requesting termination of the current transaction
PERRN	Bidirectional (STS)	All	Active low parity error signal
SERRN	Bidirectional (OD)	T	Active low system error signal. This signal reports PCI address parity errors.
REQN	Output	M	Active low output used by the PCI Master controller to request bus ownership
GNTN	Input	M	Active low input from the system arbiter indicating that the core may claim bus ownership
INTAN	Bidirectional	TAll	Active low interrupt input and request
INTBN	Input	All	Active low input interrupt
INTCN	Input	All	Active low input interrupt

Note: Active low signals are designated with a trailing uppercase N.

Downloaded from Arrow.com.

Table 20 • PCI Bus Interface Signals (continued)

Name	Type	Used On	Description
INTDN	Input	All	Active low input interrupt
PAR64	Bidirectional	64	Upper parity signal. Parity is even across AD[63:32] and CBE[7:4]. This signal is not required for 32-bit PCI systems.
REQ64N	Bidirectional (STS)	64	Active low signal with the same timing as FRAMEN indicating that the Master requests a data transfer over the full 64-bit bus. This signal is not required for 32-bit PCI systems.
ACK64N	Bidirectional (STS)	64	Active low output from the Target indicating that it is capable of transferring data on the full 64-bit PCI bus. This signal is driven in response to the REQ64N signal and has the same timing as DEVSELN. This signal is not required in 32-bit PCI systems.
M66EN	Bidirectional (OD)	All	Active high signal indicating that the core supports 66 MHz operation. This output will be driven LOW if the MHZ_66 parameter is NOT set. When it is set, the output is tristated. If hot-swap is enabled, this is also used as an input to verify the PCI clock frequency during hot-swap insertion cycles. The M66EN PCI signal pin should be pulled up with a 5 kΩ resistor.

Note: Active low signals are designated with a trailing uppercase N.

7.2 Backend System-Level Signals

CorePCIF buffers the PCI clock and reset internally onto global networks. The outputs shown in Table 21 allow these global networks to be used for additional user backend logic.

Table 21 • System-Level Signals

Name	Type	Width	Description
CLK_OUT	Output	1	Clock output. The core uses an internal clock buffer. This is the buffered version of the clock and should be used for clocking any other logic in the FPGA that is clocked by the PCI clock (see **Clocking**).
CLK_IN	Input	1	When the GENERATE_PCICLK parameter is set, this input is used to drive the PCI clock output, and also clocks the internal core logic. CLK_OUT should be used to clock additional logic inside the FPGA (see **Clocking**).
RST_OUTN	Output	1	Reset output. This is a buffered version of the PCI reset. If USE_GLOBAL_RESET = 1, a global buffer drives the internal reset network in the core as well as this reset output. If USE_GLOBAL_RESET = 0, a buffer tree is created and a reset output from the tree is fed to the reset output. RST_OUTN should be used for resetting any other logic in the FPGA. If the hot-swap function is enabled, this reset will also be asserted during a hot-swap insertion or extraction cycle per the Hot-Swap Specification.
FRAMEN_OUT	Output	1	Buffered version of the PCI FRAMEN signal intended for connection to a PCI arbiter. Care must be exercised when using this output to avoid causing PCI setup timing issues.
IRDYN_OUT	Output	1	Buffered version of the PCI IRDYN signal intended for connection to a PCI arbiter. Care must be exercised when using this output to avoid causing PCI setup timing issues.
SERRN_OUT	Output	1	Buffered version of the PCI SERRN signal. Allows the backend logic to know whether SERRN has been asserted. Care must be exercised when using this output to avoid causing PCI setup timing issues.
CFG_STATUS	Output	16	Provides the current value of the PCI configuration status register.

Downloaded from Arrow.com.

7.3 Backend Target and Master Dataflow Signals

The signals in Table 22 are used for Target and Master data transfers between the core and the user's backend logic. All these inputs and outputs are synchronous to the PCI clock. Users should ensure that setup times are met across this interface.

Table 22 • **Dataflow Interface Signals**

Name	Type	Width	Description
BAR_SELECT	Output	3	Active high bus indicating which BAR is being used for the current transaction. Values '000' to '101' indicate BARs 0 to 5, '110' indicates the Expansion ROM, and '111' indicates that no transaction is in progress. This output becomes valid on the same clock cycle where DP_START is asserted and returns to '111' on the same clock cycle where DP_DONE is asserted.
RD_CYC	Output	1	Active high signal indicating a read transaction from the backend. This output becomes valid on the same clock cycle where DP_START is asserted and returns to 0 on the same clock cycle where DP_DONE is asserted.
WR_CYC	Output	1	Active high signal indicating a write transaction from the backend. This output becomes valid on the same clock cycle where DP_START is asserted and returns to 0 on the same clock cycle where DP_DONE is asserted.
XFER_64BIT	Output	1	Active high signal indicating that the transfer is a 64-bit transfer. This output becomes valid on the same clock cycle where DP_START is asserted and returns to 0 on the same clock cycle where DP_DONE is asserted.
DP_START	Output	1	DP_START is an active high pulse indicating that a PCI transaction to the backend is beginning. A DP_START will always be followed by a DP_DONE when the cycle terminates.
DP_DONE	Output	1	Active high pulse indicating that a PCI transaction to the backend has finished. DP_DONE pulses will also occur when the core is inactive at a time when other PCI devices complete their PCI access cycles.
RD_STB_IN	Input	1	Active high read strobe indicating that the backend is ready to provide data to the core. Data will only be transferred when both RD_STB_IN and RD_STB_OUT are active. If the signal does not become active within the limits defined by the PCI bus, the read cycle will be terminated and the PCI bus terminated with a disconnect without data.
RD_STB_OUT	Output	1	Active high read strobe indicating that the core is ready to fetch data from the backend. Data will only be read when both RD_STB_IN and RD_STB_OUT are active. The core will read data from the MEM_DATA_IN bus on the next rising clock edge, i.e., while the strobes are active if RD_SYNC is LOW, or on the following clock edge if RD_SYNC is HIGH.
WR_BE_RDY	Input	1	Active high input indicating that the backend is ready to receive data from the core. If the ready signal does not become active within the time limits defined by the PCI bus, a disconnect without data will be initiated.
WR_BE_NOW	Output	4/8	Active high output indicating that the data should be written to the backend device now. Four write strobes are provided for 32-bit cores, one per byte. For example, WR_BE_NOW[0] indicates that data bits 7:0 should be written. 64-bit cores provide eight write strobes, one per byte.

Downloaded from Arrow.com.

Table 22 • **Dataflow Interface Signals *(continued)***

Name	Type	Width	Description
MEM_ADD	Output	N	Memory address bus, where *N* is defined by the parameter MADDR_WIDTH. The lowest two bits of the address bus will contain the lowest two bits of the PCI address bus. These address lines can be ignored for memory transfers but may be used to verify the legality of I/O byte accesses.
MEM_DATA_IN	Input	32/64	Data input, used for normal Target and Master data transfers as well as backend access to the DMA control registers
MEM_DATA_OUT	Output	32/64	Data output, used for normal Target and Master data transfers as well as backend access to the DMA control registers
MEM_DATA_OE	Output	1	Active high data enable for the lower 32 bits of MEM_DATA_OUT. This is intended as an output enable if MEM_DATA_IN and MEM_DATA_OUT are connected to bidirectional I/O pads to create a bidirectional MEM_DATA bus.
MEM_DATA_OE64	Output	1	Active high data enable for the upper 32 bits of MEM_DATA_OUT. This is intended as an output enable if MEM_DATA_IN and MEM_DATA_OUT are connected to bidirectional pads to create a bidirectional MEM_DATA bus.
BYTE_VALN	Output	1	Active low strobe indicating that the BYTE_ENN outputs are valid.
BYTE_ENN	Output	4/8	Active low byte enables. To achieve high throughput, CorePCIF normally reads all four bytes of data independently of the byte enable requests from the PCI bus. If the backend logic is required to support byte read operations, the backend should wait until BYTE_VALN is active (LOW) and then use this bus as read byte enables. Using this transfer mode will significantly slow throughput. These outputs can also be used to verify the legality of an I/O byte access before asserting WR_BE_RDY by comparing with the lowest two bits of MEM_ADDR. If an illegal I/O access is detected, the ERROR input can be asserted to cause a Target abort.
RD_SYNC	Input	1	When LOW, this signal indicates that the core will sample data on the rising clock edge while RD_STB_OUT and RD_STB_IN are active. When HIGH, this signal indicates that the core will sample data on the clock cycle after RD_STB_OUT and RD_STB_IN are active. This should be set HIGH when synchronous memories are connected to the backend interface.
RD_FLUSH	Input	6	Only has an effect when the FIFO recovery logic is enabled. If active (HIGH) when DP_START occurs, the internal FIFO will be flushed. RD_FLUSH[0] is used to flush the internal FIFO on BAR 0, RD_FLUSH[1] flushes the BAR 1 FIFO, etc. When the FIFOs are flushed, any data that was stored in the internal FIFO will be lost.
FIFO_EMPTYN	Input	6	Only used when the FIFO recovery logic is enabled and ENABLE_FIFOSTAT = 1. Active low input indicating that the external FIFO connected to the core is empty. The core uses this to set the external FIFO empty bits in the FIFO status register. FIFO_EMPTYN[0] sets the bit for BAR 0, etc. (This input is not used by any of the control logic in the core. It is only connected to the FIFO status register.)

Downloaded from Arrow.com.

7.4 Backend Target Dataflow Signals

The additional signals in Table 23 are used only for Target data transfers between the core and the user's backend logic. These signals only function when the TARGET parameter is set. All these inputs and outputs are synchronous to the PCI clock.

Table 23 • **Target Mode Control Signals**

Name	Type	Width	Description
BUSY	Input	1	Active high input indicating that the backend controller cannot complete the current transfer. When BUSY is asserted, the core may perform a PCI retry, a disconnect without data, or a disconnect with data cycle, depending on the state of the internal pipeline and the backend WR_BE_RDY and RD_STB_IN signals.
ERROR	Input	1	Active high signal that will force the PCI core to terminate the current transfer with a Target abort cycle. Once asserted, ERROR must be held active until DP_DONE occurs.
EXT_INTN	Input	1	Active low interrupt from the backend. When PCI interrupts are enabled, this will cause an INTAN signal to be asserted.
INTAN_OUT	Output	1	Active low output indicating the status of the PCI INTAN signal. This allows the backend logic to respond to an interrupt from another PCI device that has asserted INTAN.

7.5 Backend Master Dataflow Signals

The additional signals in Table 24 are used only for Master data transfers between the core and the user's backend logic. These signals only function when the MASTER parameter is set. All these inputs and outputs are synchronous to the PCI clock.

Table 24 • **Master Mode Signals**

Name	Type	Width	Description
MAST_ACTIVE	Output	1	Indicates (active high) that the current transaction is a Master transaction initiated by the DMA engine. MAST_ACTIVE becomes active one clock cycle before DP_START and goes inactive one clock cycle before DP_DONE. This output can be delayed externally by a clock cycle so it aligns with DP_START, DP_DONE, and the other backend control signals if required.
MAST_BUSY	Output	1	Indicates (active high) that the core is processing a DMA request. This signal becomes active when the DMA request is set in the DMA control register and stays active until the DMA completes.
DMA_BAR	Output	3	Indicates which BAR the DMA engine wishes to access. This output can be used with multiple FIFOs on the backend to multiplex their EMPTY/FULL signals to the RD_BUSY_MASTER and WR_BUSY_MASTER inputs.
WR_BUSY_MASTER	Input	1	When HIGH, a DMA write to the backend cycle will not be started.
RD_BUSY_MASTER	Input	1	When HIGH, a DMA read from the backend cycle will not be started.

Downloaded from Arrow.com.

Table 24 • **Master Mode Signals *(continued)***

Name	Type	Width	Description
STOP_MASTER	Input	1	When HIGH, an active DMA cycle will be stopped. Once asserted, this signal should be held asserted until DP_DONE is asserted. It may continue to be held active after DP_DONE has been asserted. If active when a DMA cycle starts, the core will transfer one word on the PCI bus before terminating the PCI transfer. After STOP_MASTER is asserted, it is possible that one or more data transfers to or from the backend may occur. For backend write cycles, one more data transfer will always occur. For backend read cycles, additional data transfers will happen if RD_STB_OUT was active and RD_STB_IN was inactive the clock cycle before STOP_MASTER was asserted. Typically, a FIFO empty output will be directly connected to both the RD_STB_IN and STOP_MASTER inputs.
STALL_MASTER	Input	1	If HIGH when CorePCIF starts a DMA cycle on the backend, the core will assert DP_START and delay asserting FRAME on the PCI bus until STALL_MASTER is deasserted (LOW), which signifies that the backend's data is now ready. This can be used to support backends that take many clock cycles to become ready. STALL_MASTER must be asserted on the clock cycle after MAST_ACTIVE becomes active. This is the same cycle in which DP_START occurs. The operation of the STALL_MASTER input is described in detail in **STALL_MASTER Operation**.

7.6 Backend Master DMA Register Access Signals

The signals listed in Table 25 are used to allow the backend to access the internal Target configuration space and DMA registers to initiate DMA Master transfers. These signals only function when the BACKEND parameter is set. The interface supports byte-wide operations if required. All these inputs and outputs are synchronous to the PCI clock.

Table 25 • **Backend DMA Register Access Signals**

Name	Type	Width	Description
BE_REQ	Input	1	A request from the backend to the core to take control of the backend interface. This signal is active high, and should be synchronous to the PCI clock.
BE_GNT	Output	1	A grant from the core giving control to the backend logic. When the BE_GNT signal is active and a transaction to the PCI Target controller occurs, the PCI controller will respond with a retry cycle. If a PCI cycle is in progress when BE_REQ is asserted, BE_GNT will not assert until completion of the current PCI cycle. If the backend must take control during an active PCI transfer cycle, it may assert the STOP or STOP_MASTER inputs, causing the current PCI cycle to terminate.
BE_READ	Input	1	Active high synchronous read enable for the DMA registers. It will be ignored if BE_GNT is inactive. During read cycles, there is a two-clock-cycle latency from BE_READ and BE_ADDRESS to valid data on MEM_DATA_OUT.
BE_WRITE	Input	4	Active high synchronous write enable for the DMA registers. One enable is provided for each of the four bytes; BE_WRITE[0] active will write bits [7:0] and will be ignored if BE_GNT is inactive.
BE_ADDRESS	Input	8	Address input that addresses the 256-byte configuration space. The lower two bits are ignored. The DMA registers are at addresses 50, 54, 58, and 5C hex.

Downloaded from Arrow.com.

Table 25 • **Backend DMA Register Access Signals *(continued)***

Name	Type	Width	Description
BE_CFGLOCK	Input	1	When 0, the complete internal configuration space can be read from and written to the backend interface. When 1, the main PCI configuration space (00–3F hex) can only be read; writes are prevented. This prevents the backend interface from modifying the PCI configuration space and potentially causing errors on the PCI bus. Writes to the DMA control registers are still allowed.

7.7 Hot-Swap Interface

The signals in Table 26 are used to implement the interface between the CorePCIF hardware connection process and the external physical connection process, as described in the PICMG 2.1 R2.0 Compact PCI Hot-Swap specification. These signals only function when the HOT_SWAP_ENABLE parameter is set.

Table 26 • **Hot-Swap Interface Signals**

Name	Type	Width	Description
HS_BDSELN	Input	1	Active low signal indicating that the board is selected and all other pins are fully connected. This input should be synchronous to the PCI clock.
HS_SWITCHN	Input	1	Active low signal from the board ejector switch. This input should be synchronous to the PCI clock and debounced outside the core.
HS_POWGOODN	Input	1	Active low signal indicating that the power supply is within specification. This input should be synchronous to the PCI clock.
HS_POWFAILN	Input	1	Active low signal indicating that the power supply is outside specification or a fault has occurred. This input should be synchronous to the PCI clock.
HS_ENUMN	Output (OC)	1	Active low signal notifying the system host that the board either has been inserted or is about to be extracted. This is an open-collector output and must be directly connected to an FPGA I/O pin.
HS_LEDN	Output	1	Active low signal to drive the external blue LED
HS_HEALTHYN	Output	1	Active low signal indicating that the board is healthy and may be released from reset and allowed onto the PCI bus

Downloaded from Arrow.com.

8 Timing Diagrams

Figure 11 through Figure 67 show the backend timing diagrams for different core operations. The timing diagrams are taken directly from core simulations. Table 27 summarizes the timing waveforms included in this handbook. Should additional waveforms be required, customers are encouraged to run simulations of the core. These can be done with the free, downloadable Evaluation version of the core.

Table 27 • **Example Waveforms**

Description	Figure(s)
Single-cycle read and write	Figure 11 to Figure 13
Burst transfer at maximum transfer rate	Figure 14 to Figure 16
Burst transfer with a slow PCI Master	Figure 17 to Figure 19
Burst transfer with a slow backend	Figure 20 to Figure 22
FIFO recovery operation	Figure 23 to Figure 25
Byte-controlled transfers	Figure 26 to Figure 28
64-bit burst transfer	Figure 29 to Figure 31
Slow read transfers	Figure 32 to Figure 33
Backend-terminated (BUSY) cycle at transfer start	Figure 34 to Figure 35
Backend-terminated (ERROR) cycle at transfer start	Figure 36
Backend-terminated (BUSY) cycle during data burst	Figure 37 to Figure 39
PCI configuration cycle	Figure 40 to Figure 41
PCI interrupt generation	Figure 42
Simple DMA transfer	Figure 43 to Figure 47
DMA cycle with a FIFO backend	Figure 48
STOP Master assertion during data burst	Figure 49 to Figure 52
RD_BUSY_MASTER and WR_BUSY_MASTER operation	Figure 53 to Figure 54
STALL_MASTER operation	Figure 55 to Figure 58
DMA register access from the backend	Figure 59 to Figure 63
DMA direct transfers	Figure 64 to Figure 65
Hot-swap insertion and extraction	Figure 66 to Figure 67

The figures typically show three sets of waveforms for each transfer type: one write and two read cycles. The read transfer is shown with a nonpipelined backend (RD_SYNC = 0) and a pipelined backend (RD_SYNC = 1). The waveforms show 32-bit operation; 64-bit operation is identical to 32-bit operation. A single set of waveforms is shown illustrating the 64-bit burst operation.

8.1 Single-Cycle Read and Write

Figure 11 to Figure 13 show basic single-cycle read and write accesses to the backend. The core will pulse DP_START active and indicate whether it is a read or write cycle and which BAR is being accessed. BAR_SELECT and RD_CYC/WR_CYC will become valid on the same clock cycle where DP_START is asserted and will remain active until DP_DONE is asserted. They are deasserted on the clock cycle after DP_DONE. Although the PCI interface requests only a single data word read, the core may read more than one word from the backend interface.

Downloaded from Arrow.com.

For read cycles, the core indicates that it is ready to read data by asserting RD_STB_OUT. The backend logic indicates that it has data available by asserting RD_STB_IN. When both of these signals are active, the core will read data from the backend. If RD_SYNC = 0, data is sampled on the clock edge during which the strobes are active (Figure 11). When RD_SYNC = 1, data is sampled on the following clock edge (Figure 12).

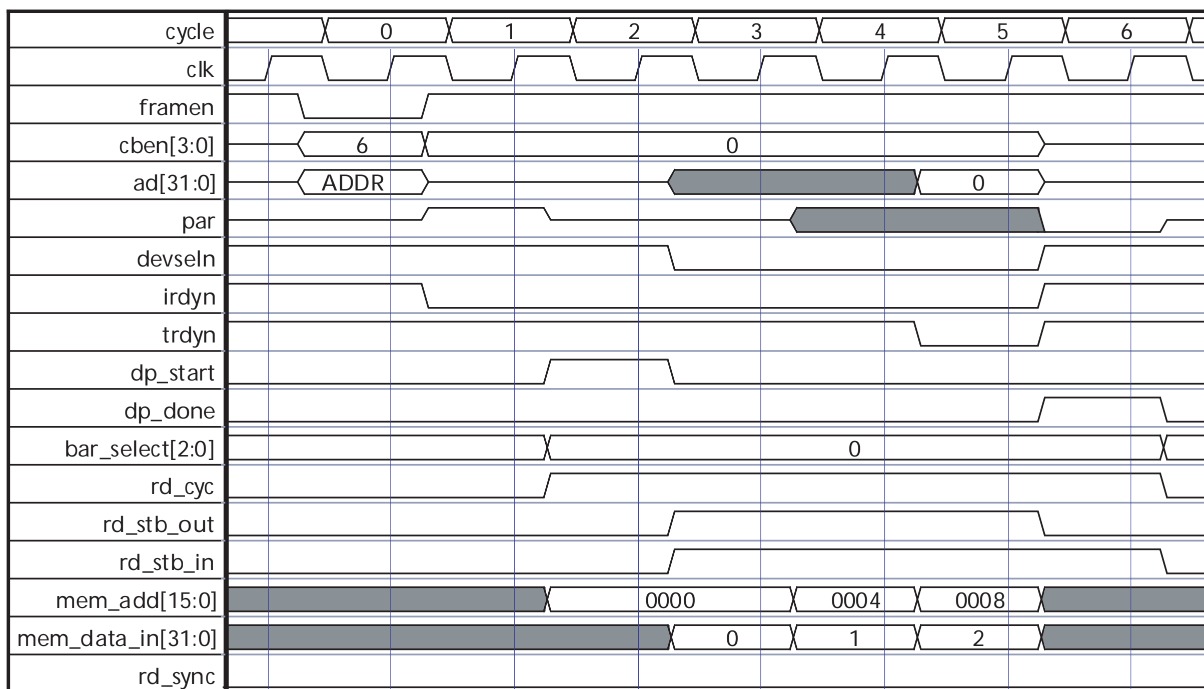
The PCI specification states that the maximum delay from FRAMEN assertion to the first data word transferred on the PCI bus is 16 clock cycles. The core takes two clock cycles from the time FRAMEN is asserted to assert DP_START. RD_STB_OUT is asserted one clock cycle later, and then an additional two clock cycles are required for the data to pass back through the core.

The backend must assert RD_STB_IN within 11 clock cycles. If the backend does not assert RD_STB_IN within this time, the core will automatically terminate the PCI transfer with a Target retry. When RD_SYNC = 1, an additional cycle of delay is added at the backend interface, reducing the initial backend latency to 10 clock cycles.

Once a data burst has started, data must be transferred every eight clock cycles. If the backend fails to provide data at this rate, the core will terminate the PCI cycle with a disconnect without data to maintain PCI compliance.

When the SX-A or RTSX-S families are used, the core inserts an additional clock cycle of latency internally. Thus, in this case, the backend maximum initial latency is reduced by one clock cycle. This is summarized in Figure 12.

Figure 11 • **Backend Read Cycle (RD_SYNC = 0)**



Downloaded from Arrow.com.

Figure 12 • **Backend Read Cycle (RD_SYNC = 1)**

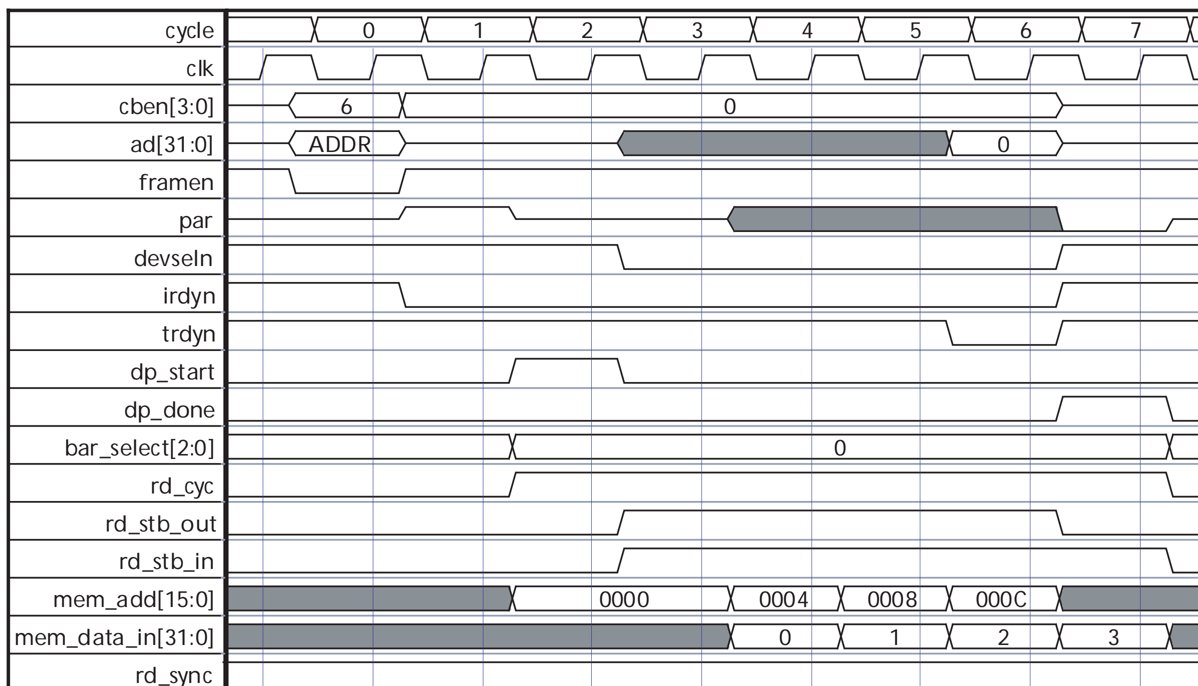


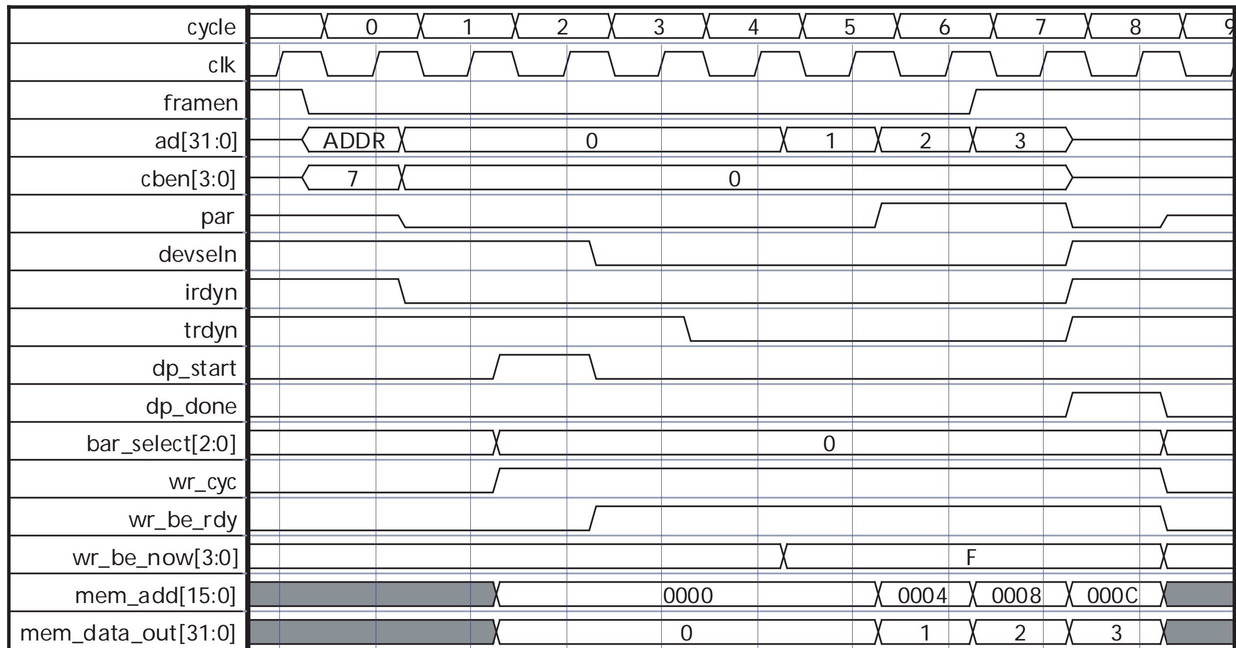
Table 28 • **Backend Initial Access Time Limits—Delay Allowed from DP_START to RD_STB_IN or WR_BE_RDY (clock cycles)**

	Read		
Family	**RDSYNC = 0**	**RDSYNC = 1**	**Write**
ProASIC3/E	11	10	13
ProASICPLUS	11	10	13
Axcelerator	11	10	13
RTAX-S	11	10	13
RTSX-S	10	9	13
SX-A	10	9	13

For write cycles, the backend indicates that it is ready to accept data by asserting WR_BE_RDY. The core then indicates that it is ready to accept data from the PCI bus by asserting TRDYN. When the core receives data from the PCI bus, it asserts the WR_BE_NOW strobes at the same time that the address and data are valid. For 32-bit PCI transfers, four WR_BE_NOW signals are provided and used to validate each byte. Thus, if the PCI Master performs a byte write, only one of the four write strobes will be active when the write occurs.

Downloaded from Arrow.com.

Figure 13 • **Backend Write Cycle**



8.2 Burst Transfer at Maximum Transfer Rate

Figure 14 to Figure 16 show basic backend burst read and write cycles. These transfers are similar to the single-cycle transfers except that multiple words are transferred.

Downloaded from Arrow.com.

Figure 14 • **Backend Burst Read Cycle (RD_SYNC = 0)**

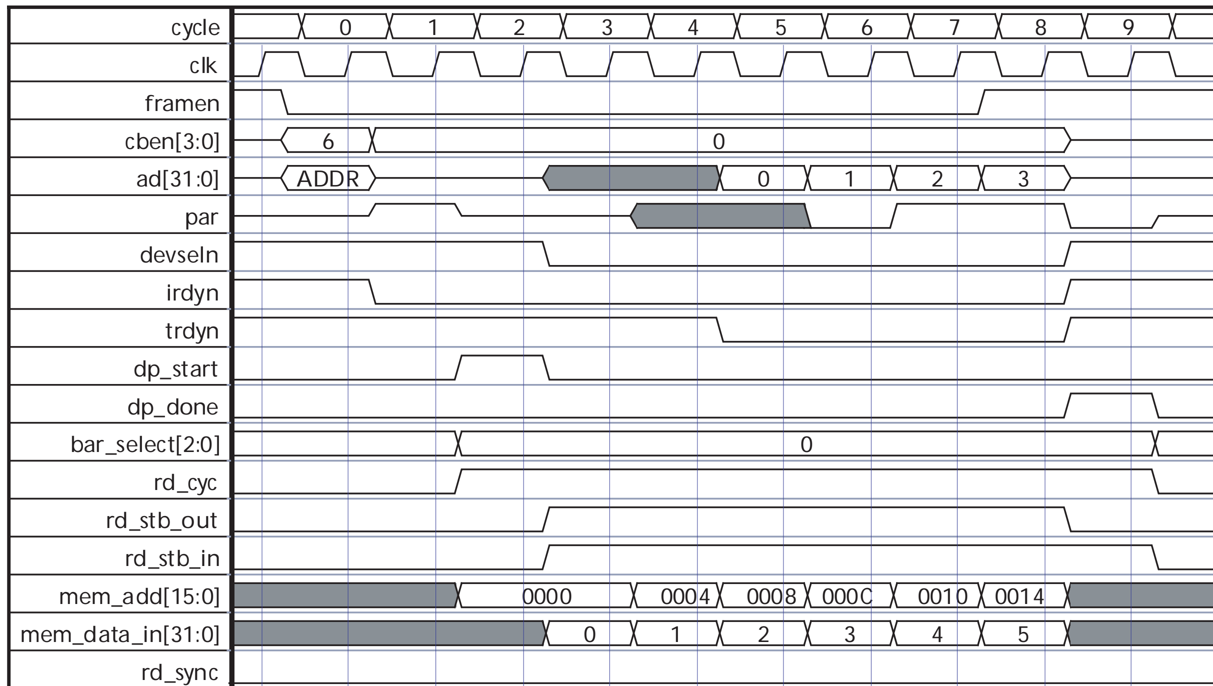
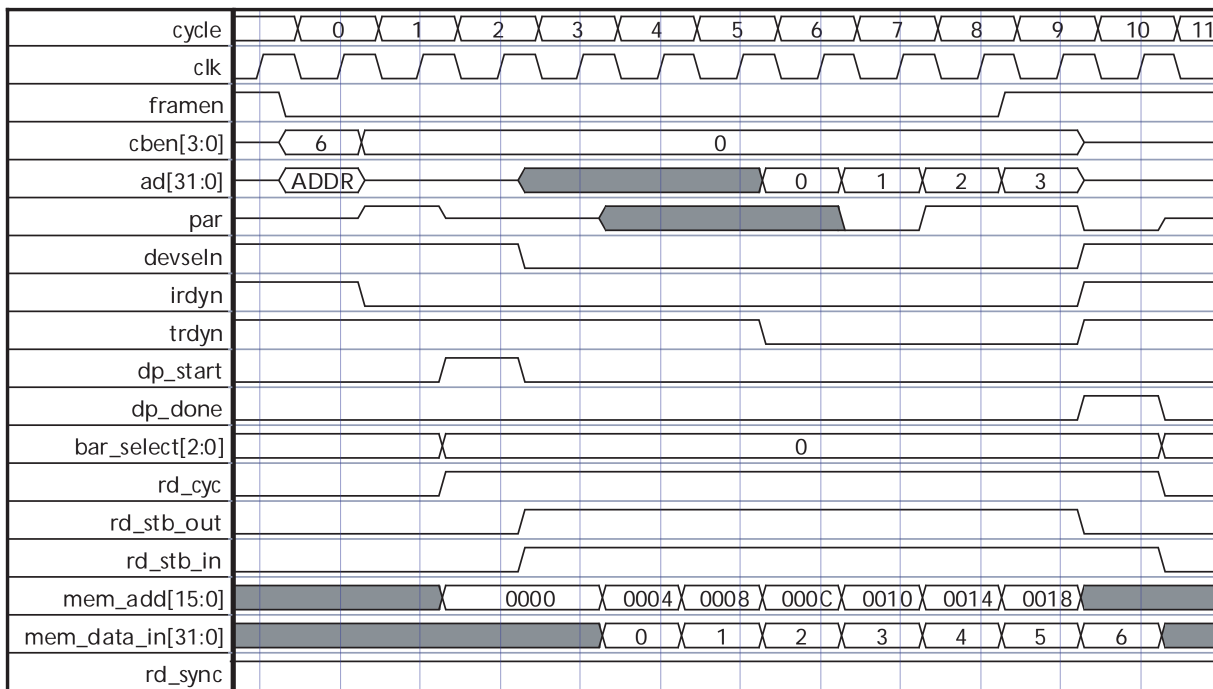
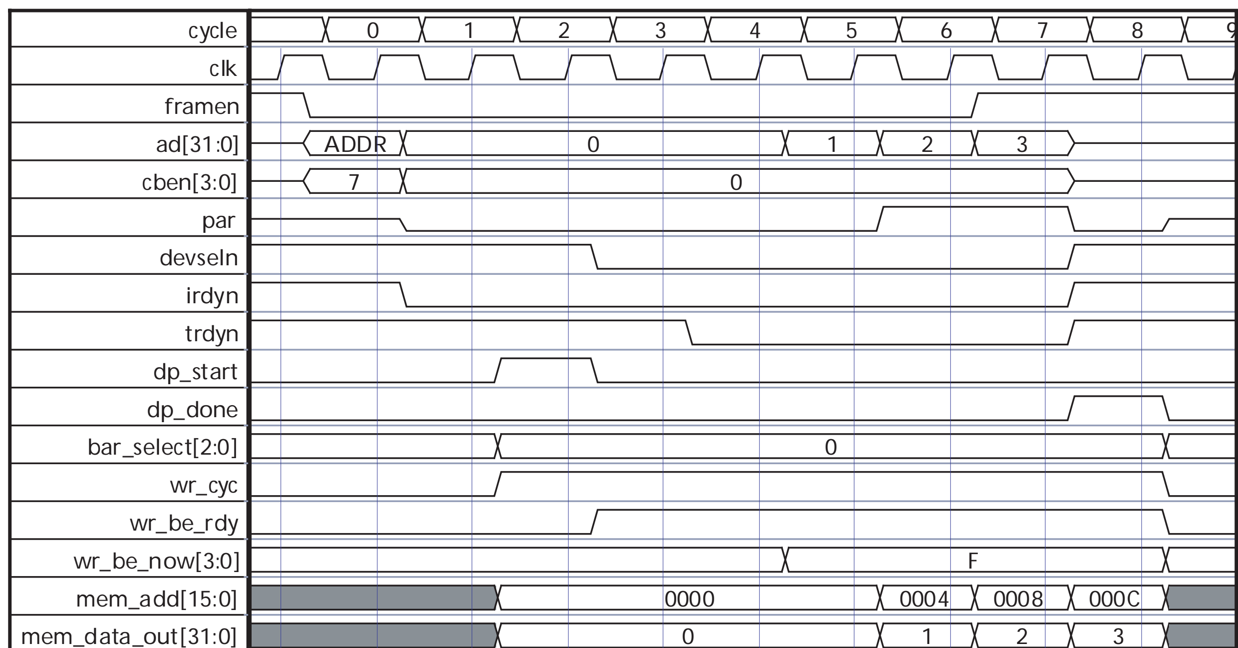


Figure 15 • **Backend Burst Read Cycle (RD_SYNC = 1)**



Downloaded from Arrow.com.

Figure 16 • **Backend Burst Write Cycle**



8.3 Burst Transfer with a Slow PCI Master

Figure 17 to Figure 19 show burst transfers with a PCI Master that transfers one word of data every three clock cycles. The backend in this case is capable of transferring data every clock cycle. The core reads data from the backend and stores it in the internal FIFO. Figure 17 shows seven data words being read from the backend (cycles three to nine). These words are then transferred on the PCI bus at the rate governed by the PCI Master. The core stores a maximum of five words internally. By cycle nine, the core has read seven words from the backend and transferred two words on the PCI bus. It then stops reading data from the backend. When the number of words stored in the core drops to four, the core starts reading data from the backend again.

Figure 18 shows the same slow PCI Master reading data from the backend when RD_SYNC = 1. The main difference here is that the core samples data on the clock edge following the strobe assertion. Since data is transferred a clock cycle later, the internal FIFO fill level is different than in Figure 17.

In the case of write transfers, the backend logic continuously asserts WR_BE_RDY, indicating it is ready to accept data. Data is then written to the backend every three clock cycles. The backend address increments after each write completes (Figure 19).

Downloaded from Arrow.com.

Figure 17 • **Backend Read Cycle with Slow PCI Master (RD_SYNC = 0)**

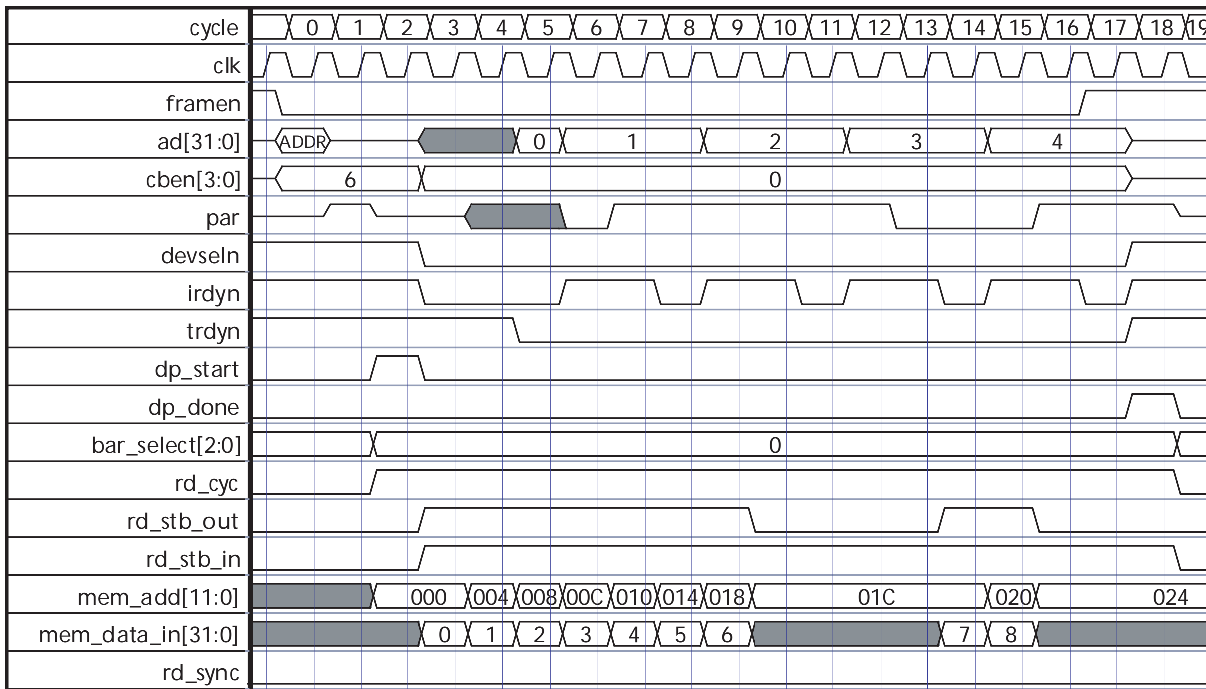
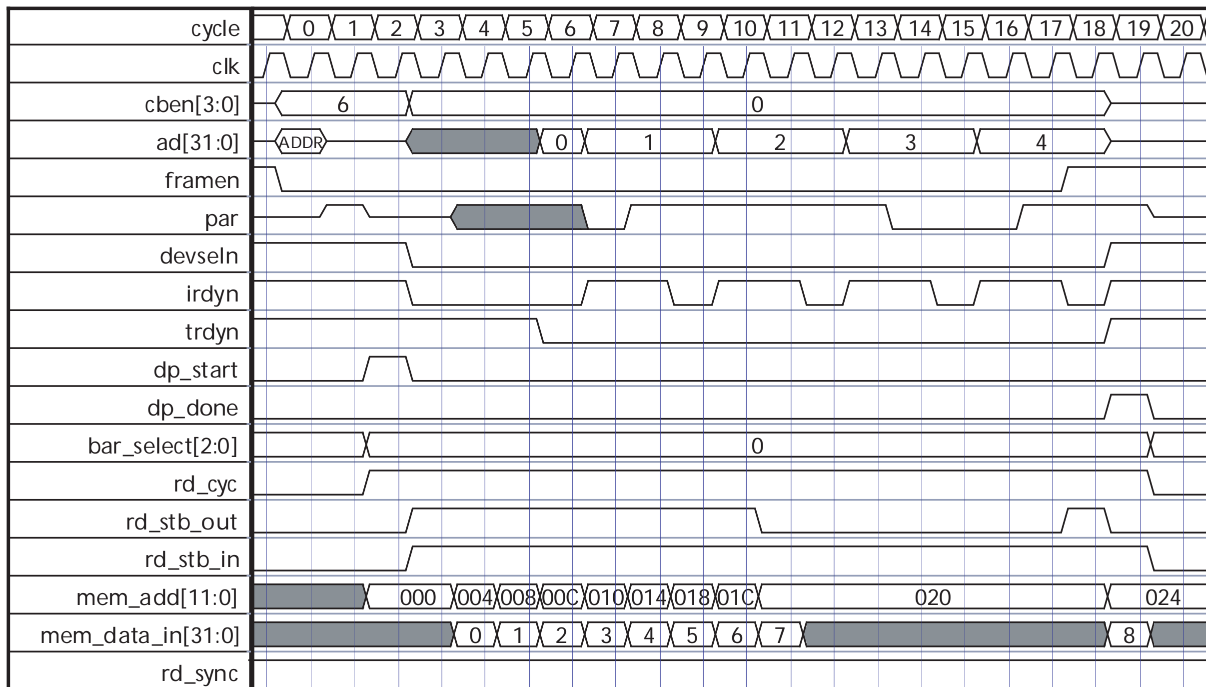


Figure 18 • **Backend Burst Read Cycle with Slow PCI Master (RD_SYNC = 1)**



Downloaded from Arrow.com.

Figure 19 • **Backend Burst Write Cycle with Slow PCI Master**

8.4 Burst Transfer with a Slow Backend

Figure 20 to Figure 22 show burst transfers with a backend that transfers one word of data every three clock cycles. The PCI Master in this case is capable of transferring data every clock cycle. During these transfers, the internal FIFO does not fill up, so the core keeps the RD_STB_OUT signal active, as it is ready to accept data. When the backend asserts the RD_STB_IN signal, data is transferred to the core and then transferred on the PCI bus two clock cycles later (three clock cycles for the SX-A and RTSX-S families).

Downloaded from Arrow.com.

Figure 20 • **Backend Burst Read Cycle with Slow Backend (RD_SYNC = 0)**

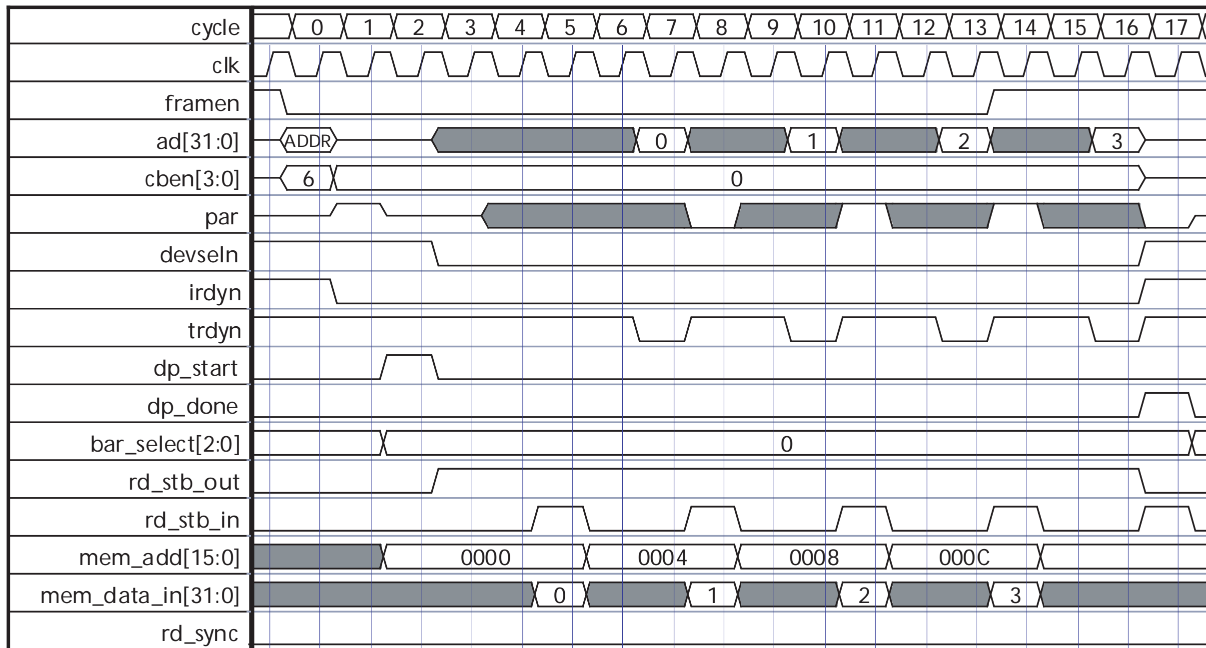
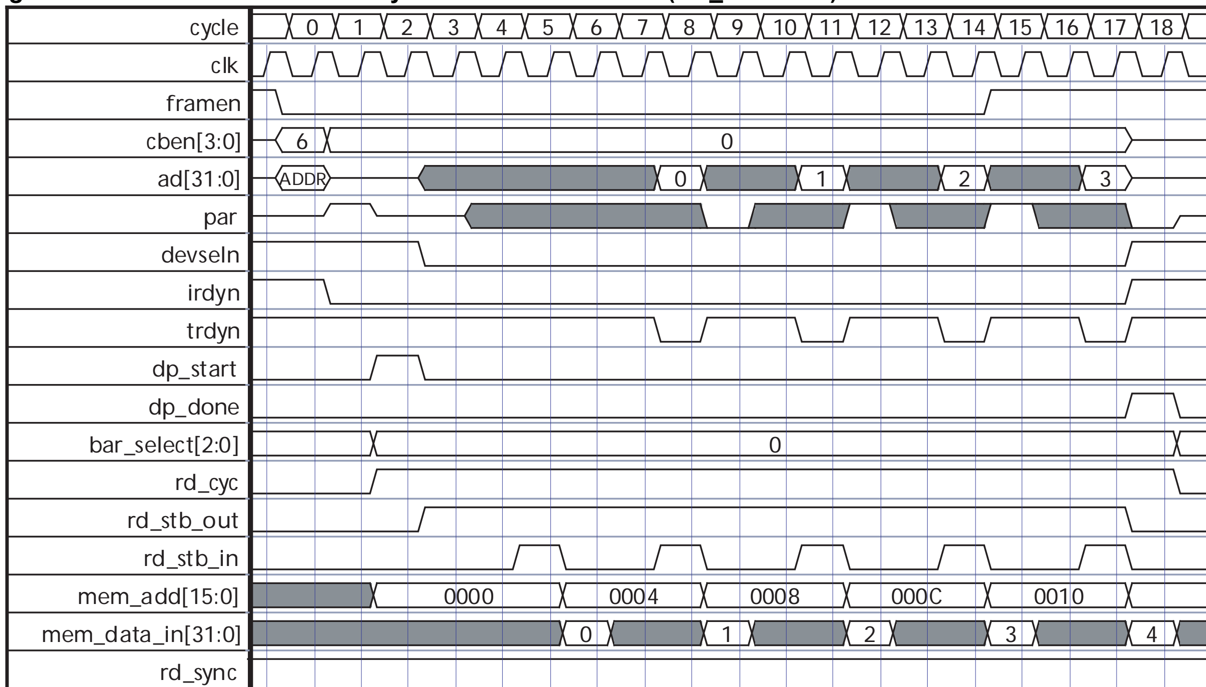


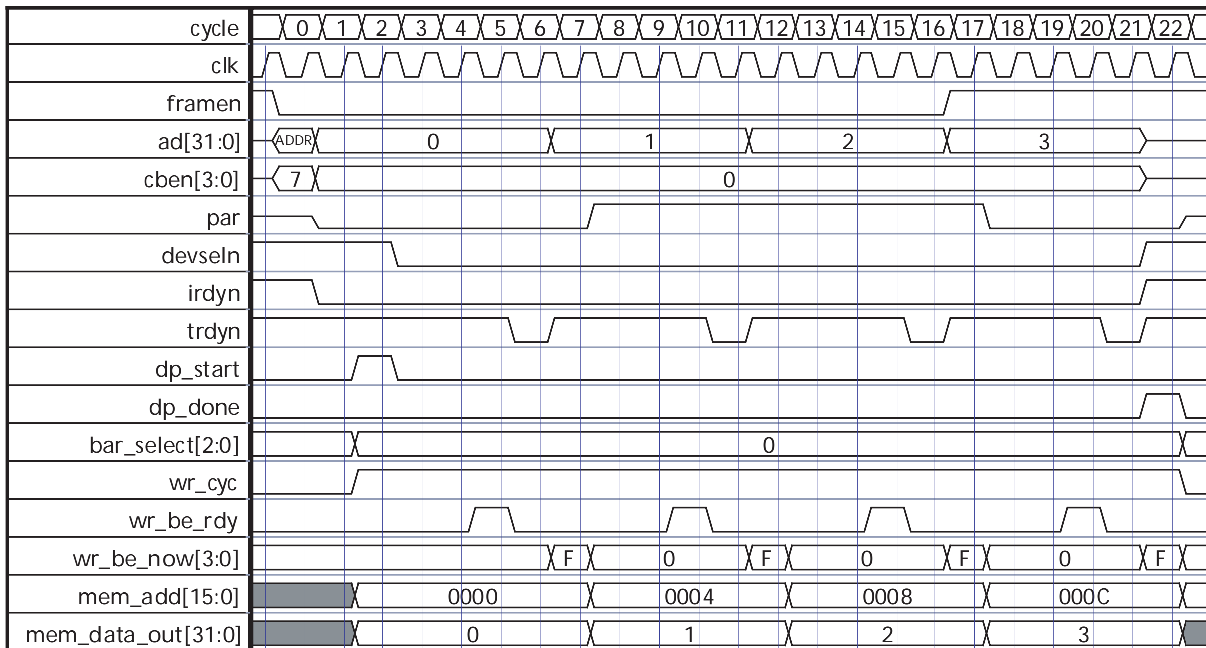
Figure 21 • **Backend Burst Read Cycle with Slow Backend (RD_SYNC = 1)**



During write transfers, the backend asserts its WR_BE_RDY signal when it is ready. This causes the core to assert TRDYN and then, assuming that IRDYN was active on the PCI bus, causes WR_BE_NOW to be asserted on the following clock edge. This allows the backend to control the transfer rate.

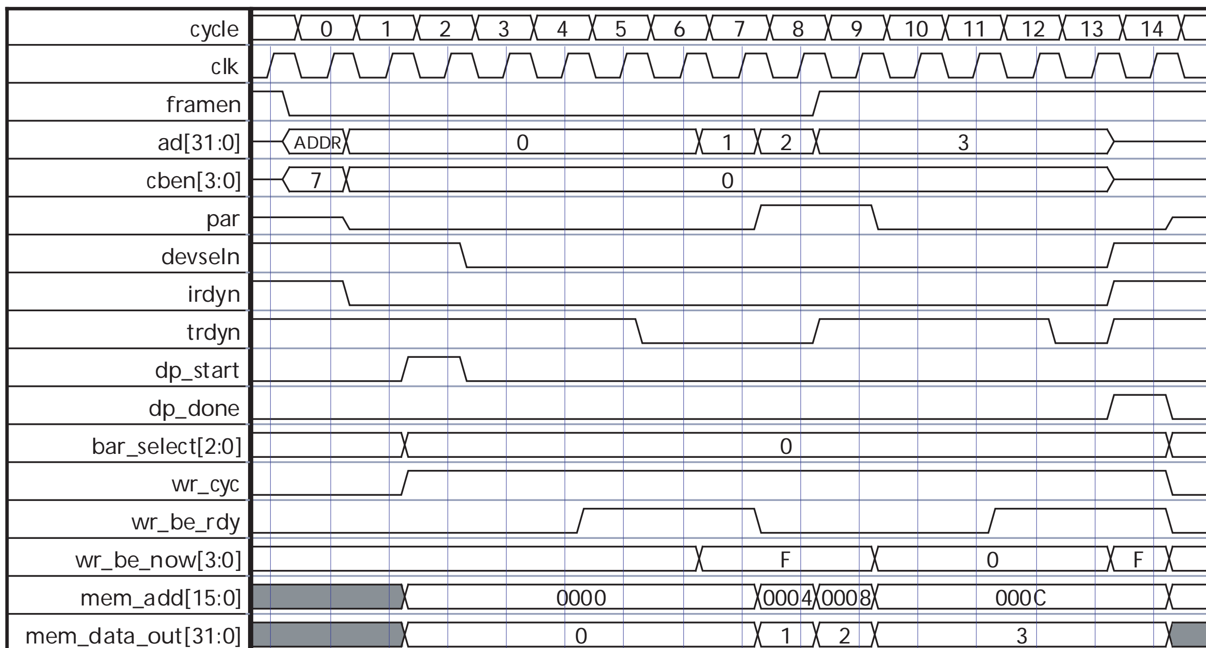
Downloaded from Arrow.com.

Figure 22 • Backend Burst Write Cycle with Slow Backend



If WR_BE_RDY has been continuously asserted and is deasserted, two additional writes to the backend may occur. To avoid these extra writes, the backend should only assert WR_BE_RDY for a single cycle and should not reassert it until the write takes place (WR_BE_NOW is asserted). These additional writes are shown in Figure 23 in cycles 8 and 9.

Figure 23 • Backend Burst Write Cycle with Additional Writes after Ready Removed



8.5 Burst Transfer with FIFO Recovery Enabled

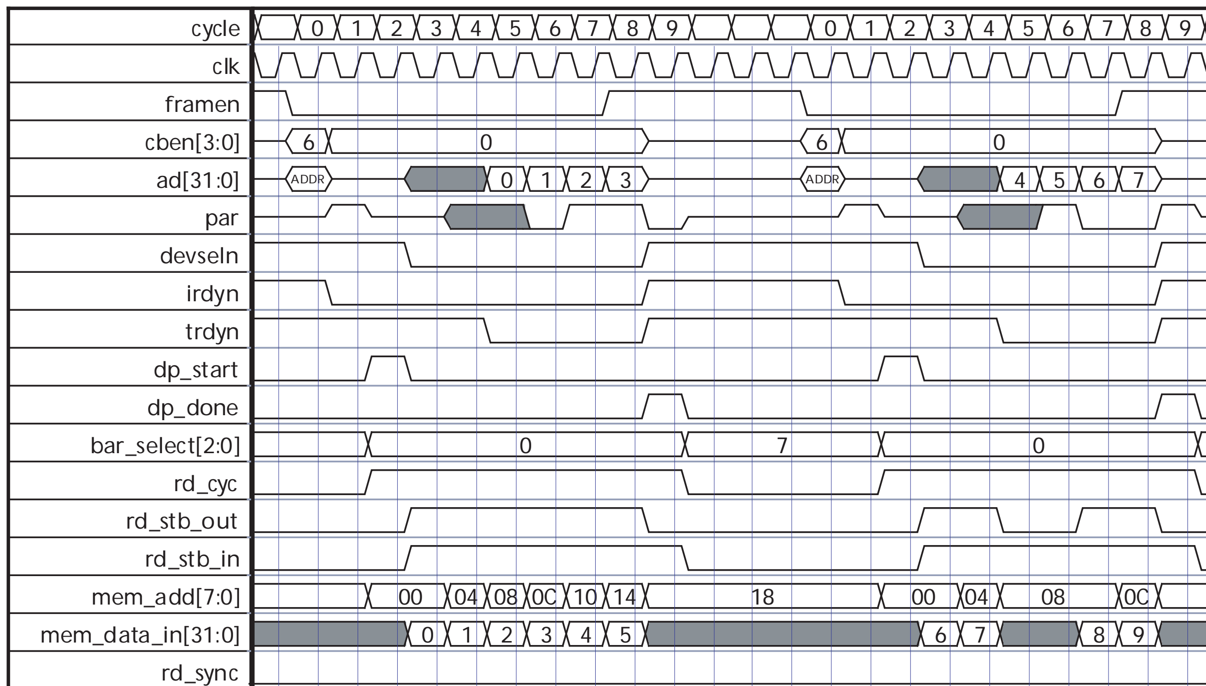
CorePCIF directly supports connection of external FIFOs. When FIFOs are connected to the core, special logic is implemented inside the core to prevent data loss. As seen in Figure 16 to Figure 18, the core reads ahead of the transfer on the PCI bus during a burst transfer so it can maintain high

Downloaded from Arrow.com.

throughput. This is illustrated in Figure 11. The core actually transfers four words on the PCI bus but reads seven from the backend interface. Without the optional FIFO recovery logic, these additional three words would be lost.

When the FIFO recovery mode is enabled, BAR*i*_ENABLE = 2. The core will store these three words internally and transfer them at the start of the next read cycle from the same BAR. Figure 24 shows an initial burst read cycle, followed by a second read cycle that initially transfers data stored from the first transfer.

Figure 24 • **FIFO Recovery Operation (RD_SYNC = 0)**

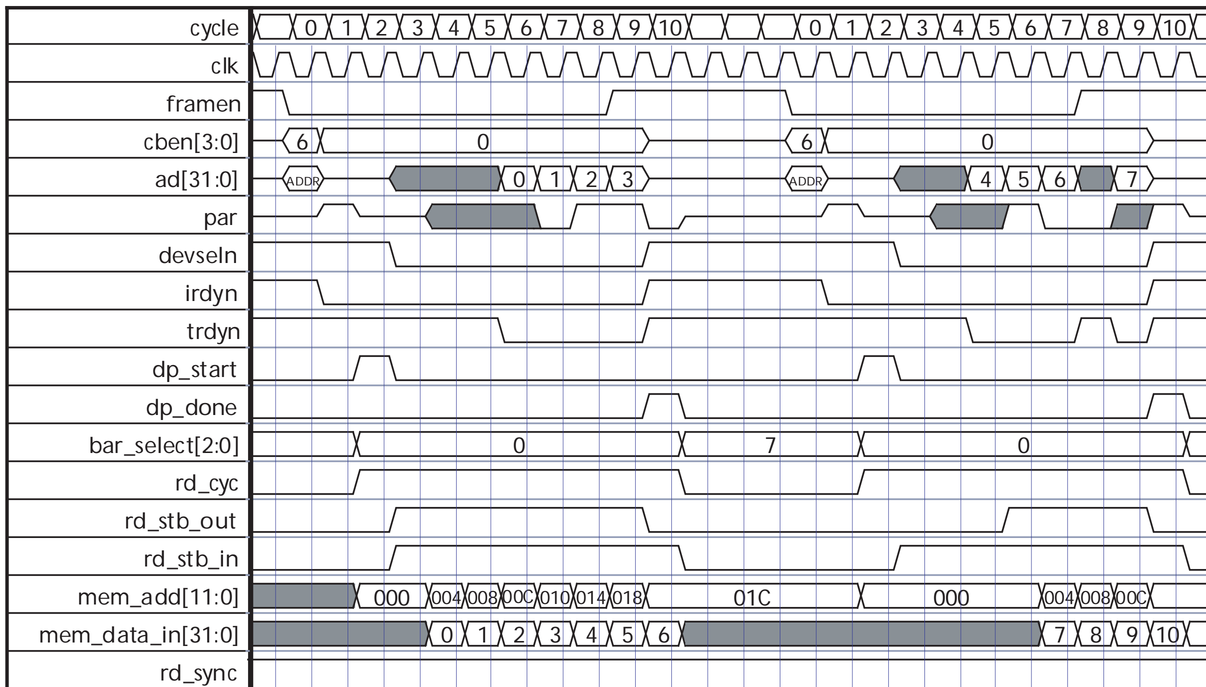


During the first read cycle, as shown in Figure 24, the core reads six words from the backend but only transfers four words on the PCI bus. The remaining two words, four and five, are stored in the core. On the second PCI burst read cycle, the core reads the next data words, six and seven, from the backend before it stops to prevent its internal storage from overflowing. At cycle five in the second PCI cycle, word four is transferred on the PCI bus. When the second PCI cycle terminates, words eight and nine are left stored in the core.

When RD_SYNC = 1, a very similar pair of transfers occurs, but in this case, the first transfer actually reads seven words and transfers four words on the bus, leaving three words stored in the core between the transfers.

Downloaded from Arrow.com.

Figure 25 • FIFO Recovery Operation (RD_SYNC = 1)



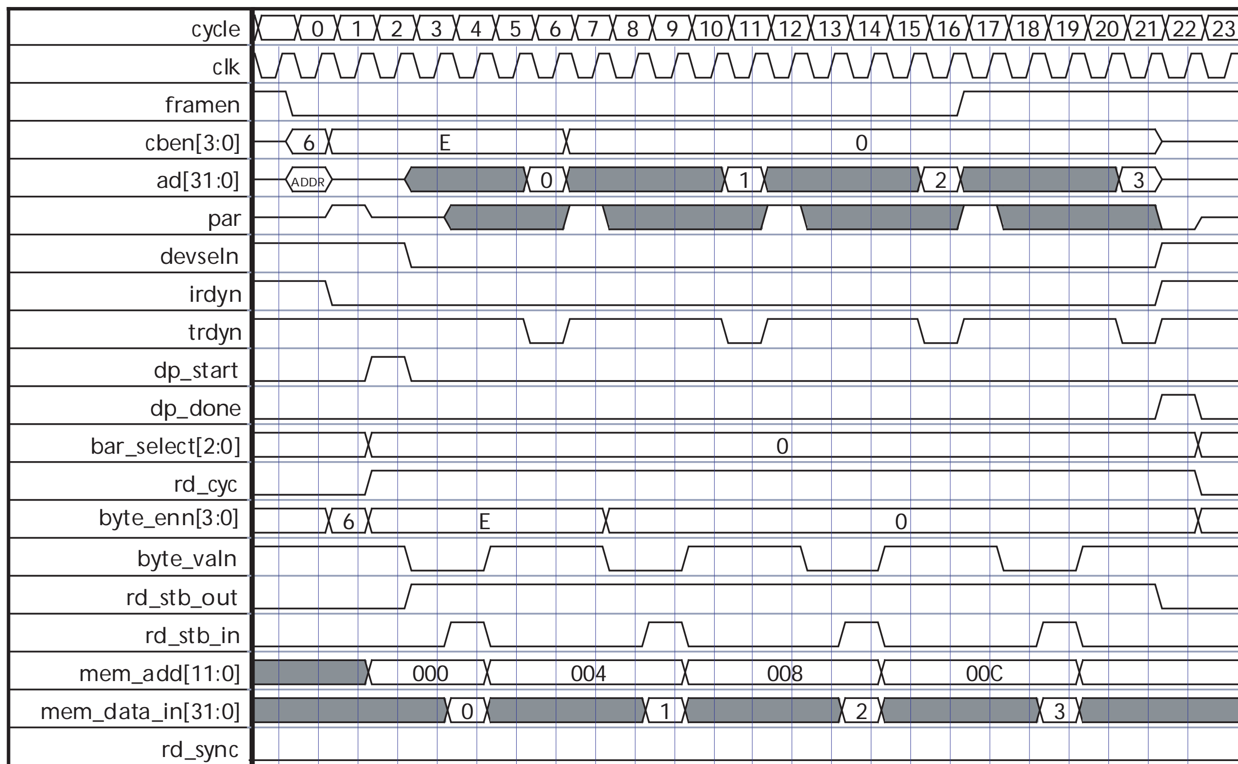
8.6 Byte-Controlled Transfers

In most systems, the backend ignores the byte requests from the PCI Master and simply reads all four or eight bytes from the backend. If required, the core can make the PCI byte enables available to the backend. This significantly reduces the read bandwidth, as shown in Figure 26, where a data transfer takes place every five clock cycles. If RD_STB_IN is asserted in the same clock cycle where BYTE_VALN is active, this can be reduced to four cycles.

The PCI Master sets the CBEN signals initially for each byte-controlled read transfer. These take one clock cycle to propagate to the core backend on the BYTE_ENN output (active low). The core indicates the validity of these signals by asserting the BYTE_VALN output (active low). At this point, the backend logic can perform its byte-controlled read operation and assert RD_STB_IN. Two cycles later, the data appears on the PCI bus and the core asserts TRDYN. Only now can the PCI Master start the next read cycle, updating the PCI CBEN lines for the next transfer, and the whole process repeats.

Downloaded from Arrow.com.

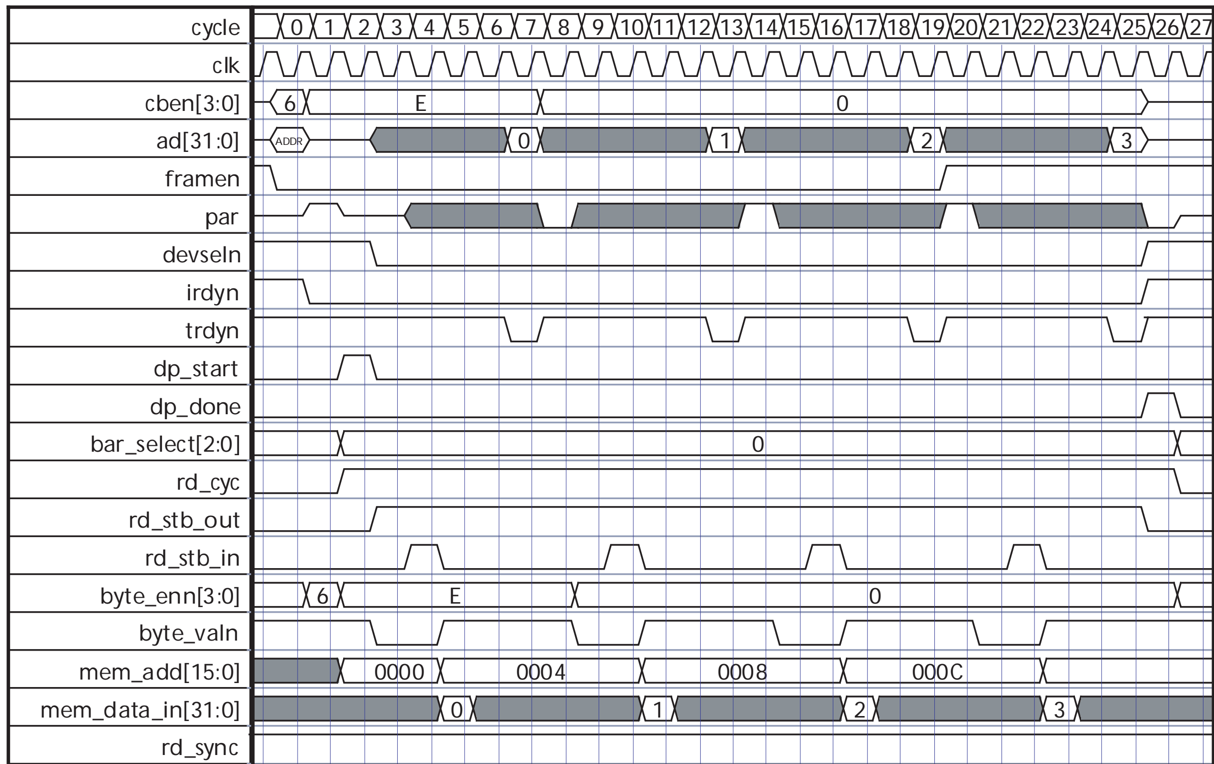
Figure 26 • **Backend Byte Read Cycle (RD_SYNC = 0)**



When RD_SYNC = 0, it will require a minimum of four clock cycles per transfer. When RD_SYNC = 1 (Figure 27), this increases to five clock cycles per transfer. If ProASIC3/E technology is being used, it will take up to six clock cycles per transfer.

Downloaded from Arrow.com.

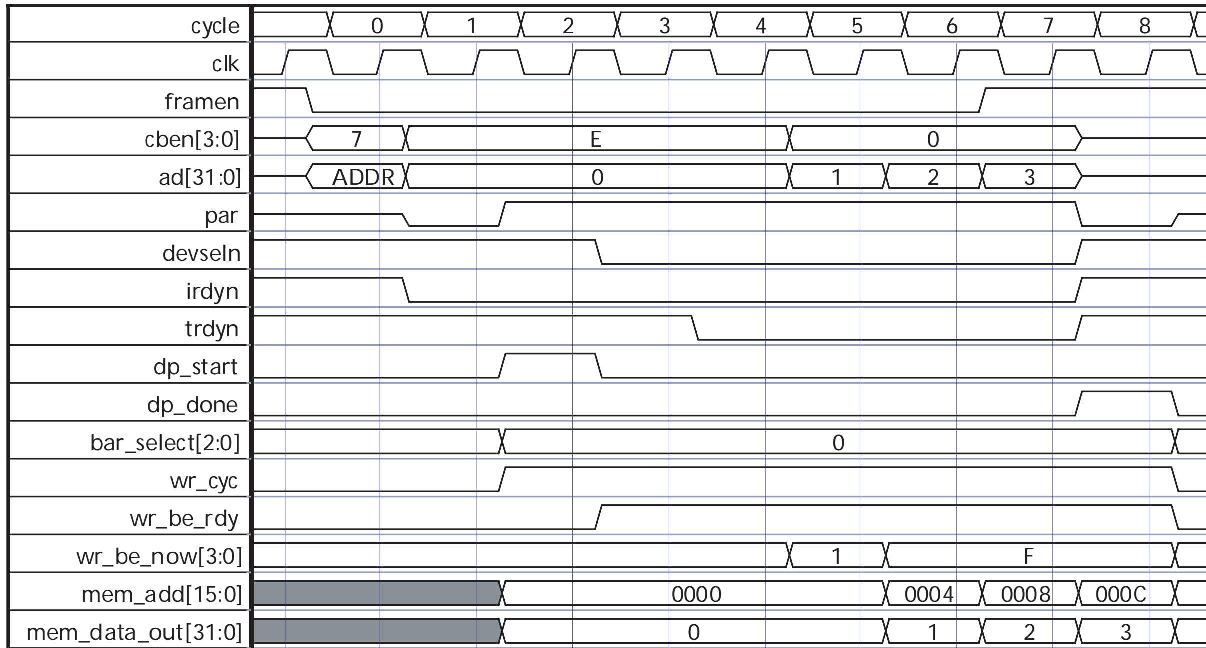
Figure 27 • **Byte Burst Read Cycle (RD_SYNC = 1)**



Byte-controlled write transfers do not suffer the same handicap as read transfers. The core always validates the write by setting the four or eight WR_BE_NOW signals with each data transfer. Figure 28 shows a write transfer that writes byte 0 on the first transfer and then all four bytes on the following transfers. Some systems may require that I/O accesses be verified for legality, i.e., AD[1:0] and CBEN[3:0] are consistent. In this case, the backend should wait until BYTE_VALN is active (LOW) and then verify MEM_ADD[1:0] and BYTE_VALN[3:0] for consistency (PCI Specification 3.2.2.1). If okay, the backend should assert WR_BE_RDY or RD_STB_IN; otherwise, it should assert the ERROR input to cause a Target abort.

Downloaded from Arrow.com.

Figure 28 • **Byte Burst Write Cycle**



8.7 64-Bit Burst Transfer

When operating in 64-bit mode (Figure 29 through Figure 31), the backend protocol is similar to that for 32-bit operation. The main differences are that the data bus width increases to 64 bits and four additional write strobes are provided along with four additional read byte strobes to support byte operations. Also, the address increments by eight rather than by four after each data transfer.

Downloaded from Arrow.com.

Figure 29 • 64-Bit Burst Read Cycle (RD_SYNC = 0)

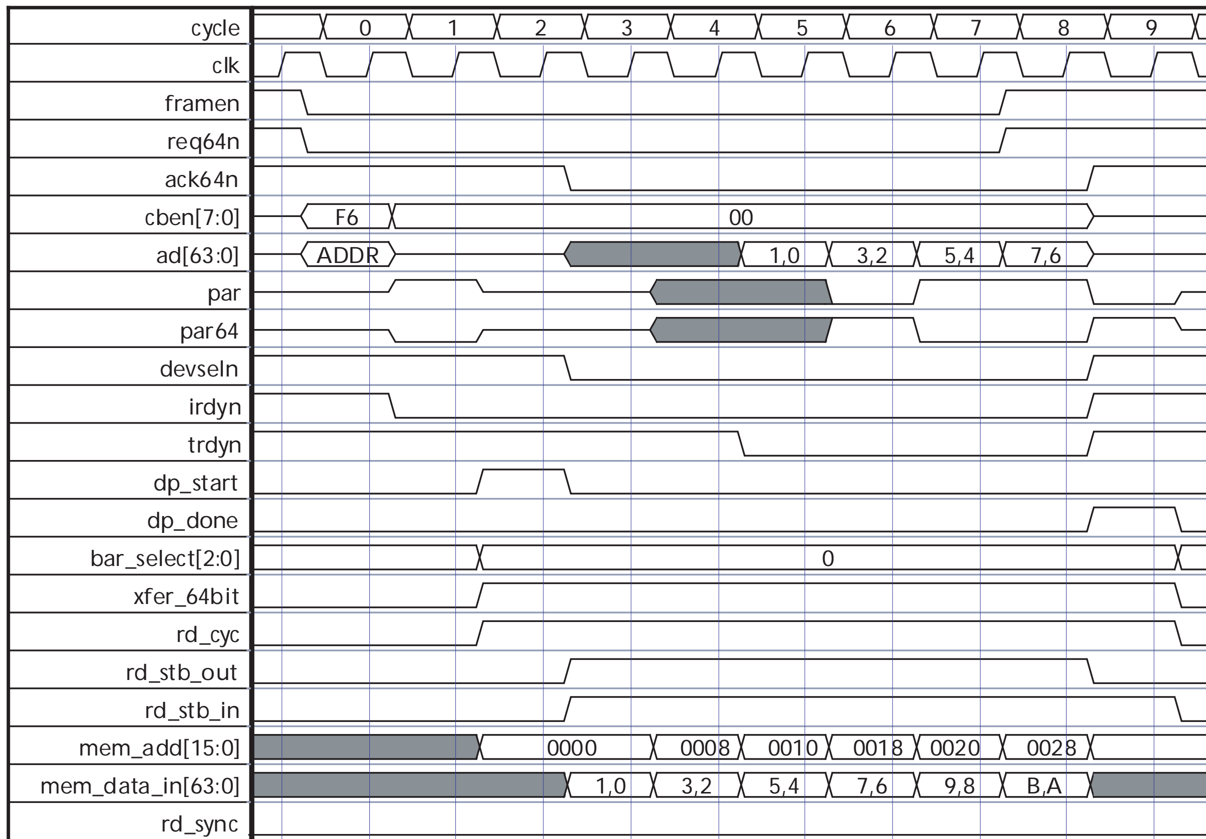
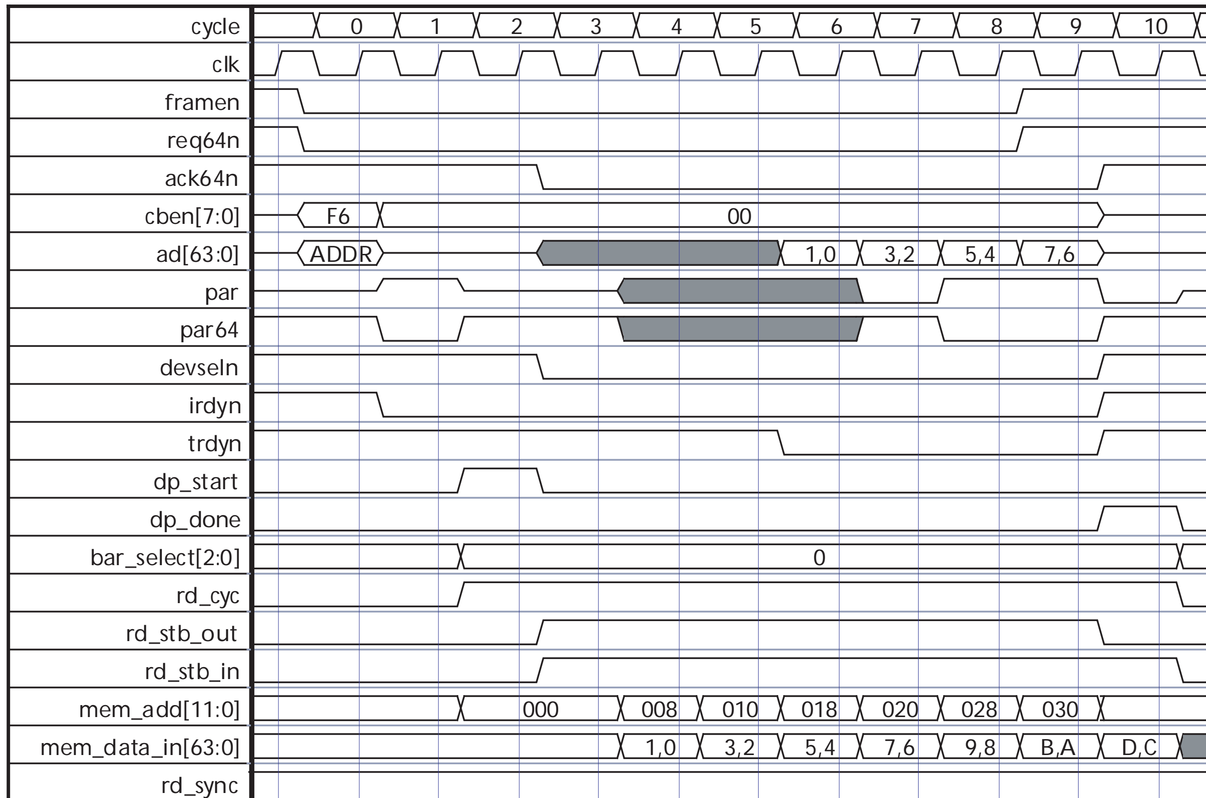
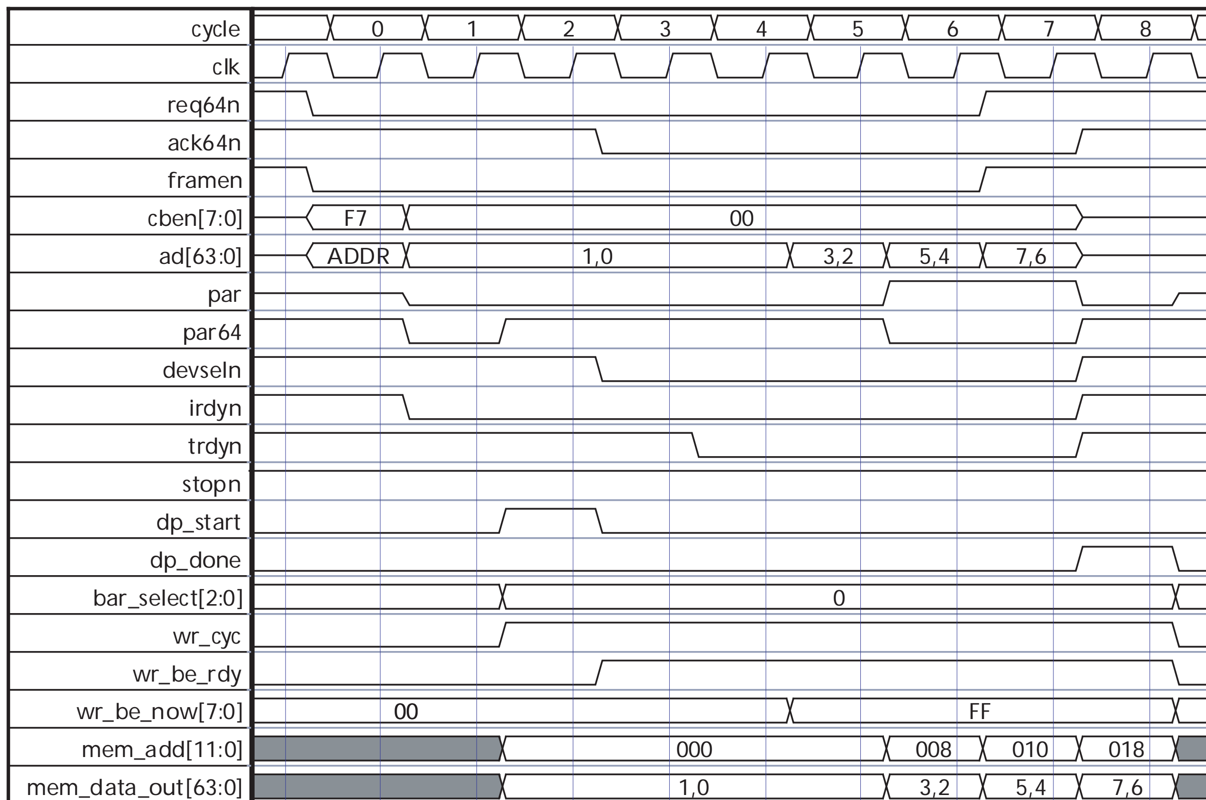


Figure 30 • 64-Bit Burst Read Cycle (RD_SYNC = 1)



Downloaded from Arrow.com.

Figure 31 • **64-Bit Burst Write Cycle**



8.7.1 Operating Note

When configured with 64-bit functionality, the core should only be used in 64-bit environments. The core does not implement the logic to allow a 64-bit core to function correctly when inserted into a 32-bit slot. The inclusion of this logic would significantly increase the amount of FPGA resources the core requires.

8.8 Slow Read Transfers

When the SLOW_READ parameter is set, CorePCIF transfers read data from the backend interface once every two clock cycles when RD_SYNC = 0 and once every three clock cycles when RD_SYNC = 1, assuming the Master holds IRDYN active, as shown in Figure 32 and Figure 33. If the Master deasserts IRDYN, additional clock cycles will be inserted between the data transfers. Write transfers operate as normal, and transfers every clock cycle are supported.

Enabling the SLOW_READ function removes the need for the internal data buffer, and hence reduces the gate count requirements of the core considerably, especially when the SX-A and RTSX-S families are used.

Downloaded from Arrow.com.

Figure 32 • **Slow Read Transfer (RD_SYNC = 0)**

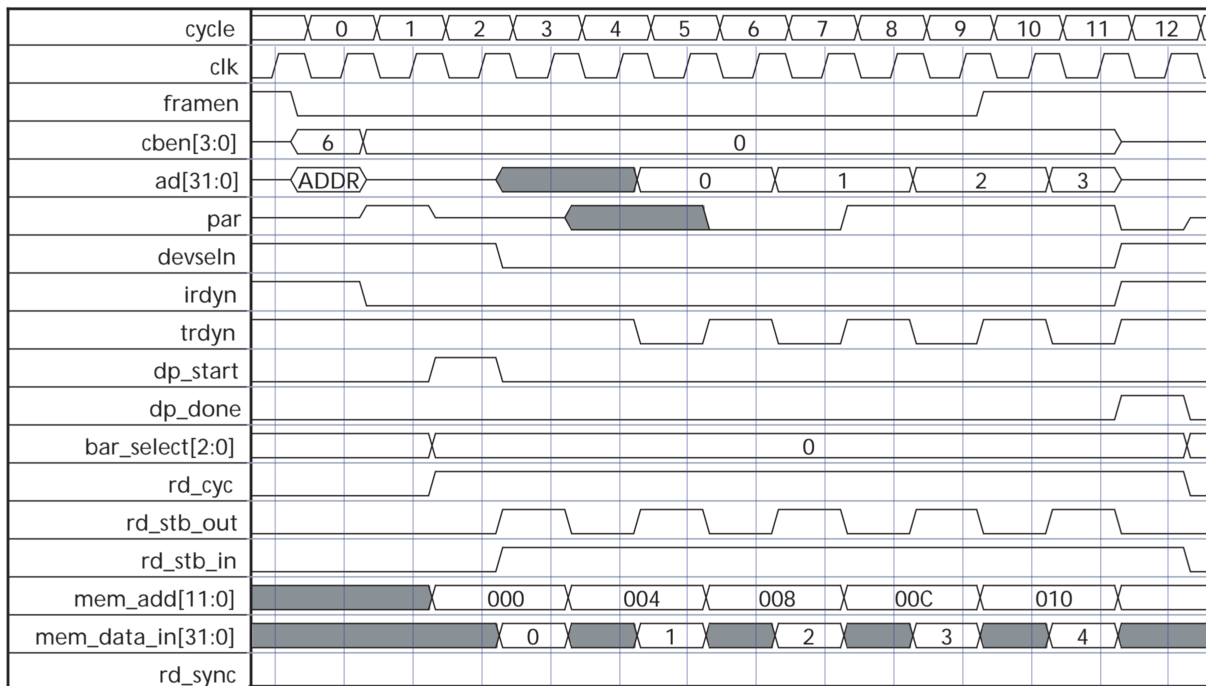
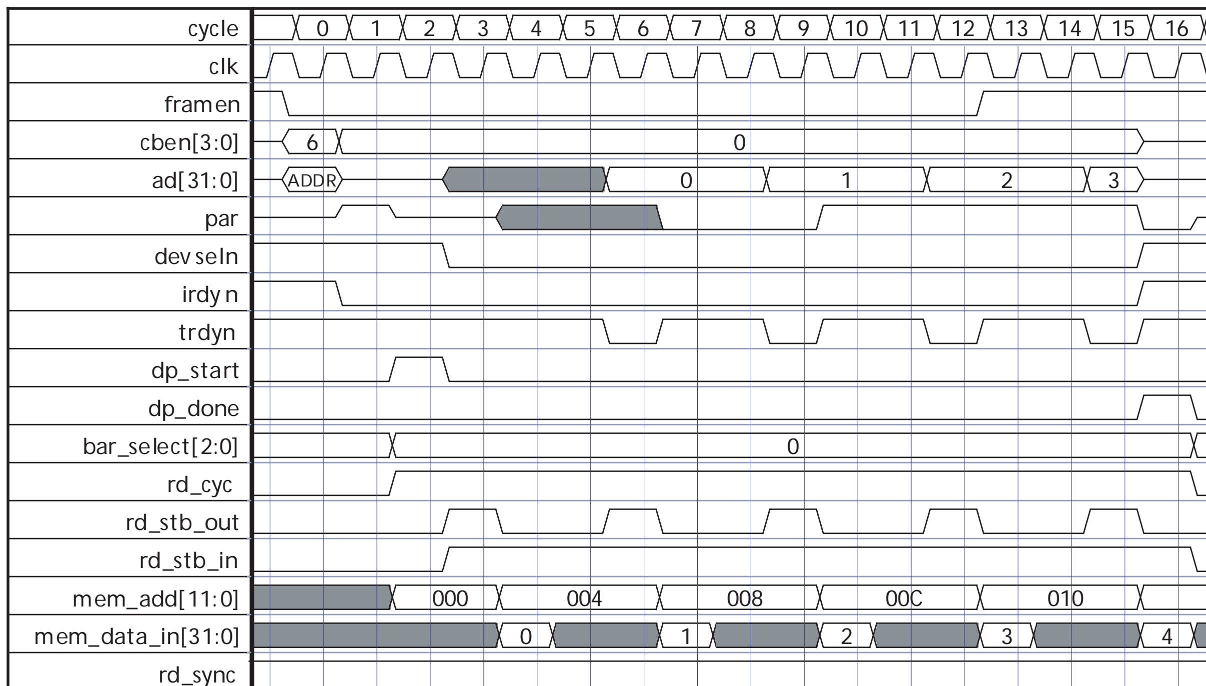


Figure 33 • **Slow Read Transfer (RD_SYNC = 1)**

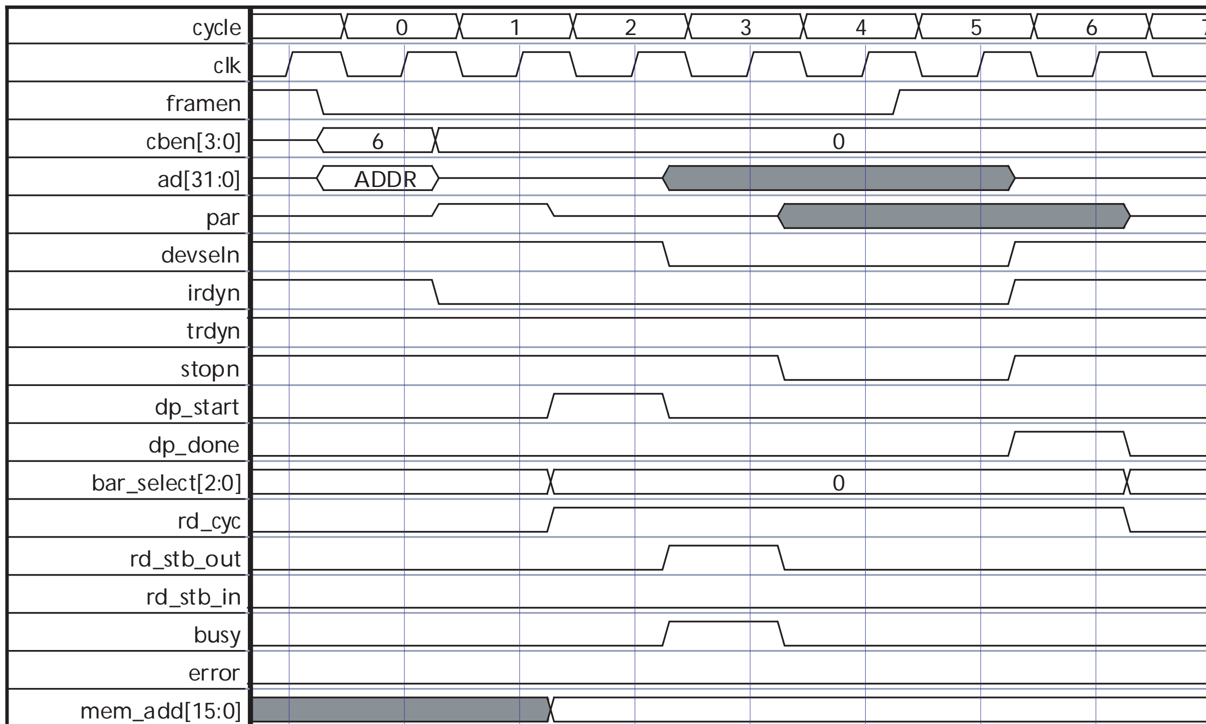


8.9 Backend-Terminated (BUSY) Cycle at Transfer Start (Target)

If the backend knows that it will not be able to fetch or accept a data transfer within the initial transfer period (Figure 12), it may immediately assert BUSY. This will cause the core to terminate the cycle with a retry (Figure 34).

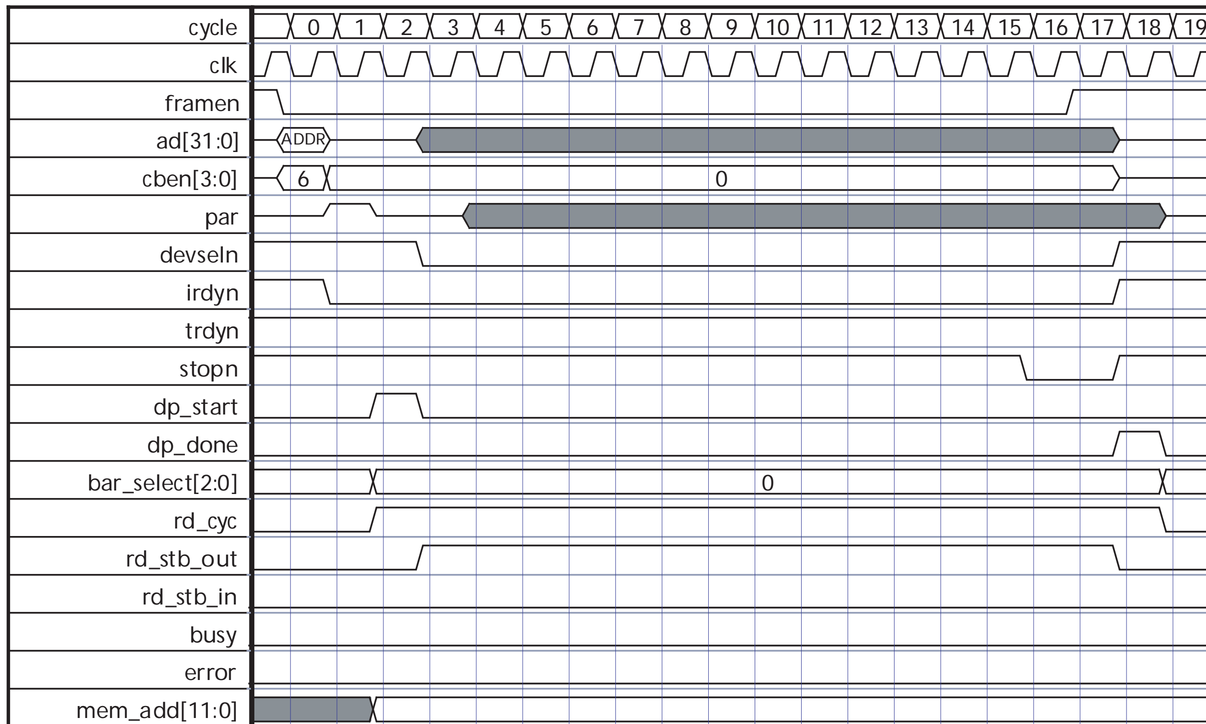
Downloaded from Arrow.com.

Figure 34 • Backend-Terminated (BUSY) Cycle at Transfer Start



If the backend does not assert the RD_STB_IN or WR_BE_RDY signal within the required time, the core will automatically terminate the PCI cycle by asserting the STOPN signal (Figure 35).

Figure 35 • Backend Fails to Assert RD_STB_IN

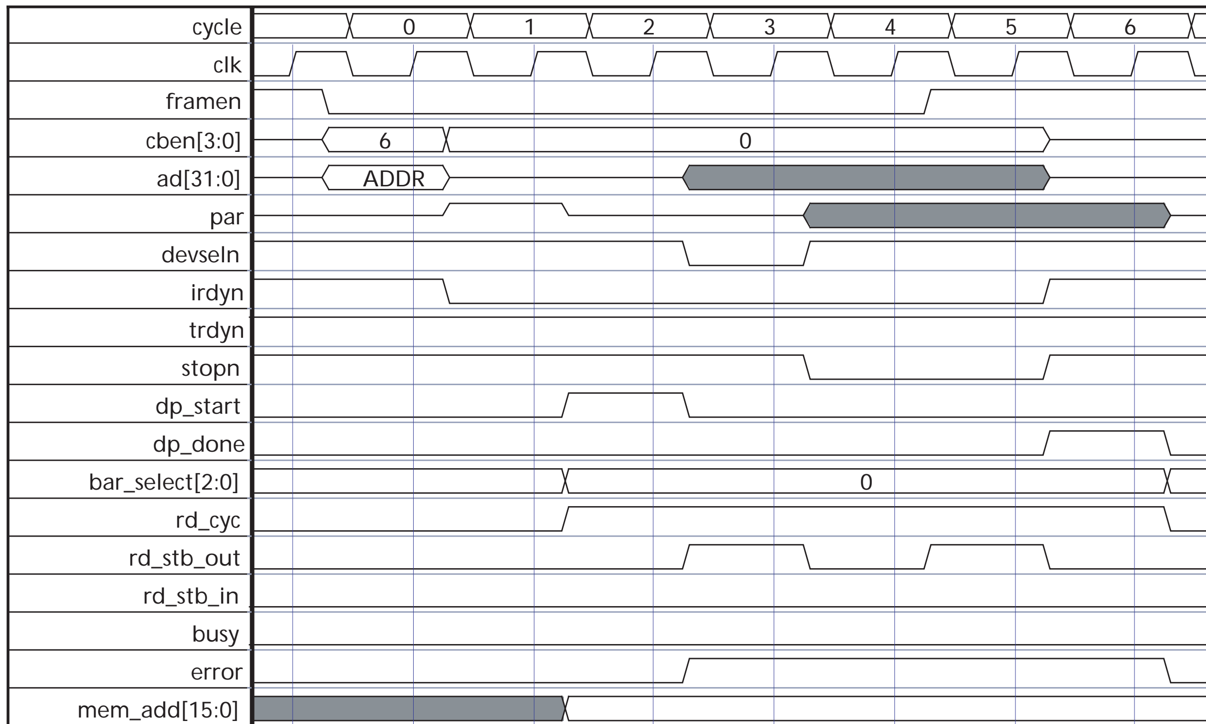


Downloaded from Arrow.com.

8.10 Backend-Terminated (ERROR) Cycle at Transfer Start (Target)

If the backend detects an illegal transfer, it may cause a Target abort by asserting the ERROR input (Figure 36). Once ERROR is asserted, the backend is required to hold the ERROR input active until DP_DONE occurs.

Figure 36 • **Backend-Terminated (ERROR) Cycle at Transfer Start**



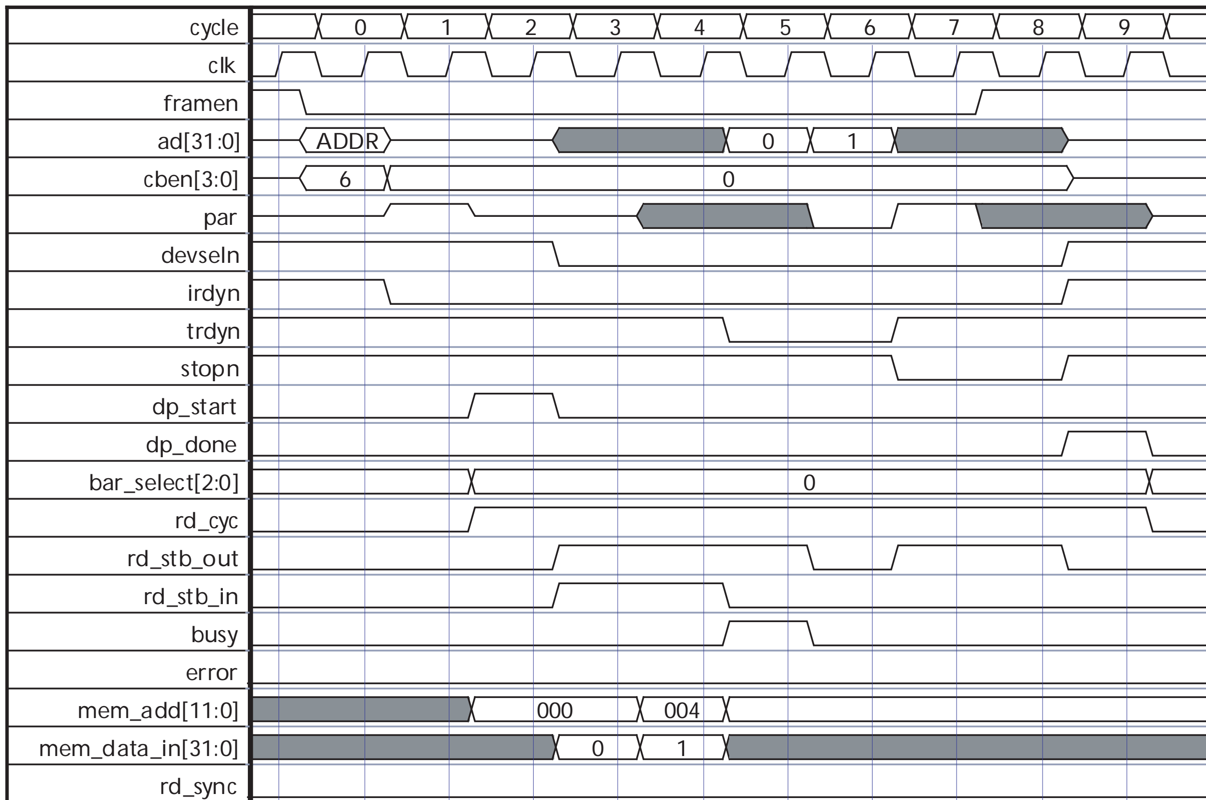
8.11 Backend-Terminated (BUSY) Cycle during Data Burst (Target)

If the backend runs out of data during a burst transfer (a FIFO empties), the backend can terminate the transfer by asserting BUSY. The core will then terminate the PCI cycle with a disconnect with or without data, depending on the state of the internal data pipe. The core will delay asserting the PCI STOPN signal until the internal FIFO is empty. It is recommended that the RD_STB_IN signal be deasserted once BUSY is asserted. Otherwise, the core will continue to accept data from the backend and transfer it on the PCI bus.

If the backend does not assert the RD_STB_IN or WR_BE_RDY signals within the eight-clock-cycle requirement, the PCI core will automatically terminate the PCI cycle.

Downloaded from Arrow.com.

Figure 37 • Backend Burst Read Cycle Terminated by BUSY (RD_SYNC = 0)



Once asserted, BUSY may be deasserted, as in Figure 37, or left asserted until the backend has data available. This would cause any subsequent PCI cycles to be terminated with a retry.

Figure 38 shows BUSY being asserted when RD_SYNC is active. Figure 39 shows BUSY asserted during a write cycle. In the case shown, the backend interface had previously indicated that it was ready to accept data by asserting WR_BE_RDY before asserting BUSY, causing two data words to be written before the PCI cycle was stopped.

Downloaded from Arrow.com.

Figure 38 • Backend Burst Read Cycle Terminated by BUSY (RD_SYNC = 1)

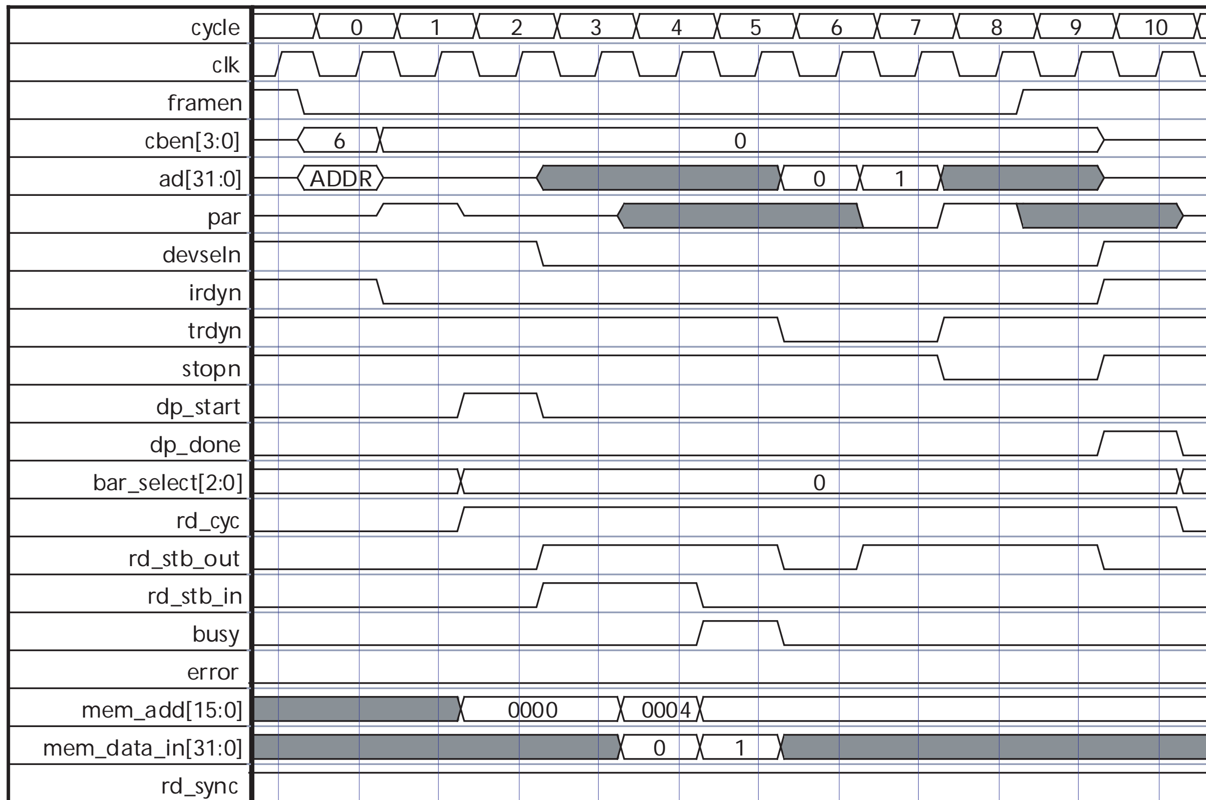
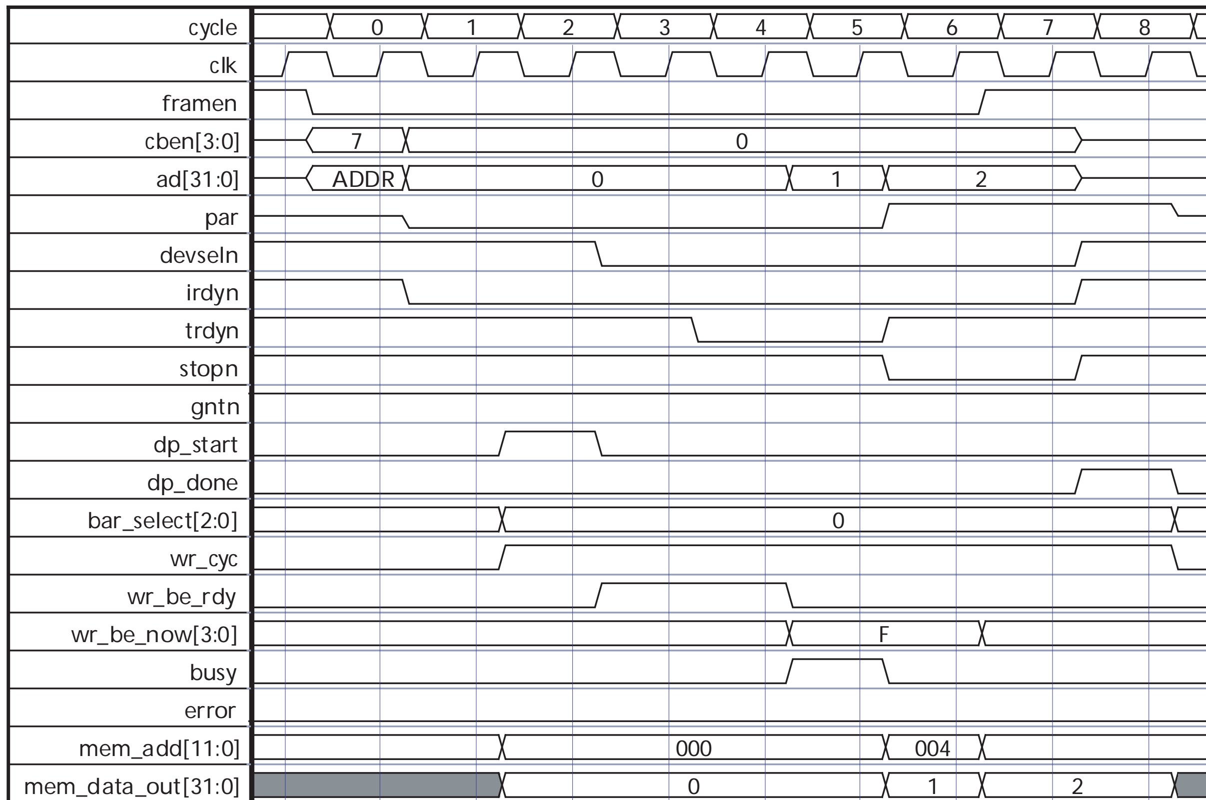


Figure 39 • Backend Burst Write Cycle Terminated by BUSY



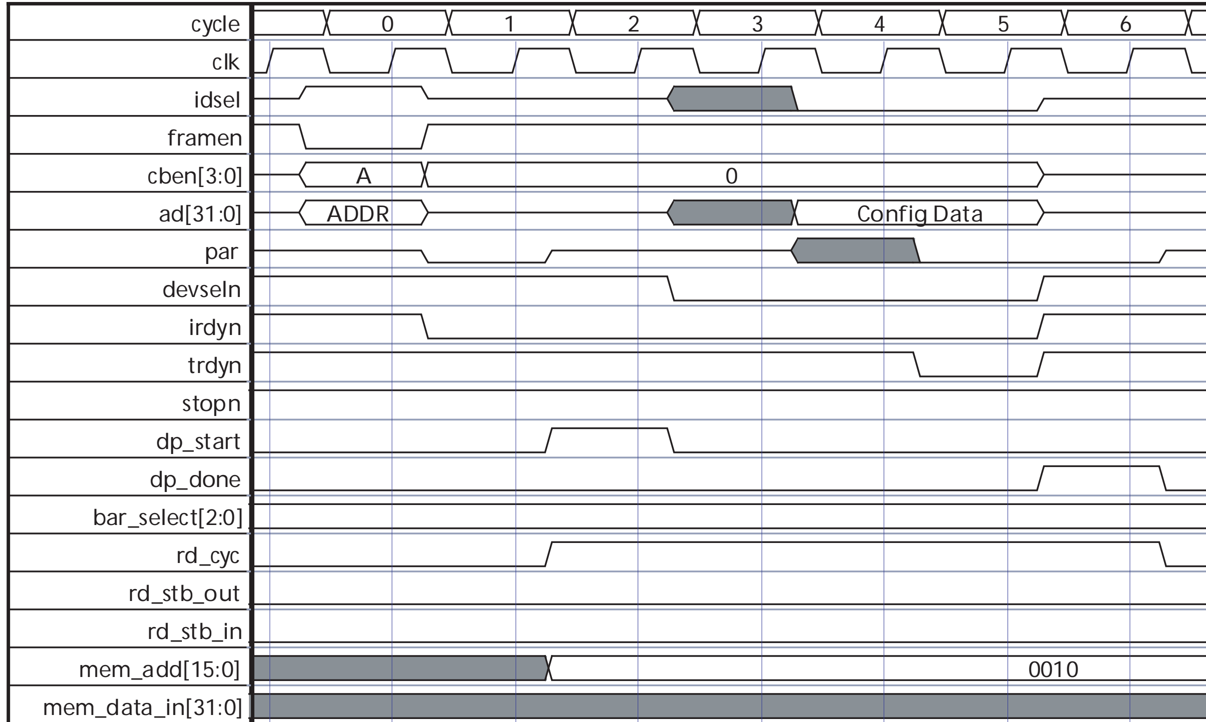
Downloaded from Arrow.com.

8.12 PCI Configuration Cycle

The core handles PCI configuration cycles without any backend involvement. The core does not allow configuration space to be added to the backend logic.

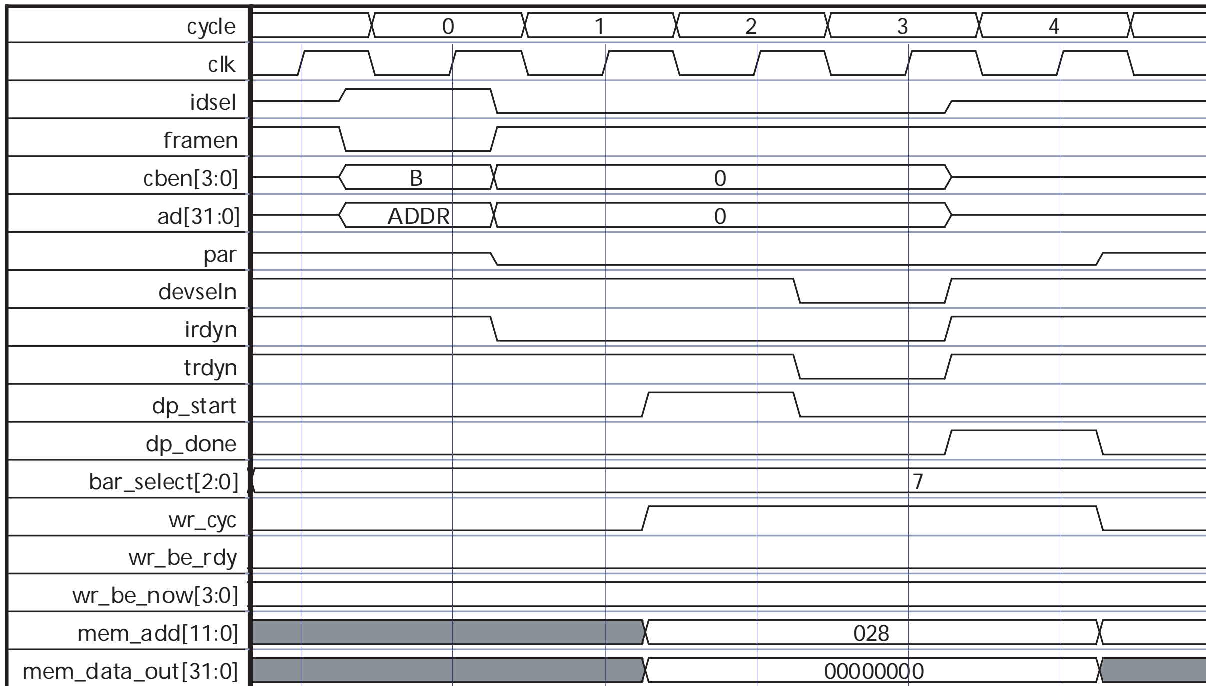
During a configuration cycle, the DP_START and DP_DONE outputs will pulse when a configuration cycle occurs, and the BAR_SELECT output will remain inactive. Figure 40 and Figure 41 show the BAR_SELECT output as inactive ('111').

Figure 40 • **Configuration Read Cycle**



Downloaded from Arrow.com.

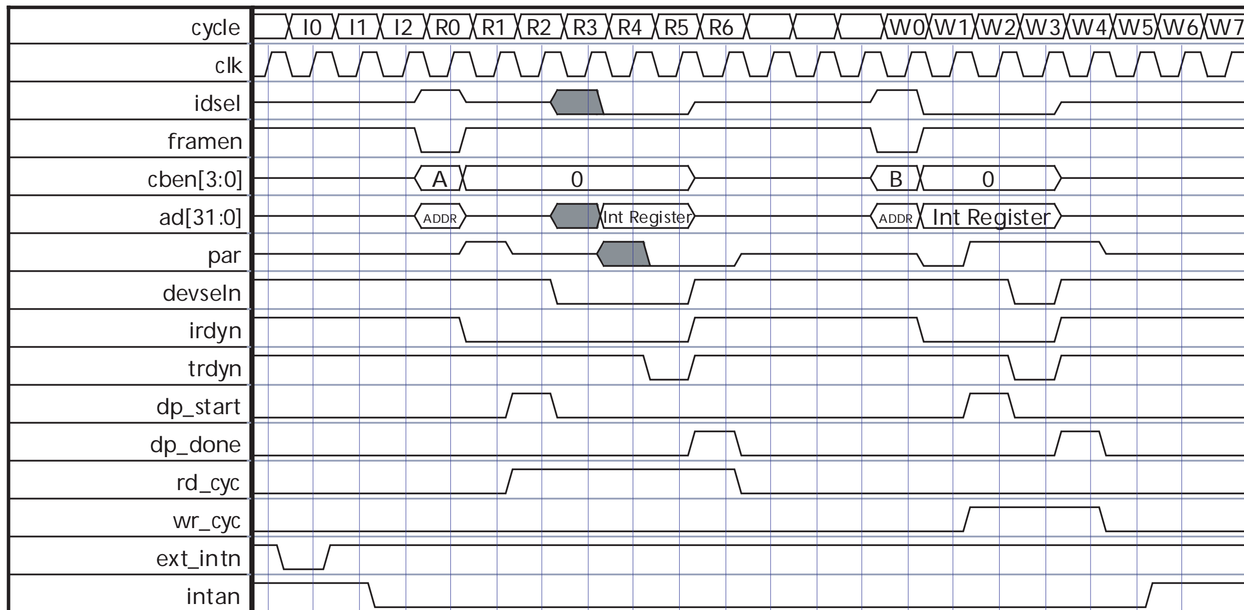
Figure 41 • **Configuration Write Cycle**



8.13 PCI Interrupt Generation

To initiate an interrupt, the backend asserts the EXT_INTN input (Figure 42). Two cycles later, the PCI INTAN interrupt signal is asserted.

Figure 42 • **PCI Interrupt Generation and Acknowledge Sequence**



When INTAN is asserted, the Master can read the Interrupt Status register to verify which device is driving INTAN (cycles R0–R6). Once it has determined which device and the reason for the interrupt, it writes to the requesting Interrupt Control/Status register to clear the interrupt request (cycles W0–W6).

Downloaded from Arrow.com.

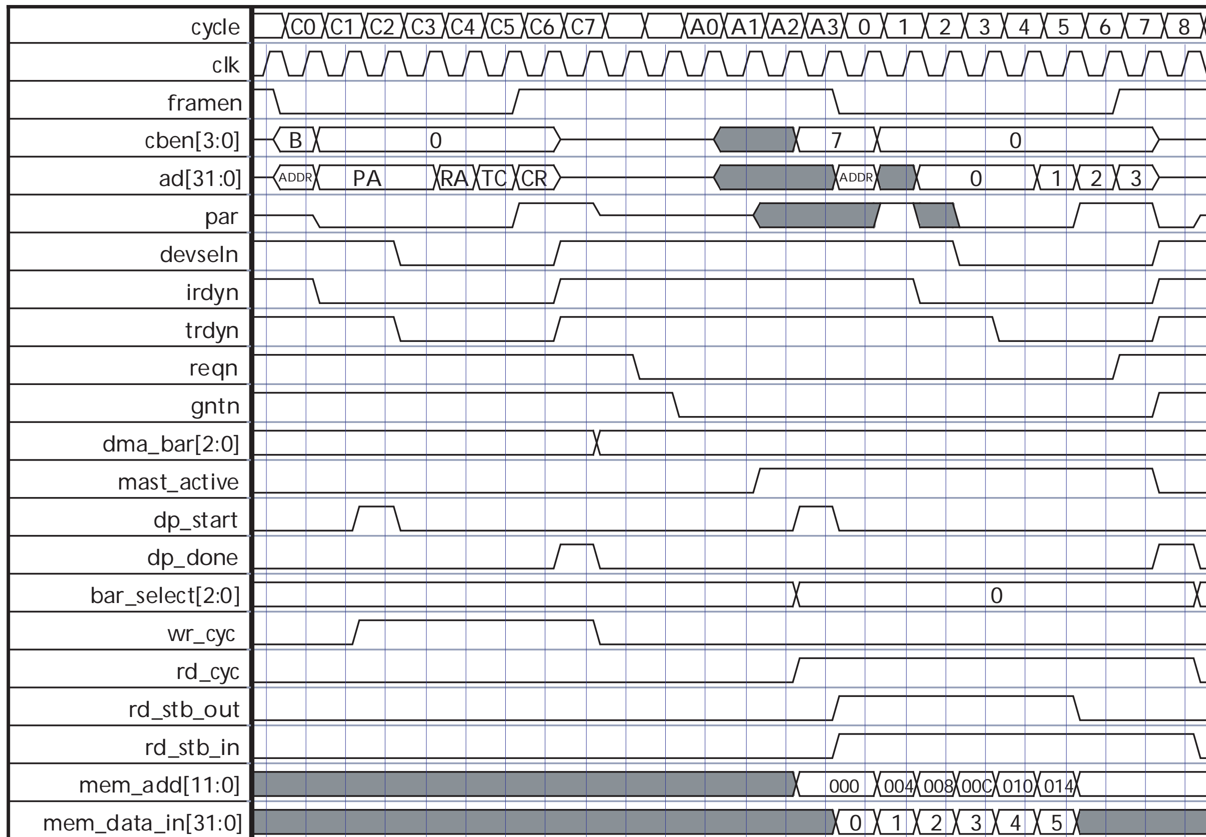
Microsemi
Power Matters.

The Interrupt Control/Status register can be accessed through the configuration space or through a memory BAR if the DMA registers are mapped to memory space.

8.14 Simple DMA Transfer

Initially, a PCI Master writes to the four DMA control registers (cycles C0 to C7 in Figure 43). Three clock cycles after the control register (CR) is written, the core asserts its PCI request signal REQN. When the PCI arbiter grants the bus and the bus is idle, several clock cycles later the core initiates a PCI cycle asserting the MAST_ACTIVE output. See cycle A2 in Figure 43.

Figure 43 • **DMA Burst Read Cycle Including DMA Start Sequence**

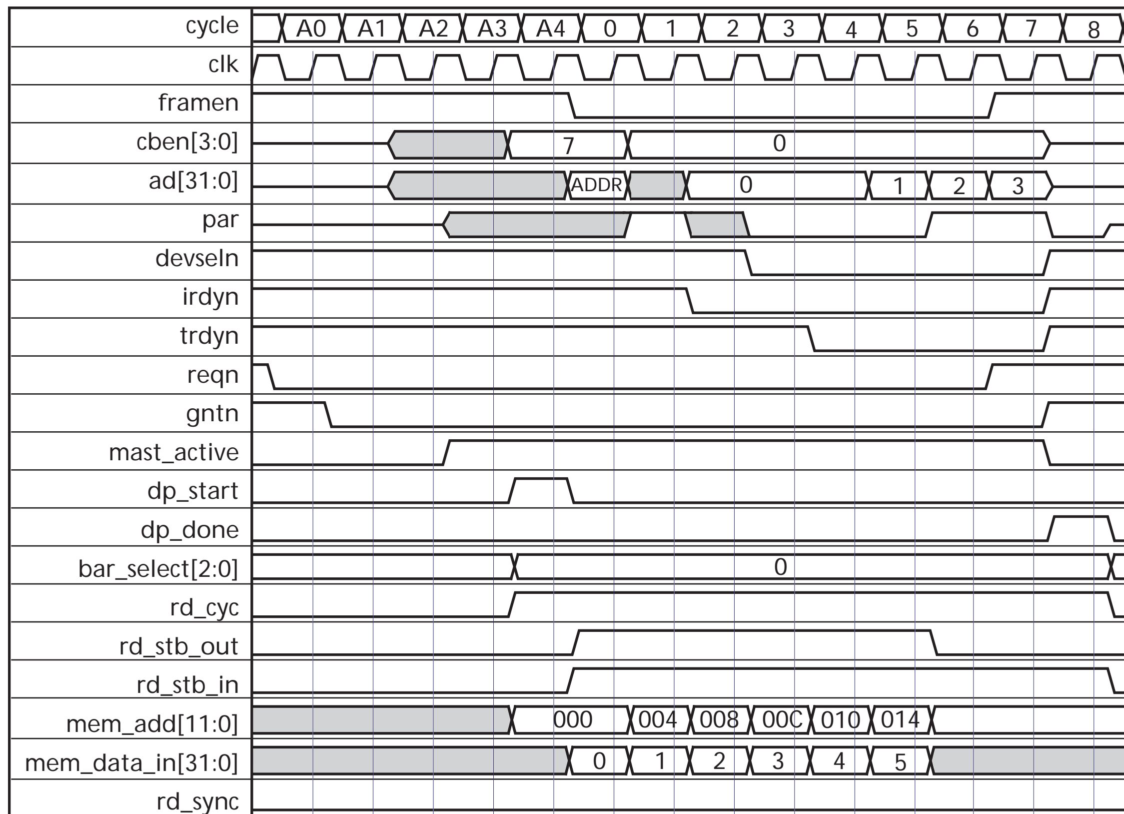


Initially, the core turns on its AD and CBE outputs at the same time that it initiates a backend cycle. The backend transfer is very similar to a Target transfer, except that the MAST_ACTIVE output is valid during the transfer. The BAR_SELECT output will be set to the value set in the DMA control register.

The core delays asserting FRAMEN during this period to allow the backend interface to become ready. During this period, the core puts the correct values on the AD and CBEN busses and sets the bus parity.

Downloaded from Arrow.com.

Figure 44 • **DMA Burst Read Cycle (RD_SYNC = 0)**



After FRAMEN has been asserted, the data transfer proceeds normally. The PCI specification requires a Master to assert IRDYN within eight clock cycles of FRAMEN assertion. For read transfers, this means that the backend must provide data within eight clock cycles of DP_START being asserted. When RD_SYNC = 0 or RD_SYNC = 1, the backend only has seven clock cycles to assert RD_STB_IN and meet the PCI latency requirements (Figure 43 and Figure 44). For SX-A and RTSX-S implementations, this is reduced to six cycles. For write transfers, the backend must assert WR_BE_NOW within eight clock cycles to meet the PCI requirements.

During the DMA startup period, the PCI arbiter may remove the bus grant before the core asserts FRAMEN. When this occurs, the PCI core is required to terminate its DMA cycle. This is shown in Figure 46, where the grant is removed after one cycle.

The core provides four additional input signals that are used to control DMA transfers: WR_BUSY_MASTER, RD_BUSY_MASTER, STOP_MASTER, and STALL_MASTER. STOP_MASTER allows a DMA transfer in progress to be stopped. WR_BUSY_MASTER and RD_BUSY_MASTER prevent DMA writes to and reads from the backend from starting. STALL_MASTER allows slow backends to meet the FRAME-to-IRDY assertion requirement for PCI.

Downloaded from Arrow.com.

Figure 45 • **DMA Burst Read Cycle (RD_SYNC = 1)**

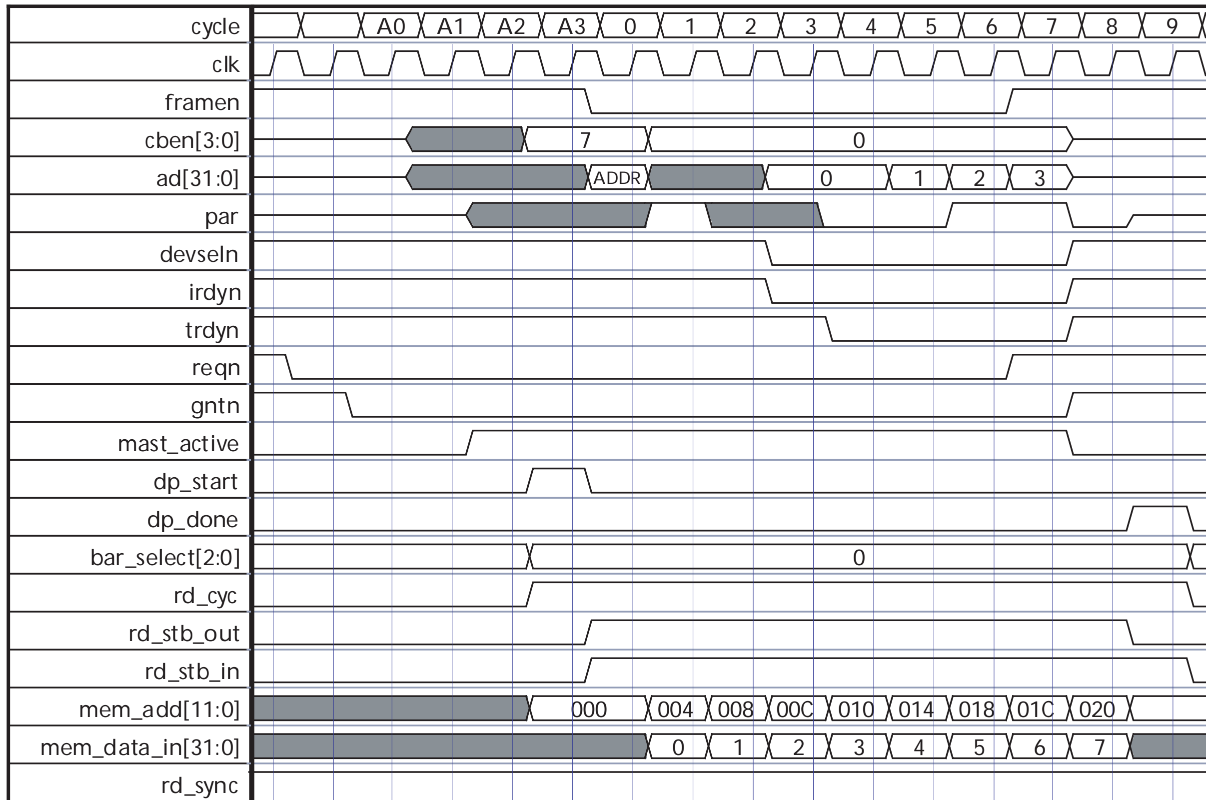
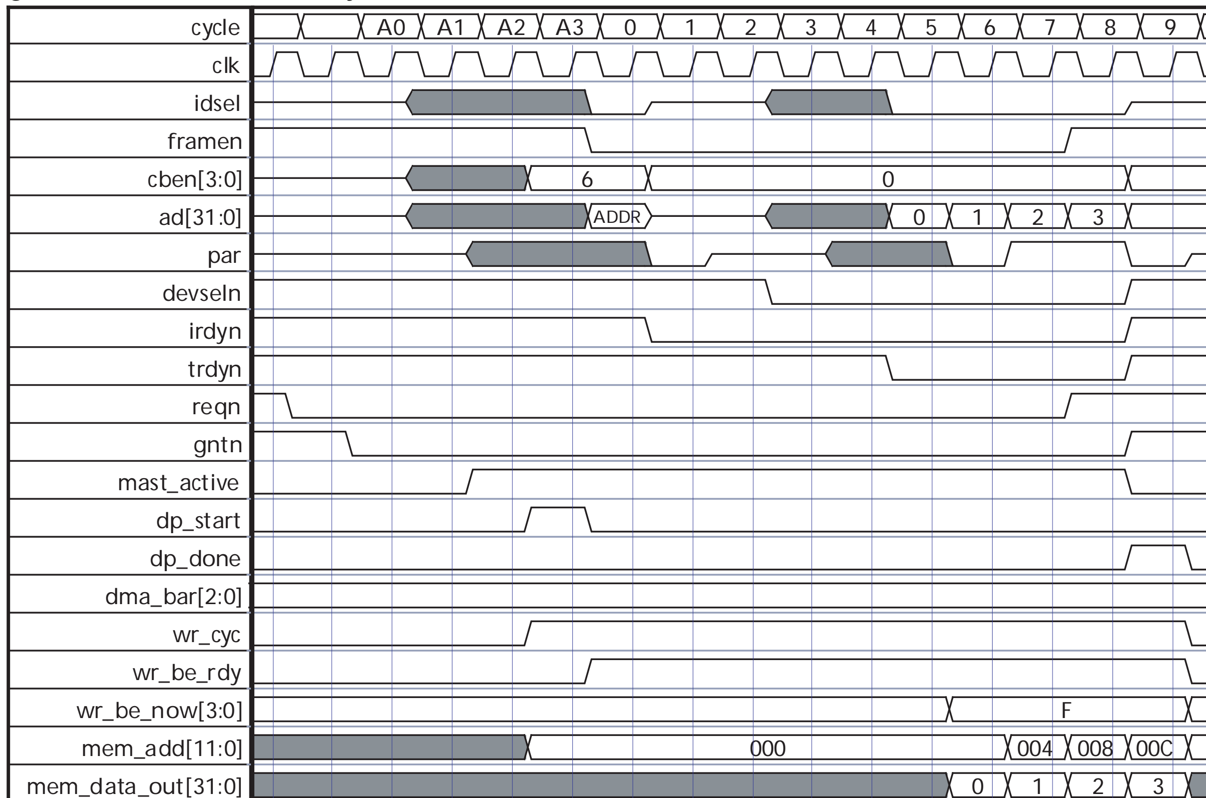


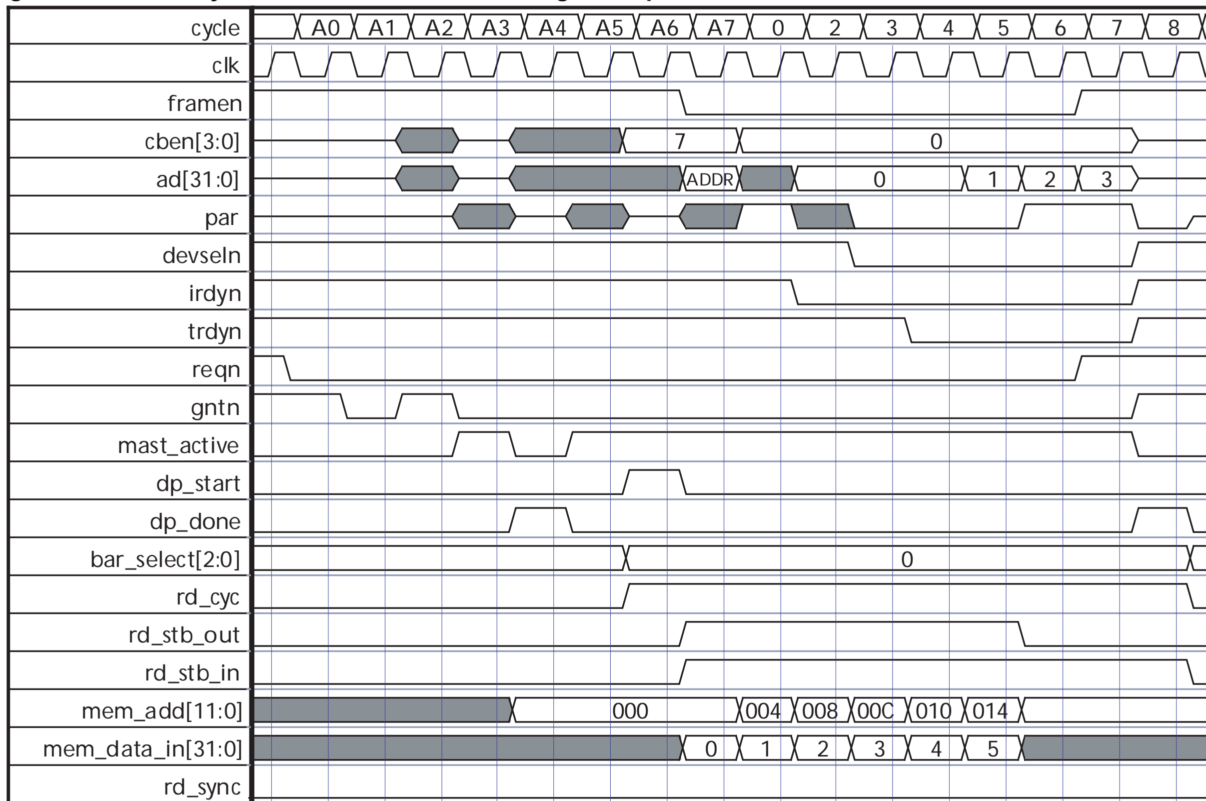
Figure 46 • **DMA Burst Write Cycle**



Downloaded from Arrow.com.

During the DMA burst write cycles, it is normal for an additional write cycle, such as cycle nine in Figure 46, to take place one clock cycle after MAST_ACTIVE has been deasserted. If necessary, MAST_ACTIVE can be delayed by a clock cycle externally to generate a version that will still be active when the last write occurs.

Figure 47 • **DMA Cycle with Grant Removal during Startup**

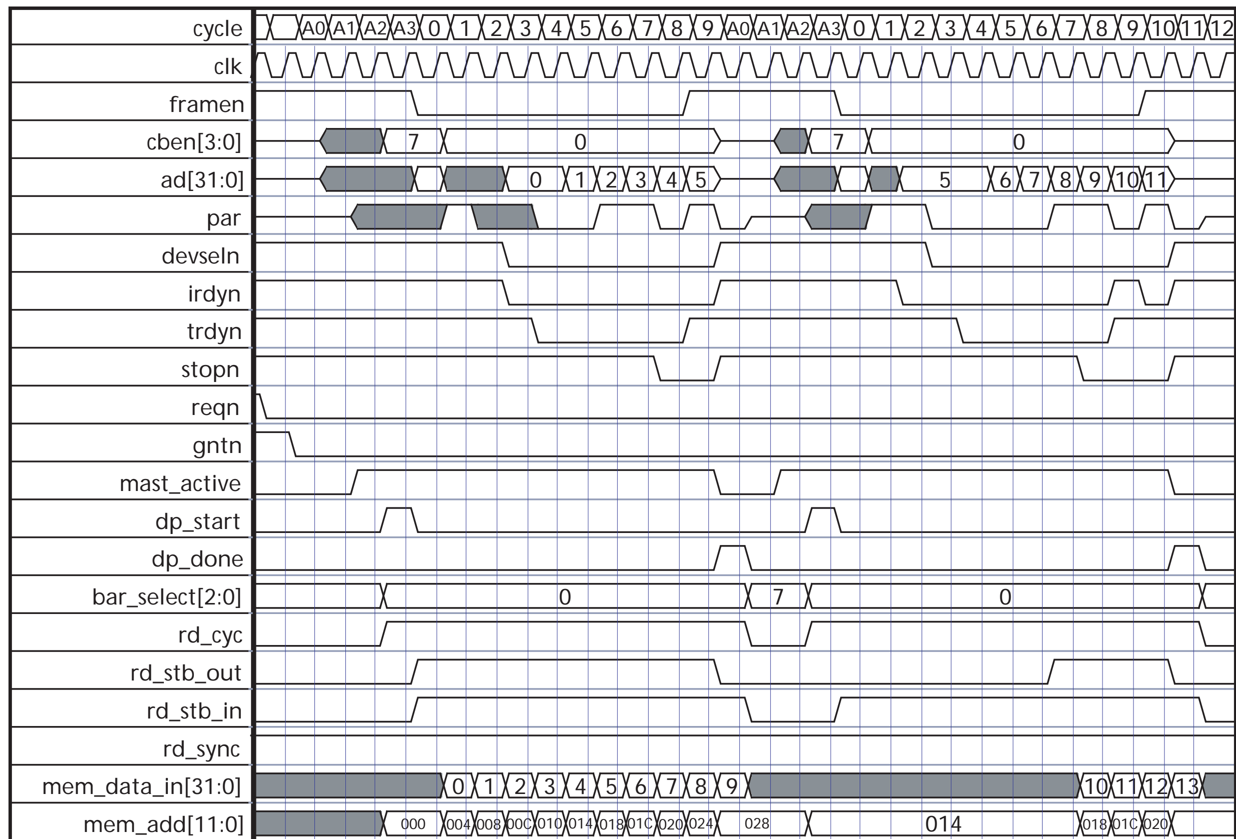


8.15 DMA Operation with a FIFO Backend

Figure 48 shows a DMA Master transfer from a backend FIFO to the PCI bus. During the transfer, the Target asserts STOPN, causing the Master to stop the transfer. During the first PCI cycle, the core reads data words 0 to 9 from the backend but only transfers 0 to 4 on the PCI bus. During the second PCI cycle, the core reads data words 10 to 13 from the backend, but transfers 5 to 9 on the PCI bus. These data words had been stored inside the core between data transfers. Words 10 to 13 are stored inside the core and will be transferred during the next PCI cycle.

Downloaded from Arrow.com.

Figure 48 • **DMA Cycle with a FIFO Backend**



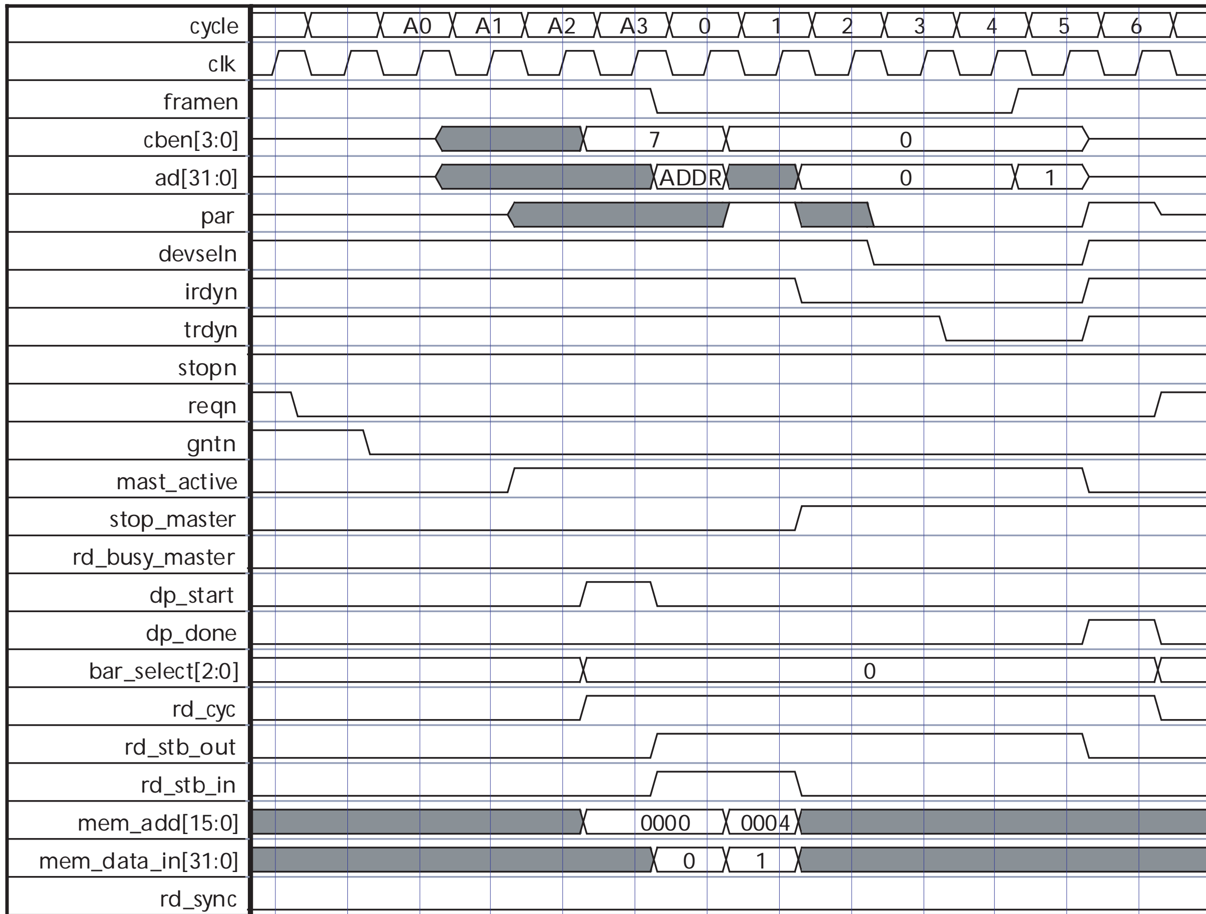
8.16 STOP_MASTER Assertion during Data Burst

If the backend asserts STOP_MASTER when a DMA transfer is taking place, the core will stop the DMA transfer as soon as possible, as shown in Figure 49 to Figure 51. Due to PCI protocol requirements, the core may need to transfer one or two additional words after STOP_MASTER has been asserted. The additional data is required to complete the PCI transfer, as when the core deasserts FRAMEN and asserts IRDYN, valid data must be provided on the bus. See cycle nine in Figure 49.

Figure 49 shows STOP_MASTER being asserted during a DMA read operation; in this case, no additional data is required after STOP_MASTER is asserted. On cycles zero and one, the core reads two words of data. The first of these words is transferred on the PCI bus on cycle four when the Target asserts TRDYN. During cycle five, the core needs to deassert FRAMEN and assert IRDYN, since STOP_MASTER has been asserted. This can occur because the second data word is stored in the core, allowing IRDYN to be asserted.

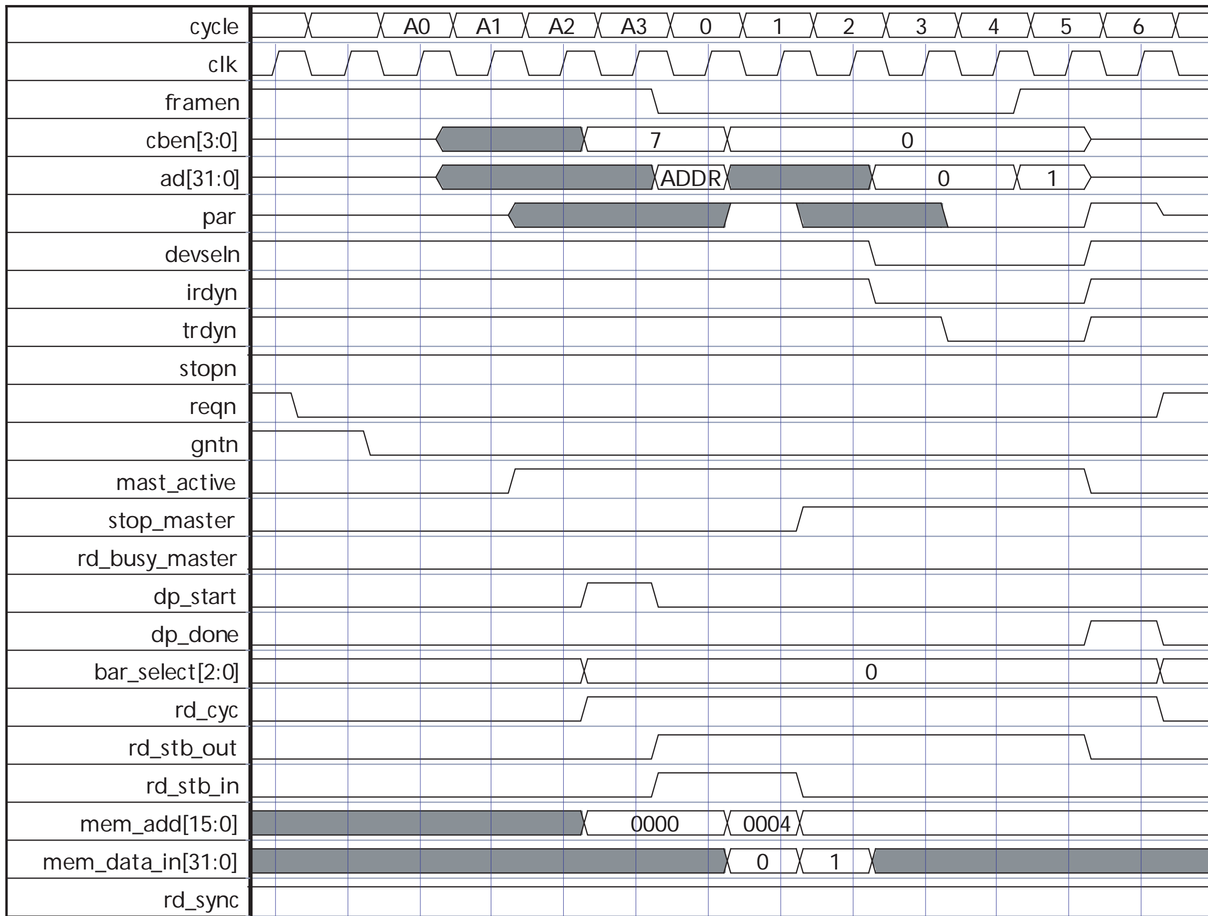
Downloaded from Arrow.com.

Figure 49 • **STOP_MASTER Assertion during DMA Burst Read Cycle (RD_SYNC = 0)**



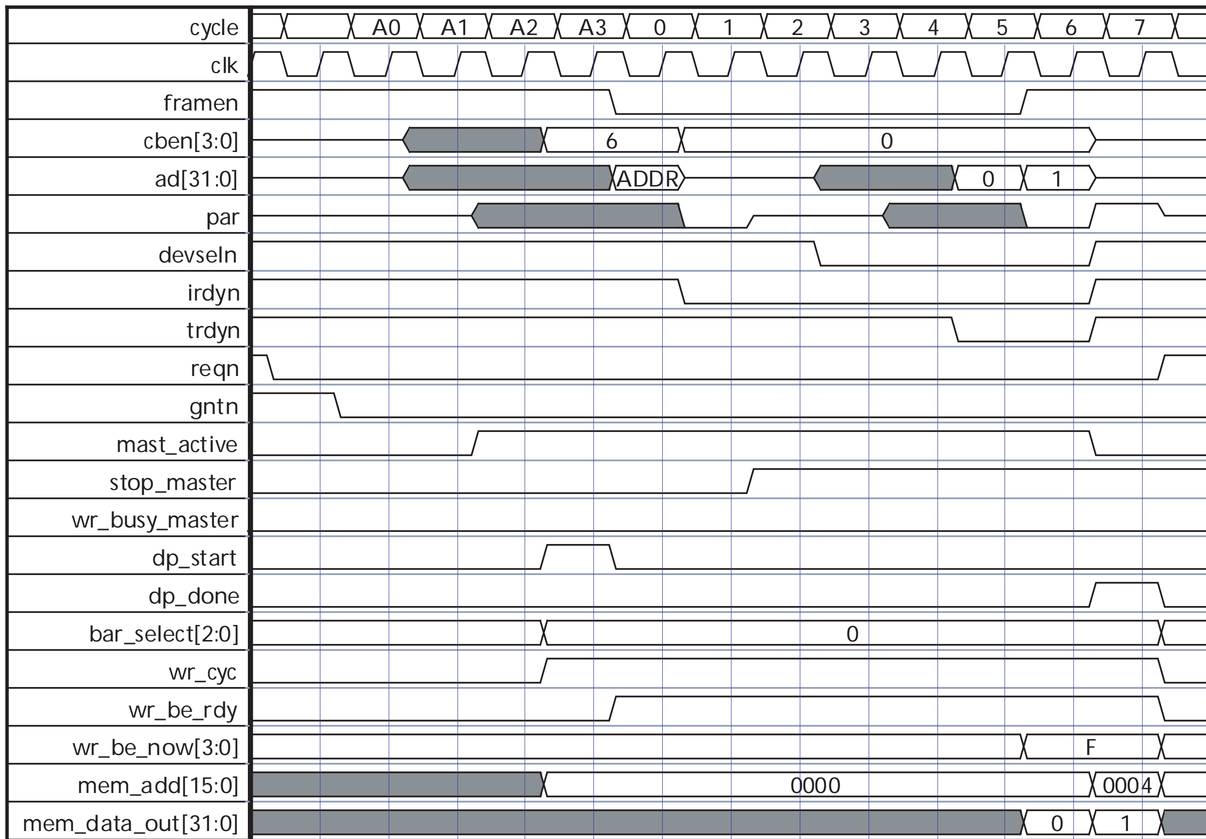
Downloaded from Arrow.com.

Figure 50 • **STOP_MASTER Assertion during DMA Burst Read Cycle (RD_SYNC = 1)**



Downloaded from Arrow.com.

Figure 51 • **STOP_MASTER Assertion during DMA Burst Write Cycle**

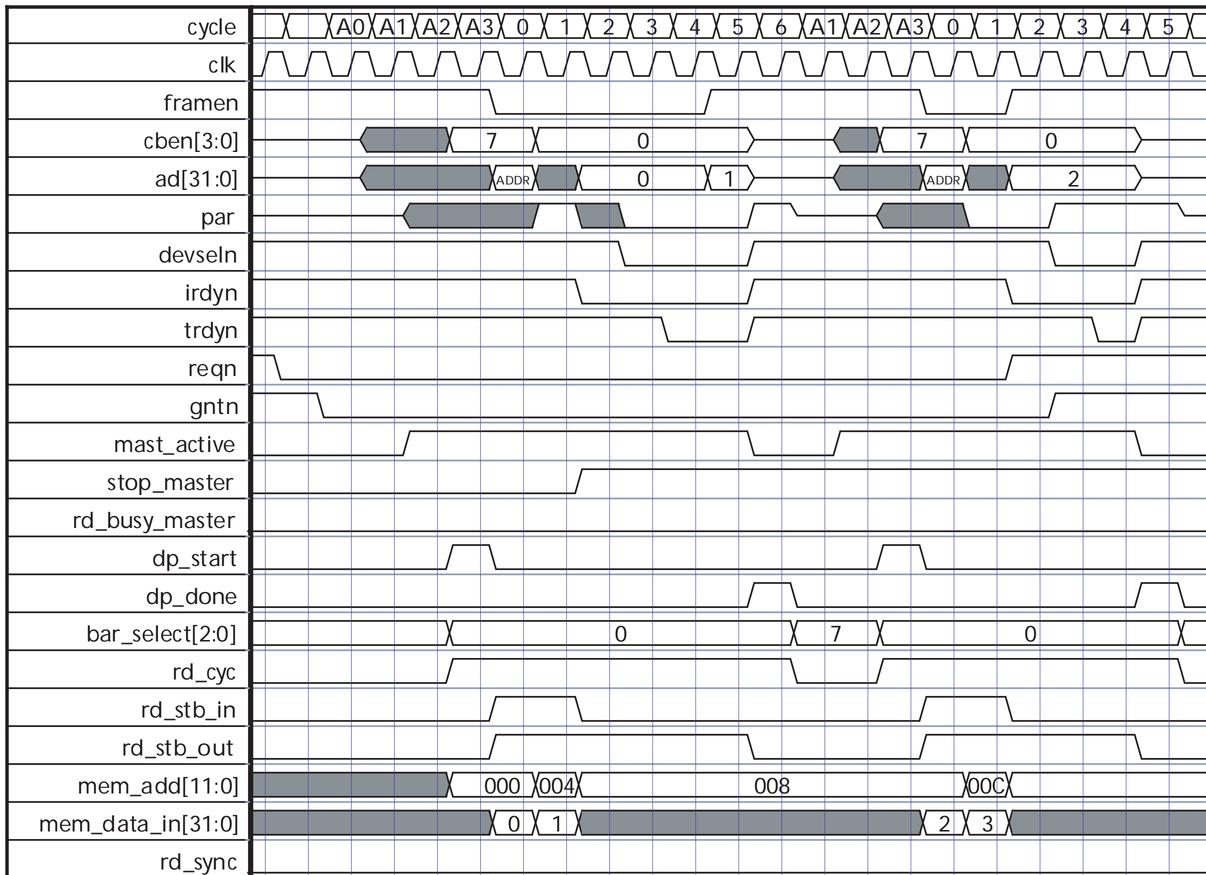


In the DMA write case, the core will always need to transfer additional data after STOP_MASTER is asserted. Depending on the state of the transfer, one or two additional words may be transferred after STOP_MASTER assertion.

If STOP_MASTER is held asserted, the core will start a DMA cycle and terminate after one word has been transferred. Figure 52 shows a DMA cycle being terminated by STOP_MASTER and a subsequent DMA cycle transferring a single word. To prevent the DMA cycle in Figure 52 from starting, the RD_BUSY_MASTER and WR_BUSY_MASTER inputs can be used.

Downloaded from Arrow.com.

Figure 52 • **STOP_MASTER Held Asserted during DMA Burst Read**



8.17 RD_BUSY_MASTER and WR_BUSY_MASTER Operation

The RD_BUSY_MASTER and WR_BUSY_MASTER inputs can be used to prevent a DMA operation from starting until the backend is able to accept or provide data. For instance, these inputs could be tied to FIFO empty and almost full flags, respectively. In this case, DMA read transfers would not be started if an external FIFO were empty, and DMA write transfers would not be started if an external FIFO were almost full.

If multiple memory interfaces are implemented, each with a FIFO, the DMA_BAR output indicates which BAR memory space the DMA controller is accessing. The appropriate FIFO empty signal should be multiplexed onto the RD_BUSY_MASTER input; likewise for WR_BUSY_MASTER.

Figure 53 shows a DMA read cycle in which RD_BUSY_MASTER is initially active. When it goes inactive (cycle R0) the core starts the DMA cycle by asserting the PCI bus request in cycle R1. The DMA transfer then progresses normally.

Downloaded from Arrow.com.

Figure 53 • **RD_BUSY_MASTER Operation**

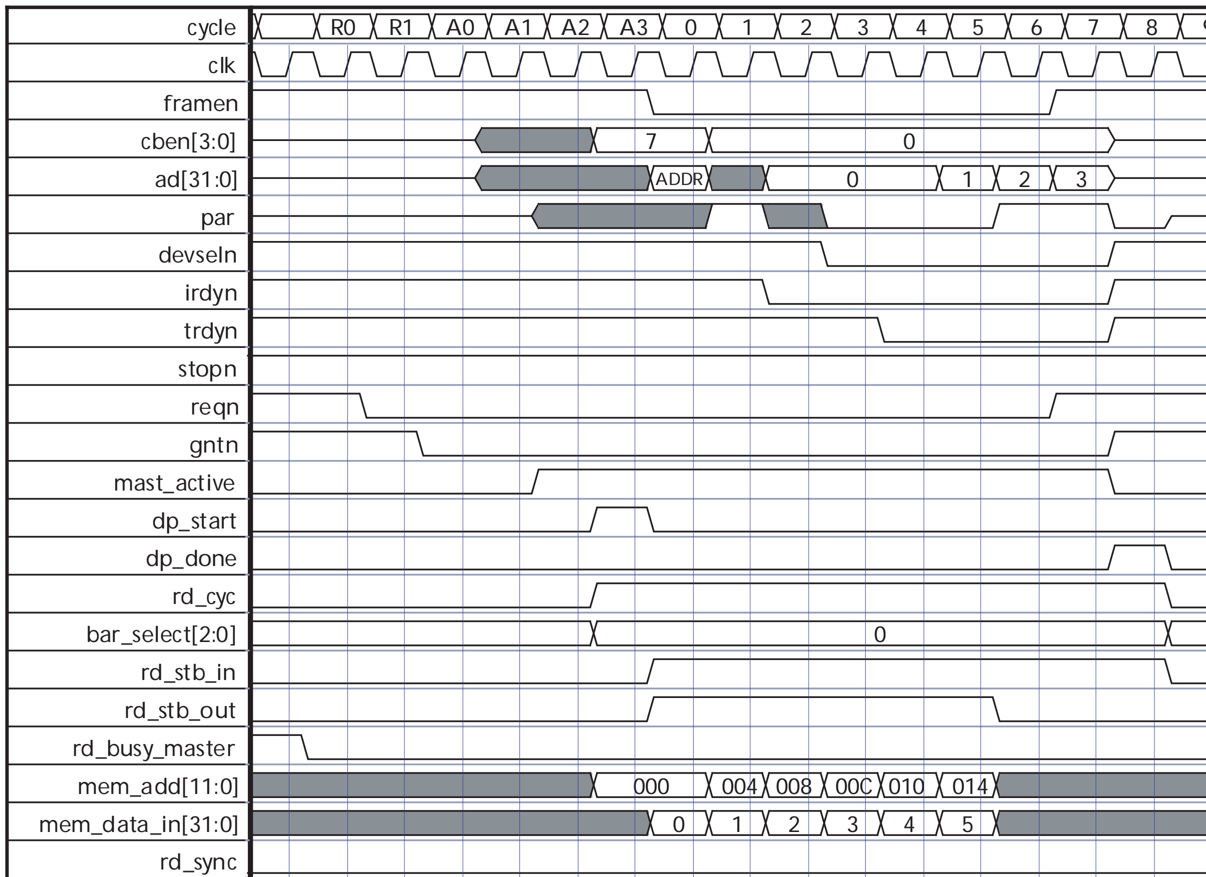
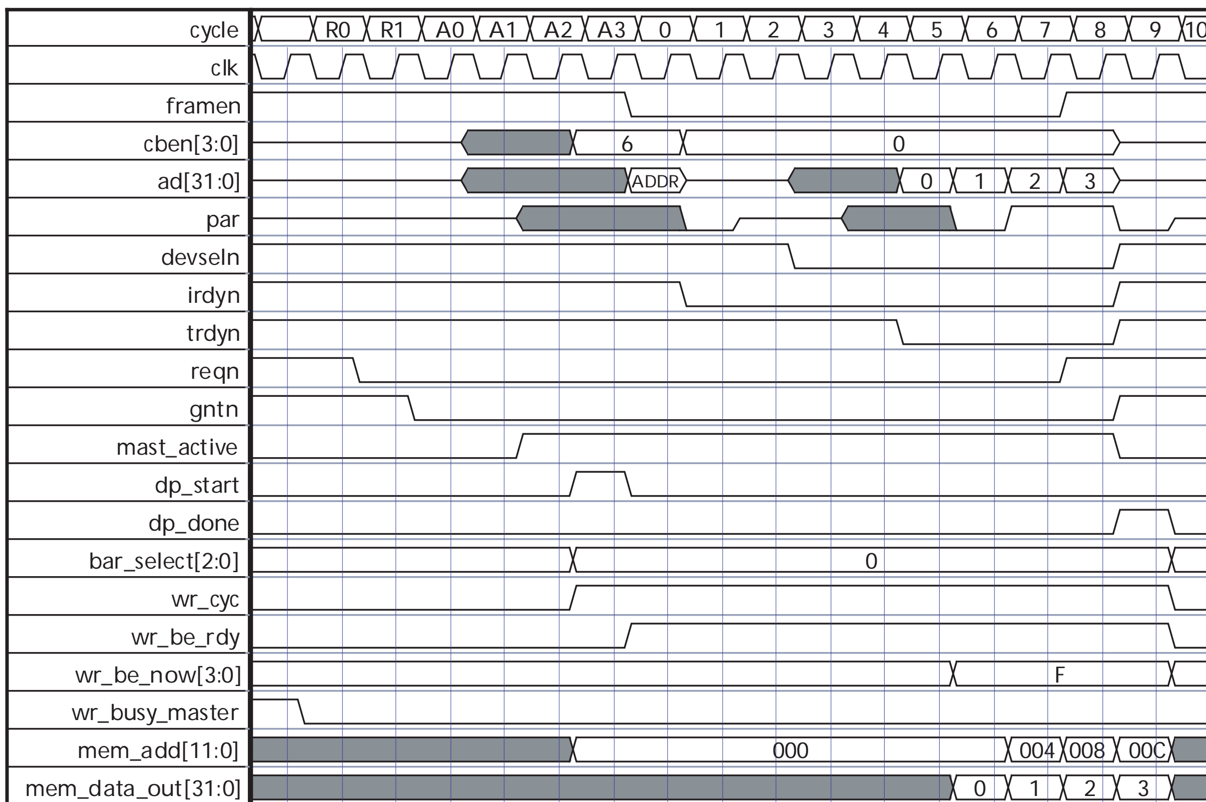


Figure 54 shows the equivalent DMA write cycle. In this case, the DMA cycle starts when WR_BUSY_MASTER is deasserted in cycle R0.

Downloaded from Arrow.com.

Figure 54 • **WR_BUSY_MASTER Operation**



These two inputs and STOP_MASTER can be used to start and stop a DMA transfer under hardware control. Once the DMA control registers have been programmed, the DMA can be started by deasserting the Master busy signals. It can be stopped by asserting the Master stop and busy inputs.

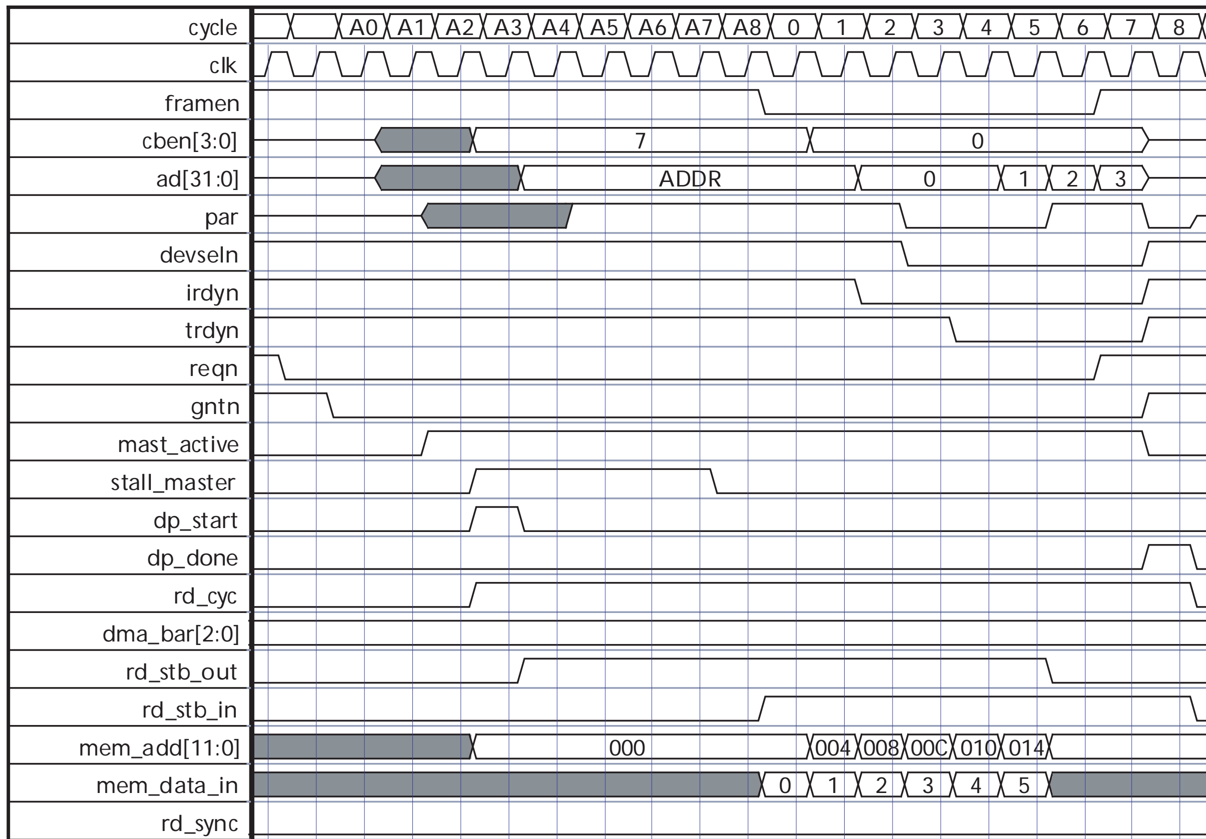
8.18 STALL_MASTER Operation

STALL_MASTER allows the backend to increase the number of clock cycles it is allowed from DP_START assertion to RD_STB_IN or WR_BE_RDY assertion for DMA transfers. As described in **Simple DMA Transfer**, the PCI specification requires a Master to assert IRDYN within eight clock cycles of FRAMEN, so the backend logic must assert these inputs within eight cycles of DP_START.

When STALL_MASTER is asserted, the core will delay the assertion of FRAMEN while the backend becomes ready. STALL_MASTER must be asserted on the clock cycle after MAST_ACTIVE becomes active, at the same time the core asserts DP_START. The core will then assert FRAMEN two clock cycles after STALL_MASTER is deasserted (with STALL_MODE = 0), and IRDYN will be asserted two clock cycles after RD_STB_IN is asserted. This allows the backend to control the FRAMEN-to-IRDYN delay (see Figure 55 through Figure 57).

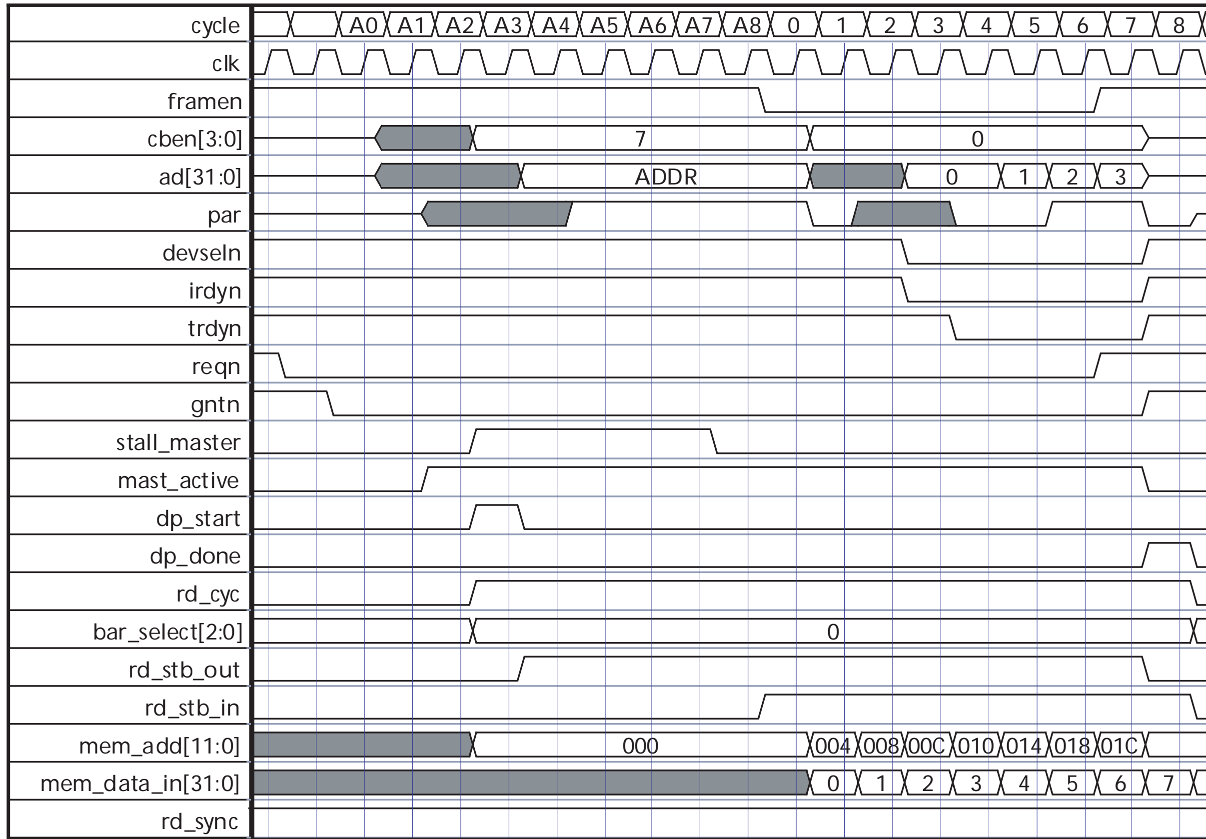
Downloaded from Arrow.com.

Figure 55 • **STALL_MASTER Assertion DMA Read Cycle (RD_SYNC = 0)**



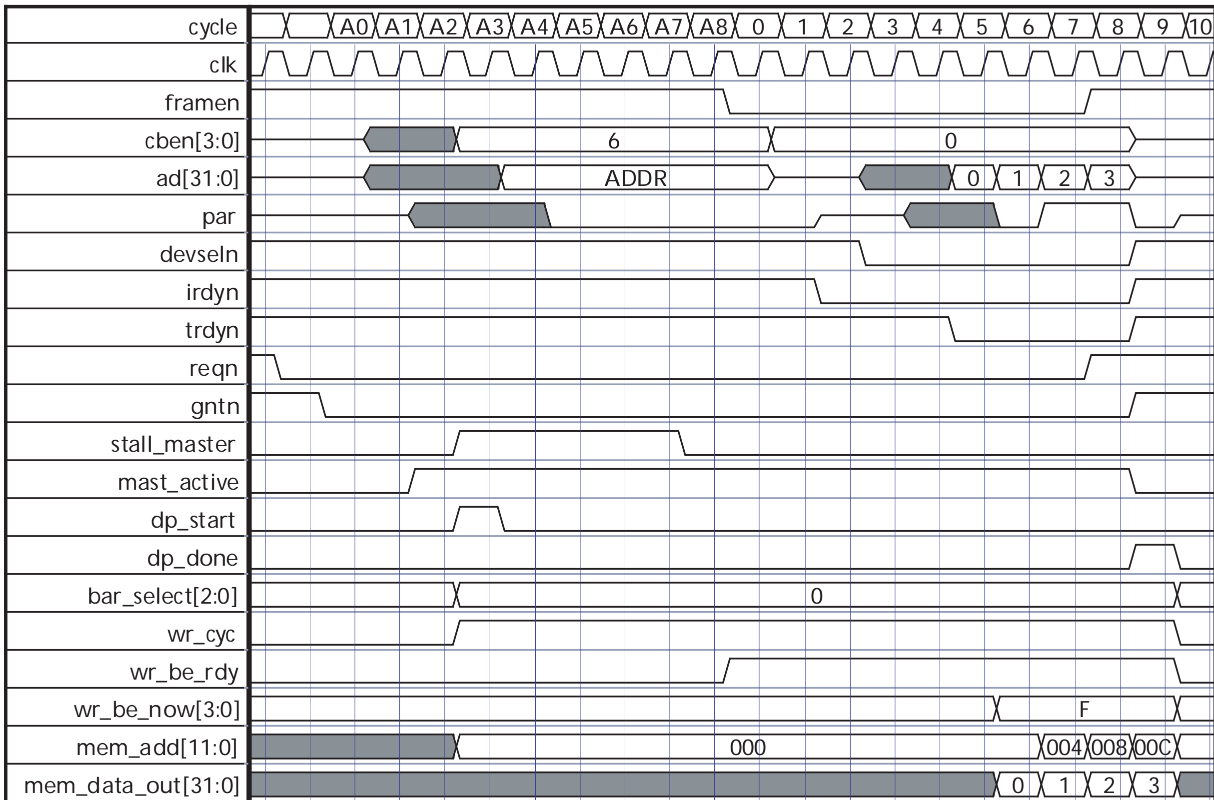
Downloaded from Arrow.com.

Figure 56 • **STALL_MASTER Assertion DMA Read Cycle (RD_SYNC = 1)**



Downloaded from Arrow.com.

Figure 57 • **STALL_MASTER Assertion DMA Write Cycle**



When STALL_MASTER is asserted, the likelihood that the PCI arbiter will remove the PCI bus grant signal during the DMA startup sequence is greatly increased. The bus is idle while the core delays asserting FRAMEN, and the arbiter may remove the bus grant if a higher priority device requests the bus. This will cause the core to terminate the DMA cycle. DP_DONE will be asserted, as shown in Figure 58.

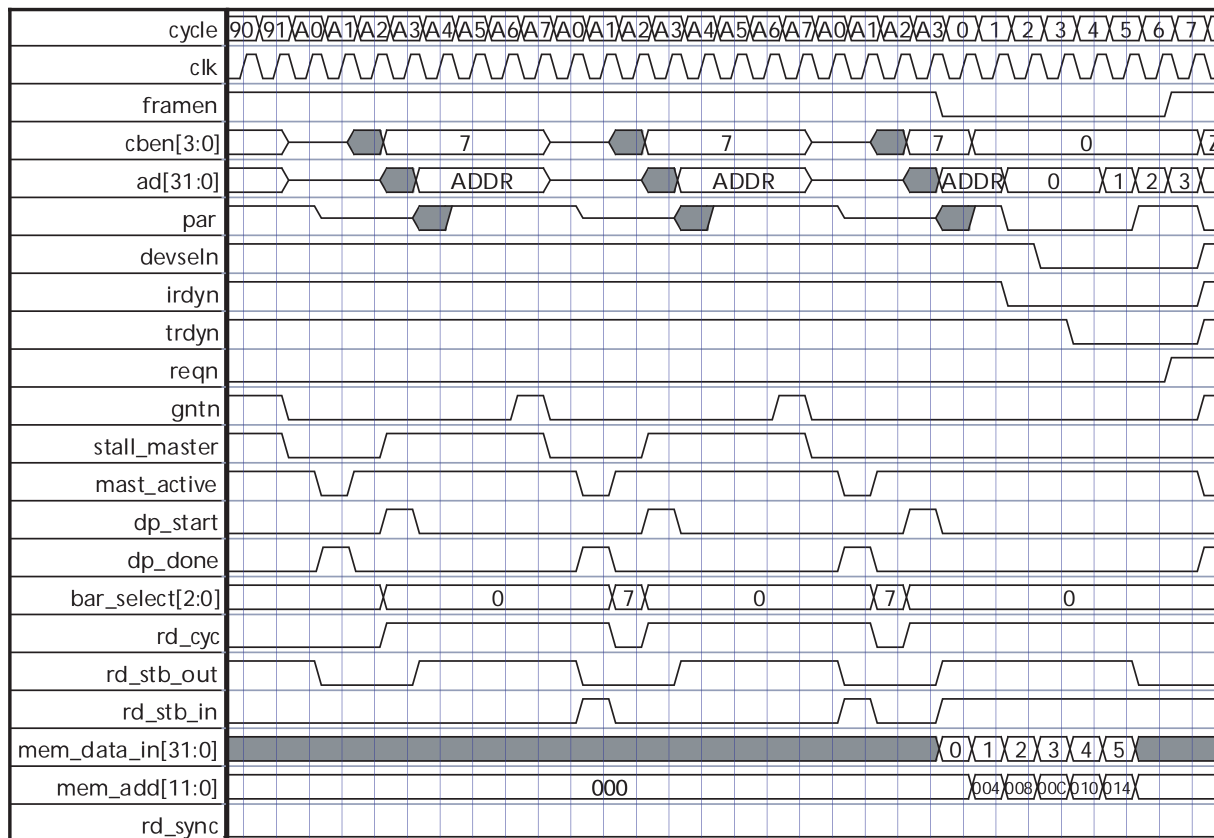
When GNTN is removed, the core aborts the current DMA cycle. The core will restart the cycle when the bus is regranted. At that time, it will reread the data from the backend. The core will not reread the data if FIFO recovery mode is enabled in the core. The core will continue to request the bus until it manages to start a cycle and FRAMEN is asserted.

In some systems, this could prevent the core from being given sufficient bus access, preventing data from being transferred at the expected rate.

STALL_MASTER must not be used to inject more than a 16-clock-cycle delay from DP_START to FRAMEN assertion. Doing so would violate the PCI specification. The core should assert FRAMEN within 16 clock cycles of GNTN assertion.

Downloaded from Arrow.com.

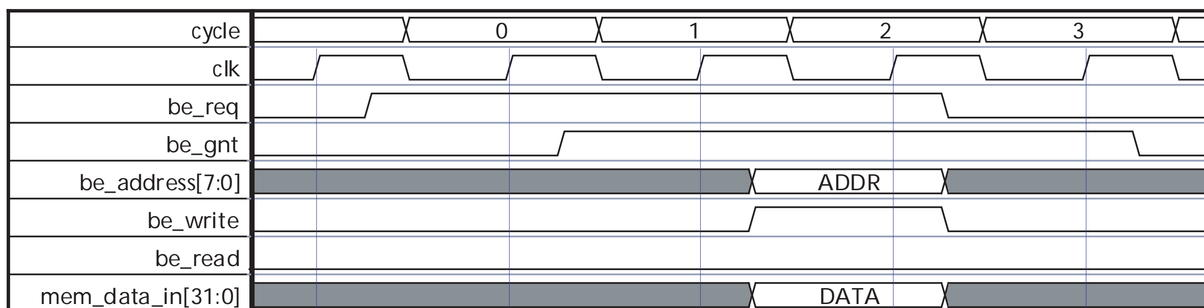
Figure 58 • STALL_MASTER Assertion and Cycle Aborted due to Loss of GNTN



8.19 DMA Register Access from the Backend

An interface is provided that allows the PCI configuration space and DMA registers to be accessed from the core backend rather than from the PCI bus. When the backend needs to access these registers, it must arbitrate for control of the core backend interface using the BE_REQ and BE_GNT handshake signals (Figure 59 to Figure 63). Once granted, it may access the DMA registers.

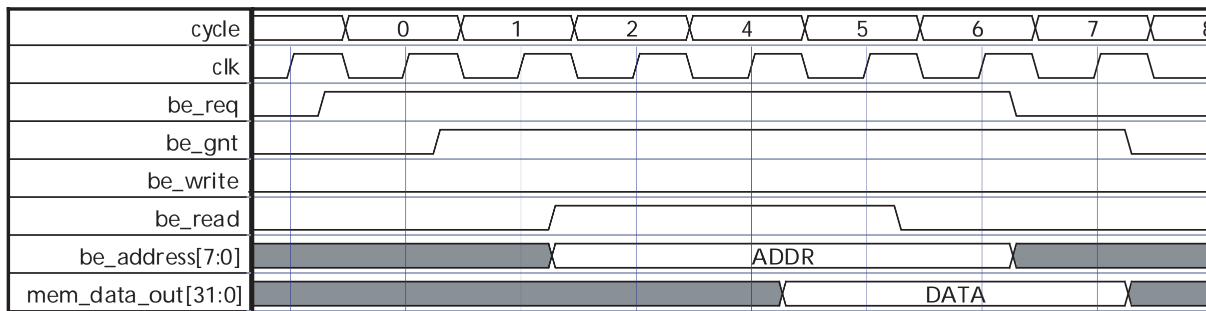
Figure 59 • DMA Register Single Write Cycle



Writes to the DMA register are accomplished by asserting BE_WRITE, valid address, and valid data at the same time. Registers can be updated one at a time or in bursts by changing the address and data while keeping BE_WRITE asserted. If required, BE_WRITE can be deasserted during a burst write. A separate 8-bit address bus, BE_ADDRESS, is provided. The data input uses the normal MEM_DATA_IN input. Four separate BE_WRITE signals are provided, one for each byte of the 32-bit wide registers.

Downloaded from Arrow.com.

Figure 60 • **DMA Register Single Read Cycle**



The registers may be read by asserting BE_READ and a valid address. Read data is pipelined with two cycles of delay between valid address and valid data. The data output shares the MEM_DATA_OUT output.

While the backend has control of the backend, BE_GNT = 1. The core will issue Target retry requests if a PCI Master attempts a Target access to the core. Therefore, the backend logic should not assert BE_REQ continuously; in that case, another Master will not be able to access the Target functions.

Figure 61 • **DMA Register Burst Write Cycle**

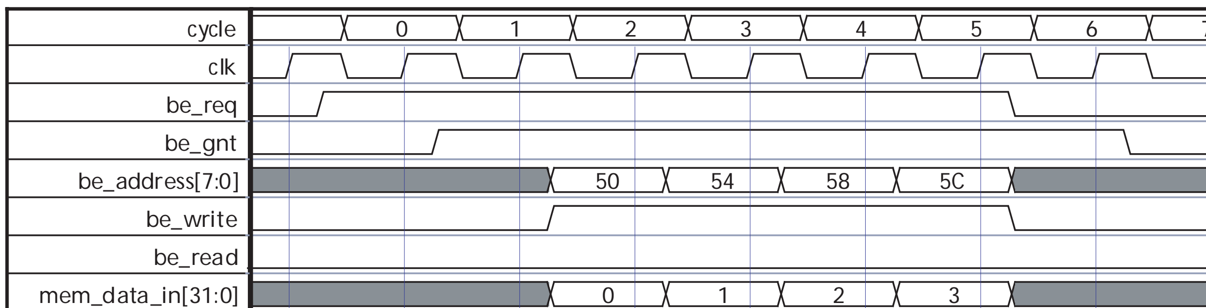


Figure 62 • **DMA Register Burst Read Cycle**

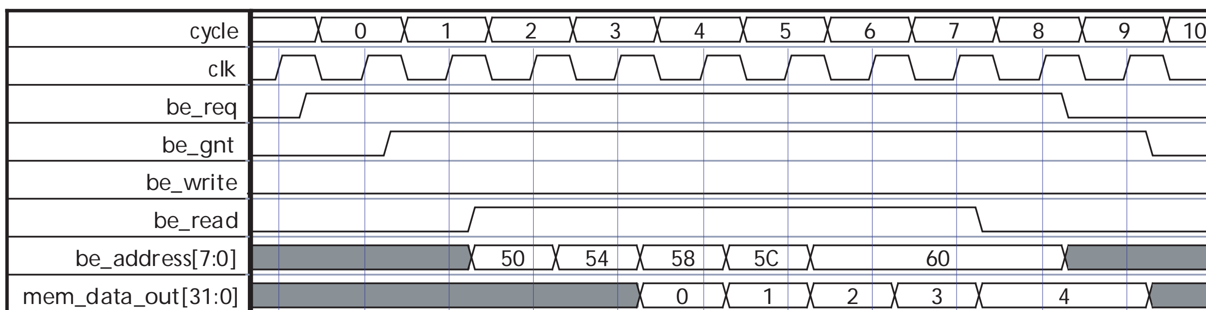
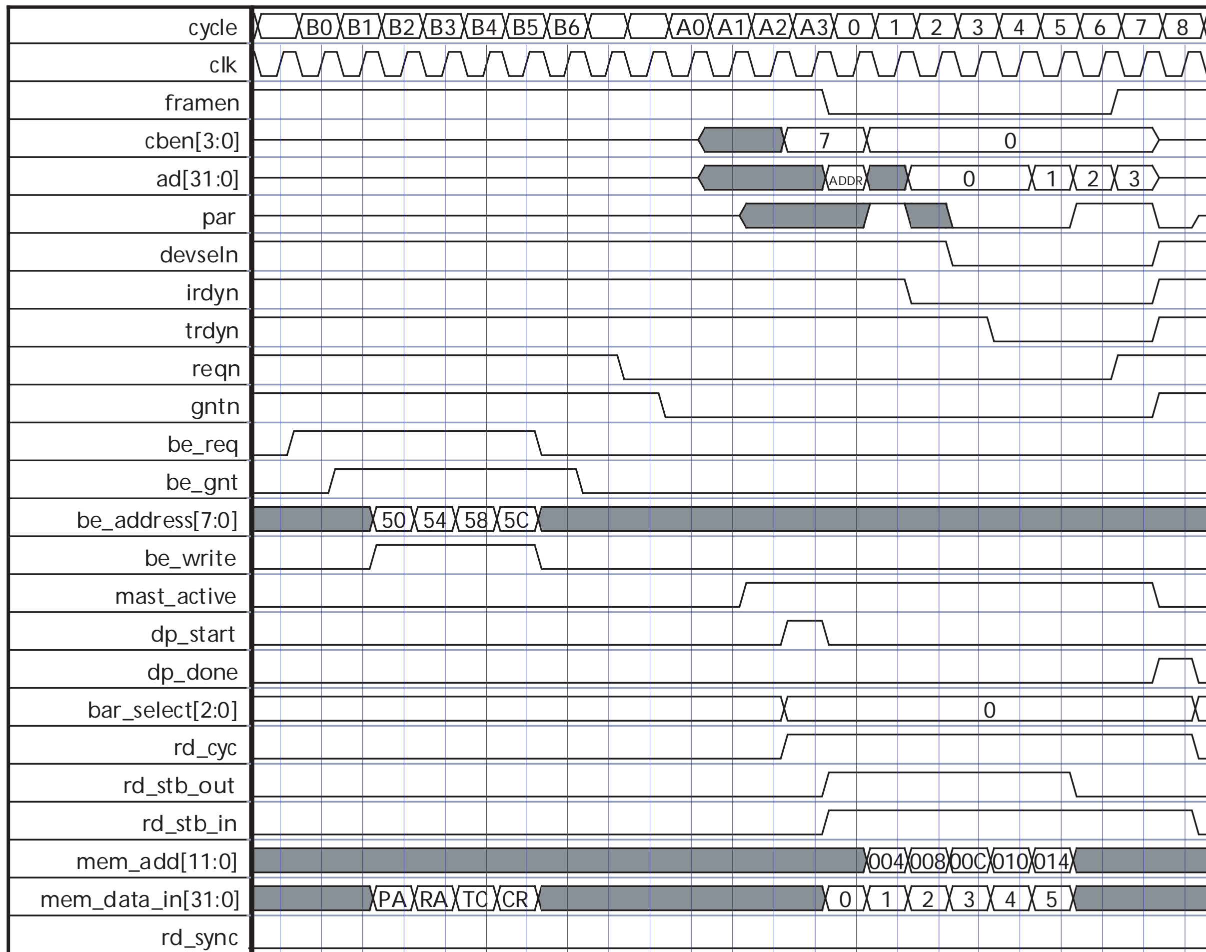


Figure 63 shows a DMA startup sequence from the backend interface. Initially, the backend requests access to the bus by asserting BE_REQ. When granted, it writes to the four DMA registers (cycles B2–B5), writing to the control register last. Three clock cycles after the write to the control register, the core asserts the PCI bus request and carries out a normal DMA transfer.

Downloaded from Arrow.com.

Figure 63 • **DMA Register Access and DMA Startup**

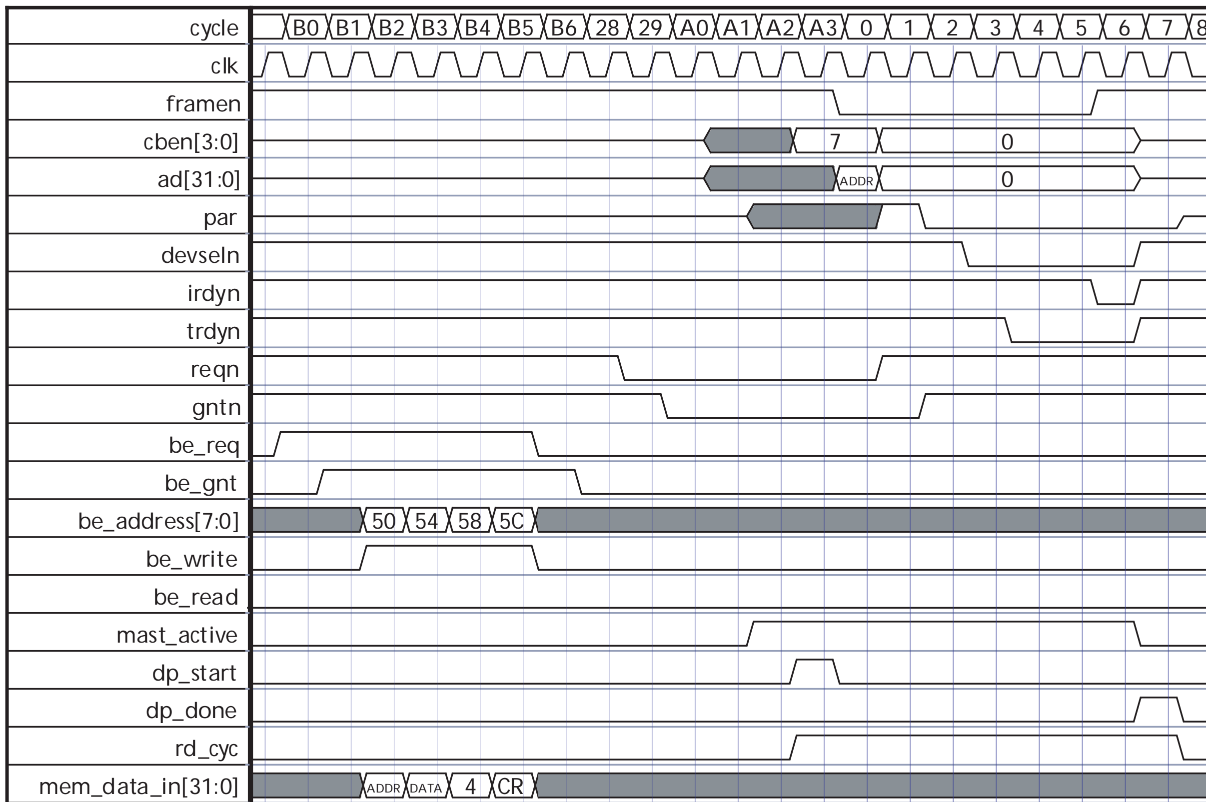


8.20 Direct DMA Transfers

CorePCIF supports direct DMA transfers. In this mode, the data used for the PCI transfer is read from or written to one of the internal DMA registers rather than the backend interface. Figure 64 and Figure 65 show a PCI write cycle and a PCI read cycle.

Downloaded from Arrow.com.

Figure 64 • **Direct DMA Write to the PCI Bus**

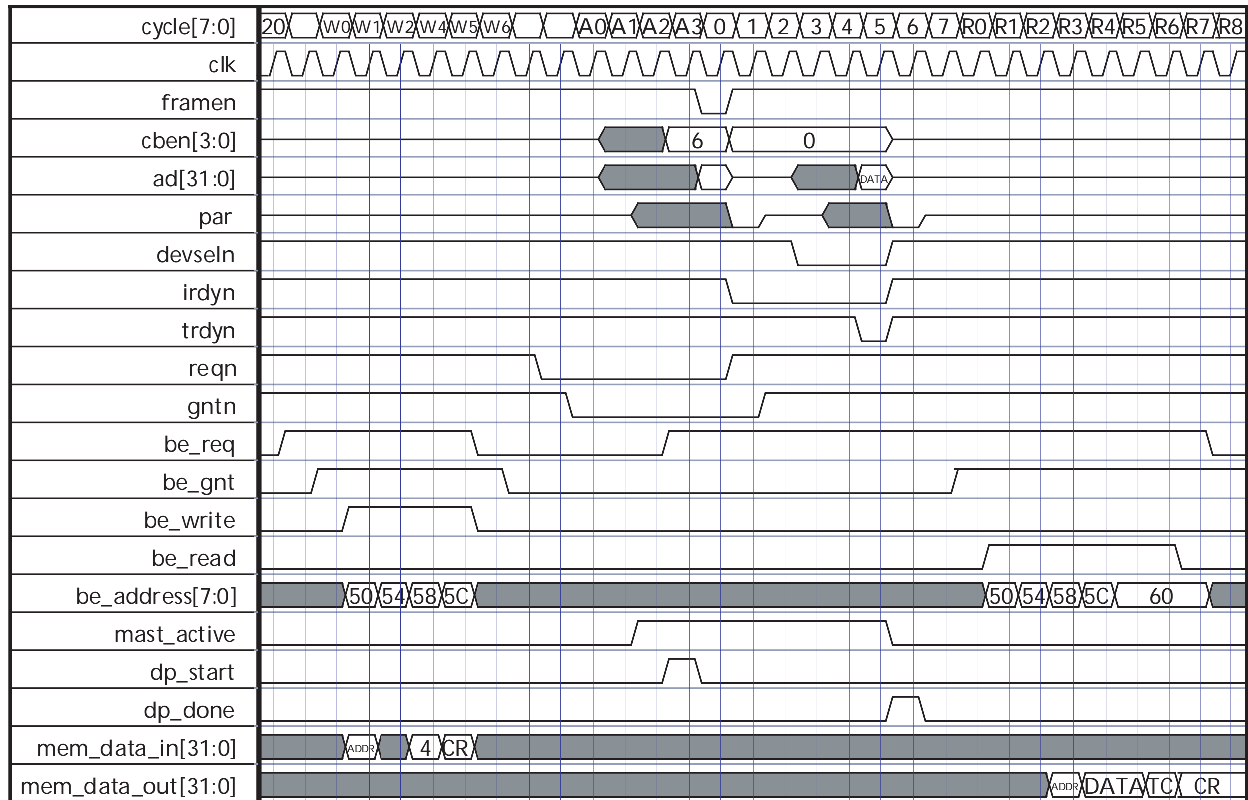


In Figure 65, the DMA registers are written during cycles B2 to B5. The PCI address is written during cycle B2, and the data values are written during cycle B3. Once the DMA control register is written (cycle B4), a DMA cycle is initiated, as normal. When DP_START is asserted, the BAR_SELECT output remains at '111', similar to a Target configuration cycle. The PCI cycle completes, with the data word being taken from the internal DMA control register.

Figure 65 shows the equivalent PCI read transfer. The data word is written into an internal DMA register. Also shown is a second backend cycle (R0–R7) that reads the data word from the internal registers once the DMA cycle completes.

Downloaded from Arrow.com.

Figure 65 • **Direct DMA Read from the PCI Bus**

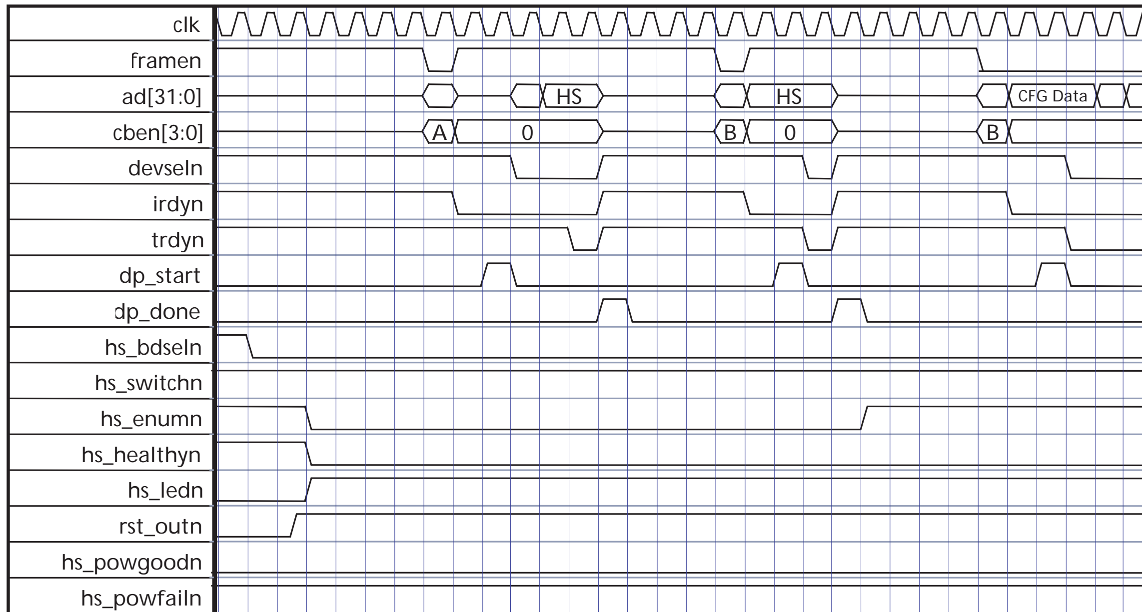


8.21 Hot-Swap Sequence

Figure 66 and Figure 67 show the switching on of the hot-swap interface signals during an insertion sequence and an extraction sequence.

Downloaded from Arrow.com.

Figure 66 • Hot-Swap Insertion Sequence

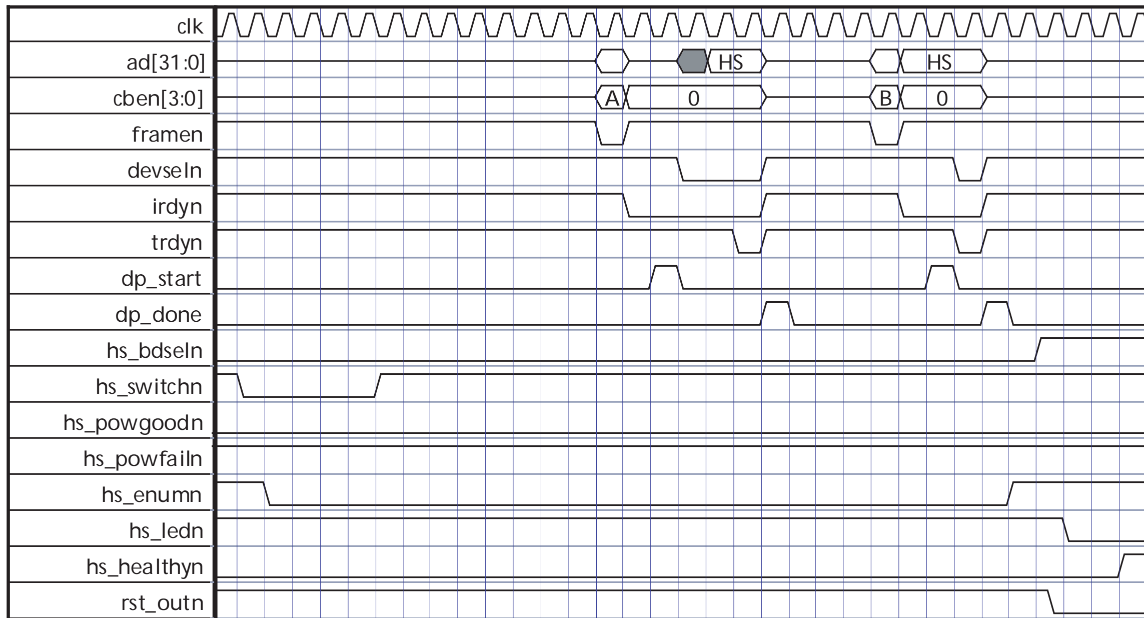


The HS_BDSELN signal becomes active at the start of the insertion sequence. This causes the core to exit the reset condition, asserting its HS_ENUMN and HS_HEALTHYN outputs. Sometime later, the PCI Master reads the hot-swap register to see why HS_ENUMN is active. In this case, bit 23 will be active, indicating an insertion. After this, the PCI Master clears the insertion status bit, causing HS_ENUMN to become inactive. The PCI Master then sets up the PCI configuration space.

The extraction process is triggered by the HS_SWITCHN input being asserted. This causes the HS_ENUMN output to be asserted. Sometime later, the PCI Master reads the hot-swap register to see why HS_ENUMN is active. In this case, bit 22 will be active, indicating an extraction request. The PCI Master clears the extraction status bit, causing HS_ENUMN to become inactive. As the board is removed, HS_BDSELN becomes active, causing the internal reset to be asserted.

Downloaded from Arrow.com.

Figure 67 • **Hot-Swap Extraction Sequence**



Downloaded from Arrow.com.

9 PCI Configuration Space

9.1 Target Configuration Space

The PCI specification requires a 256-byte configuration space (header) to define various attributes of the PCI Target, as shown in Table 29 and Table 30. All registers shown in bold are implemented. Reads of all other registers will return zero.

Table 29 • **PCI Configuration Space**

31–24	**23–16**	**15–8**	**7–0**	
Base PCI Configuration Space				**Address**
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base address #0				10h
Base address #1				14h
Base address #2				18h
Base address #3				1Ch
Base address #4				20h
Base address #5				24h
CardBus CIS Pointer (optional)				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM base address				30h
Reserved			Capability Pointer	34h
Reserved				38h
Max. Latency	Min. Grant	Interrupt Pin	Interrupt Line	3Ch

Table 30 • **PCI Configuration Space**

31–24	**23–16**	**15–8**	**7–0**	
Base PCI Configuration Space				**Address**
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
Base address #0 is used to access the buffer memory.				10h
Base address #1 is used to access the DMA control registers.				14h
Base address #2				18h
Base address #3				1Ch
Base address #4				20h
Base address #5				24h

Downloaded from Arrow.com.

Table 30 • PCI Configuration Space

31–24	**23–16**	**15–8**	**7–0**	
Base PCI Configuration Space				**Address**
CardBus CIS Pointer (optional)				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM base address				30h
Reserved			Capability Pointer	34h
Reserved				38h
Max. Latency	Min. Grant	Interrupt Pin	Interrupt Line	3Ch

CorePCIF uses the capability pointer to extend the configuration space. One or two capability structures are added. Vendor capability (ID = 9) is added along with optional hot-swap capability (ID = 6).

For a Target-only core, the capability pointer points to a vendor capability structure at address 44h (Table 31). The next pointer points to the optional hot-swap capability at address 40h. If hot-swap capability is not implemented, address 40h will be zero, and the next pointer, at address 44h, will also be zero.

For Master cores, the capability pointer points to a vendor capability structure at 4Ch (Table 32), followed by the optional hot-swap capability. If hot-swap capability is not implemented, address 40h will be zero, and the next pointer, at address 4Ch, will also be zero.

Table 31 • Capability Structure (Target-only cores with hot-swap)

31–24	23–16	15–8	7–0	Address
Upper Configuration Space				
Reserved	Hot-Swap	Next Pointer (00h)	Capability ID (06h)	40h
Microsemi Capabilities	Size (8)	Next Pointer (40h)	Capability ID (09h)	44h
Interrupt Control Register				48h
Reserved				4Ch
Reserved				50h
Reserved				54h
Reserved				58h
Reserved				5Ch

Table 32 • Capability Structure (Master cores with hot-swap)

31–24	23–16	15–8	7–0	Address
Upper Configuration Space				
Reserved	Hot-Swap	Next Pointer (00h)	Capability ID (06h)	40h
Reserved				44h
Reserved				48h
Microsemi Capabilities	Size (20)	Next Pointer (40h)	Capability ID (09h)	4Ch
PCI Address				50h

Downloaded from Arrow.com.

Table 32 • **Capability Structure (Master cores with hot-swap)**

31–24	23–16	15–8	7–0	Address
Upper Configuration Space				
Backend Address/Data Value				54h
Transfer Count				58h
DMA Control Register				5Ch

Table 33 • **Capability Structure (Target-only cores with hot-swap and FIFO status)**

31–24	23–16	15–8	7–0	Address
Upper Configuration Space				
Reserved	Hot-Swap	Next Pointer (00h)	Capability ID (06h)	40h
Microsemi Capabilities	Size (12)	Next Pointer (40h)	Capability ID (09h)	44h
Interrupt Control Register				48h
FIFO Status Register				4Ch
Reserved				50h
Reserved				54h
Reserved				58h
Reserved				5Ch

Table 34 • **Capability Structure (Master cores with hot-swap and FIFO status)**

31–24	23–16	15–8	7–0	Address
Upper Configuration Space				
Reserved	Hot-Swap	Next Pointer (00h)	Capability ID (06h)	40h
Reserved				44h
Microsemi Capabilities	Size (24)	Next Pointer (40h)	Capability ID (09h)	48h
FIFO Status Register				4Ch
PCI Address				50h
Backend Address/Data Value				54h
Transfer Count				58h
DMA Control Register				5Ch

9.1.1 Read-Only Configuration Registers

The following read-only registers, listed also in Table 29 and Table 30, have default values that are set by parameters. See the PCI specification for further information on setting these values:

- Vendor ID
- Device ID
- Revision ID

Downloaded from Arrow.com.

- Class Code
- Subsystem ID
- Subsystem Vendor ID
- Maximum Latency and Minimum Grant

Microsemi has an allocated Vendor ID that CorePCIF customers may use, and Microsemi will allocate a unique Device ID when the Microsemi Vendor ID is used. Microsemi will allocate unique subsystem Vendor IDs on request. Contact Microsemi Technical Support (tech@Microsemi.com) for more information.

The capability pointer is used to point to the CorePCIF vendor capability data, and also to the hot-swap capability, if enabled. The capability list structure varies, depending on the core configuration.

9.1.2 Read/Write Configuration Registers

The following registers have at least one bit that is both read- and write-capable. For a complete description, refer to the appropriate table.

- Command Register (04h) (Table 35)
- Status Register (06h) (Table 36)
- Memory Base Address Register Bit Definition (Table 37)
- I/O Base Address Register Bit Definition (Table 38)
- Interrupt Register (3Ch) (Table 39)
- Interrupt Control/Status Register (48h) (Table 40)
- Optional Hot-Swap Register (80h) (Table 41)

Table 35 • **Command Register 04 Hex**

Bit(s)	Type	Description
0	RW	I/O Space A value of 0 disables the device's response to I/O space addresses. Set to 0 after reset.
1	RW	Memory Space A value of 0 disables the device's response to memory space addresses. Set to 0 after reset.
2	RW	Bus Master When set to 1, this bit enables the macro to behave as a PCI bus Master. For Target-only implementation, this bit is read-only and is set to 0.
3	RO	Special Cycles Response to special cycles is not supported in the core. Set to 0.
4	RO	Memory Write and Invalidate Enable Memory Write and Invalidate Enable is not supported by the core. Set to 0.
5	RO	VGA Palette Snoop Assumes a non-VGA peripheral. Set to 0.
6	RW	Parity Error Response When 0, the device ignores parity errors. When 1, normal parity checking is performed. Set to 0 after reset.
7	RO	Wait Cycle Control No data-stepping supported. Set to 0.
8	RW	SERRN Enable When 0, the SERRN driver is disabled. Set to 0 after reset.
9	RO	Set to 0. Only fast back-to-back transactions to the same agent are allowed.
10	RW	Interrupt Disable When set, this prevents the core from asserting its INTAn output. This bit is set to 0 after reset.
15:11	RO	Reserved. Set to '00000'.

Downloaded from Arrow.com.

Table 36 • Status Register 06 Hex

Bit(s)	Type	Description
2:0	RO	Reserved. Set to '000'.
3	RO	Interrupt Status This bit reflects the status of the INTAn output.
4	RO	Capabilities List This is set to 1. CorePCIF implements a vendor capability ID and optional hot-swap capability.
5	RO	66 MHz Capable Set to 1 to indicate a 66 MHz Target, or 0 to indicate a 33 MHz Target. The value is set by the MHZ_66 parameter.
6	RO	UDF Supported Set to 0 (no user definable features).
7	RO	Fast Back-to-Back Capable Set to 0 (fast back-to-back to same agent only).
8	RW	Data Parity Error Detected If the Master controller detects a PERRn, this bit is set to 1. This bit is read-only in Target-only implementations and is set to 0. It is cleared by writing a '1'.
10:9	RO	DEVSELn timing Set to '10' (slow DEVSELn response).
11	RW	Signaled Target Abort Set to 0 at system reset. This bit is set to 1 by internal logic whenever a Target abort cycle is executed. It is cleared by writing a '1'.
12	RW	Received Target Abort If the Master controller detects a Target Abort, this bit is set to 1. This bit is read-only in Target-only implementations and is set to 0. It is cleared by writing a '1'.
13	RW	Received Master Abort If the Master controller performs a Master Abort, this bit is set to 1. This bit is read-only in Target-only implementations and is set to 0. It is cleared by writing a '1'.
14	RW	Signaled System Error Set to 0 at system reset. This bit is set to 1 by internal logic whenever the Target asserts the SERRn signal. It is cleared by writing a '1'.
15	RW	Detected Parity Error Set to 0 at system reset. This bit is set to 1 by internal logic whenever a parity error, address, or data is detected, regardless of the value of bit 6 in the command register. It is cleared by writing a '1'.

Table 37 • Base Address Registers (memory) 10 Hex to 24 Hex

Bit(s)	Type	Description
0	RO	Memory Space Indicator. Set to 0.
2:1	RO	Set to '00' to indicate anywhere in 32-bit address space.
3	RO	Prefetchable. Set by the BAR*i*_PREFETCH parameter.
31:4	RW/RO	Base Address. Depending on the BAR*i*_ADDR_WIDTH parameter, these bits may be writable or read-only. If a 128 kB address space is set (BAR*i*_ADDR_WIDTH = 17), bits 31:17 will be readable/writable, and bits 16:4 will be read-only and set to 0Sets the PCI base address of buffer memory (BAR0) or the DMA registers (BAR 1).

Downloaded from Arrow.com.

The base address registers (I/O), 10 hex to 24 hex, are automatically set based on the core configuration.

Table 38 • Base Address Registers (I/O) 10 Hex to 24 Hex

Bit(s)	Type	Description
0	RO	I/O Space Indicator. Set to 1.
1	RO	Reserved. Set to 0.
31:2	RW	Base Address. Depending on the BAR*i*_ADDR_WIDTH parameter, these bits may be writable or read-only. If a 256-byte address space is set (BAR*i*_ADDR_WIDTH = 8), bits 31:24 will be readable/writable, and bits 7:2 will be read-only and set to 0.

Table 39 • Expansion ROM Address Register 30 Hex

Bit(s)	Type	Description
0	RO	Expansion ROM enable bit. Set by the EXPR_ENABLE parameter Expansion ROM is not implemented, so bit 0 is hard coded to 0.
3110:1	RO	Reserved. Set to 0.
31:11	RW	Base Address. Depending on the EXPR_ADDR_WIDTH parameter, these bits may be writable or read-only. If a 64 kB address space is set (EXPR_ADDR_WIDTH = 16), bits 31:16 will be readable/writable, and bits 15:11 will be read-only and set to 0.

Table 40 • Capabilities Pointer 34 Hex

Bit(s)	Type	Description
7:0	R	For Target-only cores, this will be set to 44 hex. If a Master function is implemented, it will be set to 48 hex or 4C hex.
31:8	RO	Reserved. Set to 0.

Table 41 • Interrupt Register 3C Hex

Bit(s)	Type	Description
7:0	RW	Required read/write register. This register has no impact on internal logic.
31:8	RO	Set to 01 hex to indicate INTAn.

Table 42 • Hot-Swap Capability Register 40 Hex

Bit(s)	Type	Description
7:0	RO	Hot-swap capability. Set to 06 hex.
15:8	RO	Next Capability Pointer. Set to 00 hex.
16	RO	Device Hiding Arm (DHA) Since the core only supports programming interface 0, this is '0'.
17	RW	ENUM# Signal Mask (EIM) When 1, the HS_ENUMn output is held inactive.
18	RO	Pending Insert or Extract (PIE) Set when bit 22 or 23 is set.
19	RW	LED On/Off (LOO)
21:20	RO	Programming interface (PI) Fixed at '00', programming interface 0 supports INS, EXT, LOO, and EIM.

Downloaded from Arrow.com.

Table 42 • **Hot-Swap Capability Register 40 Hex**

Bit(s)	Type	Description
22	RW	ENUM# Status – Extraction (EXT) When set to 1, indicates that the board is about to be extracted. Writing a '1' clears this bit.
23	RW	ENUM# Status – Insertion (INS) When set to 1, indicates that the board has just been inserted. Writing a '1' clears this bit.
31:24	RO	Reserved. Set to 0.

Table 43 • **Microsemi Capabilities Register 44, 48, or 4C Hex**

Bit(s)	Type	Description
7:0	RO	Microsemi vendor capability. Set to 09 hex.
15:8	RO	Next Capability Pointer. Set to 40 hex.
23:16	RO	Capability Size. Set to 8, 12, 20, or 24, depending on core configuration.
25:24	RO	DMA_REG_LOG Indicates the DMA register location. 0: DMA registers are not implemented. 1: DMA registers are only mapped in the PCI configuration space. 2: DMA registers are mapped to memory locations 50–5F hex of the BAR, indicated by bits 28:26. 3: DMA registers are mapped to I/O locations 50–5F hex of the BAR, indicated by bits 28:26. These two bits are set by the DMA_REG_LOC parameter.
28:26	RO	Indicates which BAR is used to access the DMA registers if mapped to memory or I/O space. These three bits are set by the DMA_REG_BAR parameter.
29	RO	Indicates that the backend interface is enabled and has access to the DMA registers. This bit is set by the BACKEND parameter.
30	RO	When set, indicates that the BAR overflow logic in the core is disabled. Burst accesses will simply wrap within the BAR. This bit is set by the DISABLE_BAROV parameter.
31	RO	When set, indicates that the watchdog timer in the core is disabled. The core may insert more than the allowed number of wait cycles during a transfer. This bit is set by the DISABLE_WDOG parameter.

Table 44 • **Interrupt Control Register 48 Hex (MASTER = 0)**

Bit(s)	Type	Description
9:0	RO	Reserved. Set to 0.
10	W	Flush Internal FIFOs Only has an effect when the FIFO recovery logic is enabled. When written with a '1', all the internal FIFOs will be flushed. When the FIFOs are flushed, any data that was stored in them will be lost. Always returns 0 when read.
13:11	RO	Reserved. Set to 0.
14	RW	External Interrupt Status A '1' in this bit indicates an active external interrupt condition (assertion of EXT_INTn). It is cleared by writing a '1' to this bit. It is set to 0 after reset.
15	RW	External Interrupt Enable Writing a '1' to this bit enables support for the external interrupt signal. Writing a '0' to this bit disables external interrupt support.
31:16	RO	Reserved. Set to 0.

Downloaded from Arrow.com.

Table 45 • **FIFO Status Register**

Bit(s)	Type	Description
2:0	RO	Number of words queued inside the core for BAR 0
3	RO	External FIFO status for BAR 0; 0 = empty, 1 = non-empty
6:4	RO	Number of words queued inside the core for BAR 1
7	RO	External FIFO status for BAR 1; 0 = empty, 1 = non-empty
10:8	RO	Number of words queued inside the core for BAR 2
11	RO	External FIFO status for BAR 2; 0 = empty, 1 = non-empty
14:12	RO	Number of words queued inside the core for BAR 3
15	RO	External FIFO status for BAR 3; 0 = empty, 1 = non-empty
18:16	RO	Number of words queued inside the core for BAR 4
19	RO	External FIFO status for BAR 4; 0 = empty, 1 = non-empty
22:20	RO	Number of words queued inside the core for BAR 5
23	RO	External FIFO status for BAR 5; 0 = empty, 1 = non-empty
31:24	RO	Reserved. Set to 0.

Table 46 • **PCI Address Register 50 Hex**

Bit(s)	Type	Description
1:0	RW	These two bits set the lowest two bits of the PCI address. For normal DWORD-aligned transfers, these two bits should be '00'. They may be set to non-zero values to alter the requested burst order for memory accesses or to specify a byte address for I/O accesses.
31:2	RW	This location contains the PCI start address and will increment during the DMA transfer. If using 64-bit transfers, bit 2 should also be set to 0.

Table 47 • **Backend Address Register 54 Hex (ENABLE_DIRECTDMA = 0)**

Bit(s)	Type	Description
1:0	RO	Set to '00'. PCI transfers must be on DWORD boundaries.
31:2	RW/ RO	This location contains the backend start address and will increment during the DMA transfer. The width of this register depends on the MADDR_WIDTH parameter. If MADDR_WIDTH is set to 20, bits 31:20 of this register are read-only and set to 0. If using 64-bit transfers, bit 2 should be set to 0.

Table 48 • **Backend Address and Data Register 54 Hex (ENABLE_DIRECTDMA = 1)**

Bit(s)	Type	Description
31:0	RW	When DMA_BAR = '111' (<Blue>Table 50 on page 94), this register contains the 32-bit data value that will be written to or read from the PCI bus. When DMA_BAR ≠ '111', this specifies the backend address. The core will ignore bits 1 and 0 to align the transfer count to a DWORD boundary. If using 64-bit transfers, bit 2 should also be set to 0.

Downloaded from Arrow.com.

Table 49 • **DMA Transfer Count 58 Hex**

Bit(s)	Type	Description
31:0	RW	Specifies the size of a DMA transfer in bytes. For 32-bit operation, this should be a multiple of four, and for 64-bit operations, a multiple of eight. Bits 1:0 are read-only and return 0. The maximum transfer size is set by the DMA_COUNT_WIDTHMEMORY_SIZE parameter. When set to zero, $2^{\text{DMA_COUNT_WIDTHMEMORY_SIZE}}$ bytes will be transferred.

Table 50 • **DMA Control Register 5C Hex**

Bit(s)	Type	Description
1:0	RW	DMA Status 00: No Error 01: Master Abort 10: Parity Error 11: Target Abort
2	RW	DMA Done A '1' indicates that the DMA transfer is complete. Writing a '0' clears this bit.
3	RW	DMA Request Writing a '1' will initiate a DMA transfer, and the bit will remain set until the DMA transfer completes or an error occurs (Master abort or Target abort). This bit can only be set if the bus Master enable bit is set in the PCI Command register (<Blue>Table 35 on page 89).
7:4	RW	Cycle Type Sets the DMA transfer type and direction. These four bits directly set the PCI transfer type. Any of the sixteen PCI commands may be used, but the recommended commands are as follows: 0010 Data is moved from the PCI bus to the backend. An I/O Read command is used on the PCI bus. 0011 Data is moved from the backend to the PCI bus. An I/O Write command is used on the PCI bus. 0110 Data is moved from the PCI bus to the backend. A Memory Read command is used on the PCI bus. 0111 Data is moved from the backend to the PCI bus. A Memory Write command is used on the PCI bus. 1010 Data is moved from the PCI bus to the backend. A Configuration Read command is used on the PCI bus. 1011 Data is moved from the backend to the PCI bus. A Configuration Write command is used on the PCI bus. 1100 Data is moved from the PCI bus to the backend. A Memory Read Multiple command is used on the PCI bus.
8	RW	DMA Enable This bit must be set to 1 to enable any DMA transfers.
9	RW	Transfer Width Writing a '1' to this bit enables a 64-bit memory transaction. For 32-bit cores, this bit is read-only and is set to 0Reserved. Returns 0.
10	W	Flush Internal FIFOs Only has an effect when the FIFO recovery logic is enabled. When written with a '1', all internal FIFOs will be flushed. When the FIFOs are flushed, any data that was stored in them will be lost. Always returns 0 when readReserved. Returns 0.
11	RO	Reserved. Returns 0.

Downloaded from Arrow.com.

Table 50 • **DMA Control Register 5C Hex *(continued)***

Bit(s)	Type	Description
12	RW	DMA Interrupt Status A '1' in this bit indicates that the DMA cycle has completed and the interrupt is active. It is cleared by writing a '1' to this bit. Set to 0 after reset.
13	RW	DMA Interrupt Enable Writing a '1' to this bit enables the DMA Complete interrupt. Set to 0 after reset.
14	RW	Backend Interrupt Status A '1' in this bit indicates an active backend interrupt condition (backend assertion of EXT_INTn). It is cleared by writing a '1' to this bit. Set to 0 after reset. This bit can only be set when the backend interrupt is enabled (bit 15).
15	RW	Backend Interrupt Enable Writing a '1' to this bit enables the backend interrupt. Writing a '0' to this bit disables backend interrupt support.
23:16	RW	Byte Enables These eight bits directly set the byte enable values that will be used during the DMA transfer. When bit 16 is 0, CBEN[0] will be active (LOW). Bit 17 controls CBEN[1], etc. In 32-bit cores, bits 23:20 are read-only and return 0. For normal burst DMA transfers, these bits should be set to 0.
25:24	RO	Reserved. Set to 0.
28:26	RW	DMA BAR Select Used to select which of the backend memory BARs the DMA will address. These bits are used to drive the DMA_BAR and BAR_SELECT outputs during the DMA transfer. When set to '000', BAR 0 will be selected. When set to '110', the Expansion ROM will be selected. When set to '111', a direct DMA access will be done. Data will be read from and written to the DMA data registers (54h), and no backend cycle will be carried out. When direct DMA mode is used, the transfer count register (58h) must be programmed to transfer one DWORD (0004 hex). The transfer width must be set to 32 bits.
31:29	RW	Maximum Burst Length When set to '000', the Master controller will attempt to complete the requested transfer in a single burst. When set to a non-zero value, the Master will automatically break up long bursts and limit burst transfer lengths to 2^{n-1}, where *n* is the decimal value of bits 31:29. Therefore, maximum transfer lengths can be limited to 1, 2, 4, 8, 16, 32, or 64 dataphases. For example, if the maximum burst length is set to '101' (16 transfers), a 1,024-DWORD transfer count would be broken up into 64 individual PCI accesses.

Note: During a DMA transfer (while bit 3, DMA Request, is set), the user should avoid writing to this register altogether, as this will interrupt the DMA transfer process. User's should poll for bit 3 cleared and then proceed with register manipulation.

Downloaded from Arrow.com.

10 Testbench Operation

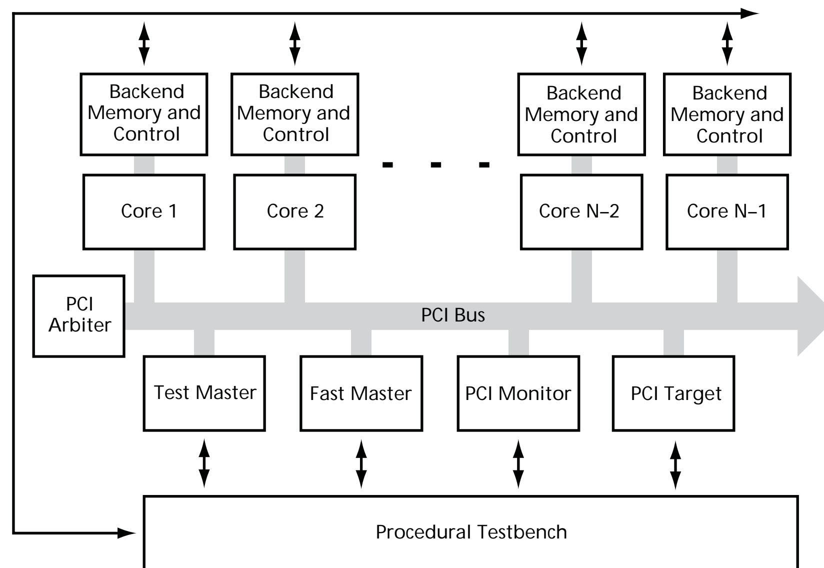
Three testbenches are provided with CorePCIF.

- VHDL verification testbench: Complex testbench that verifies core operation. This testbench exercises all the features of the core. Microsemi recommends not modifying this testbench. This VHDL testbench can be used to simulate the Verilog version of the core if a mixed-mode simulator is available.
- VHDL user testbench: Simple-to-use testbench written in VHDL. Additional optional tests are included, allowing verification of the AHB interface logic. This testbench is intended for customer modification.
- Verilog user testbench: Simple-to-use testbench written in Verilog. This testbench is intended for customer modification.

10.1 Verification Testbench

The verification testbench consists of a master test controller, a PCI monitor, an arbiter, and devices under test. The test master is used for generating configuration cycles and performing basic read-and-write tests of the macro when operating as a PCI Target. The PCI monitor checks for and flags abnormal PCI bus activity. The arbiter determines PCI bus ownership.

Figure 68 • **The Verification Testbench**



The verification testbench supports up to 14 cores connected to the PCI bus. Each core is configured differently, allowing multiple core configurations to be tested at the same time. Table 51 details the core configurations used in the standard testbench.

Table 51 • **Verification Testbench Configurations**

				BAR						
Core	**Width**	**Function**	**DMA**	**B0**	**B1**	**B2**	**B3**	**B4**	**B5**	**ROM**
1	32	TH	-		1KI	4KM	4KMF	512M	4KM	4K
2	32	TM	C4K	64KM	4KM					
3	32	TMBD	NB1G	1GM	64KI					
4	32	MBD	NB4K							

Downloaded from Arrow.com.

Table 51 • Verification Testbench Configurations

5	32	TMBD	M3B1G	1GM	1KM	1KM	256M		
6	32	TMBD	M2B16K	64KMF	64KMF	256M	64KMF		
7	32	TMS	CB4K	1KI		1KM	64I	128I	64I
8	64	T	–	64KM	1KI				
9	64	TMB	I2B	256KM	64KI	256I			
10	64	TMB	NB64K	4KMF	4KMF	4KMF	4KMF	4KMF	4KMF
11	64	MB	NB4K						
12	64	TM	M2N16K	256KM	1KM	256M			

The following coding is used within the table:

Function T = Target, M = Master, B = Backend, D = Direct DMA, S = Slow Read, H = Hot-Swap

DMA C = DMA registers in configuration space
M*n* = DMA registers in memory space using BAR *n*
I*n* = DMA registers in I/O space using BAR *n*
B = Backend access to DMA registers enabled
nn, *nn*K = Maximum DMA transfer count; 4K = 4096 bytes
N = No access; replaces C, M, I, or B
Bars*nn*, *nn*K, *nn*M = BAR size
M = Memory space
I = I/O space

The procedural testbench controls testbench operation through the test master and fast master blocks. When the testbench starts, the procedural testbench scans the PCI bus, discovering which PCI devices are connected to the bus. As shipped, it will discover 12 PCI cores with configurations as described in Table 51. It will then allocate memory space and configure all the cores.

Once all the cores are configured, the procedural testbench will prompt for which test to run. The available tests are listed below. Entering **99** will run all tests and exit the simulation. Running all tests may take more than 10 hours, depending on your computer configuration. The tests are fully detailed in Verification Testbench Tests.

```
# CorePCI Verification Testbench Commands
#
# ENTER =>  01 To Run Simple Read/Write Test
# ENTER =>  02 To Run All BAR's Read/Write Test
# ENTER =>  03 To Run Byte Enable Test
# ENTER =>  04 To Run DEVSEL Timing Test
# ENTER =>  05 To Run Address Parity Error Test
# ENTER =>  07 To Run Interrupt Test
# ENTER =>  08 To Run Data Parity Error Test
# ENTER =>  10 To Run Two Target Test
# ENTER =>  11 To Run Retry Test
# ENTER =>  13 To Run Target Abort Test
# ENTER =>  14 To Run Back-to-Back Test
# ENTER =>  15 To Run Bar Overflow Test
# ENTER =>  16 To Run Memory Read Line, Memory Read Multiple and Memory Write 
Invalidate
               Test
# ENTER =>  17 To Run Unaligned Address Transfer Test
# ENTER =>  18 To Run Target Dataflow test
# ENTER =>  19 To Run FIFO Interface test
# ENTER =>  20 To Run BAR Select test
# ENTER =>  21 To Run Read Byte Handshake tests
# ENTER =>  22 To Run Hot Swap Interface Tests
```


Downloaded from Arrow.com.

```
# ENTER =>  23 To Run Configuration Cycle Tests
# ENTER =>  24 To Run Retry/Disconnect Time Tests
# ENTER =>  25 To Run Multiple FIFOs tests
# ENTER =>  26 To Run User target tests
# ENTER =>  27 To Run FIFO Status tests
# ENTER =>  40 To Run Simple DMA Transfer Test
# ENTER =>  41 To Run Back-end Control During Cycle Test
# ENTER =>  42 To Run DMA Single Transfer Test (with and W/O Wait States)
# ENTER =>  43 To Run DMA Poll Status Test
# ENTER =>  44 To Run DMA Counts Test
# ENTER =>  45 To Run DMA Mega Test
# ENTER =>  60 To Run MASTER Mode Test
# ENTER =>  47 To Run DMA Single Transfer Test with DMA completion interrupt
enabled
# ENTER =>  48 To Run DMA Poll Status Test with DMA completion interrupt enabled
# ENTER =>  49 To Run DMA Single Transfer Test Busy_Master test
# ENTER =>  50 To Run DMA Single Transfer Test Stall_Master test
# ENTER =>  51 To Run Byte Enable Test on DMA registers
# ENTER =>  52 To Run Byte Enable Test on Back End registers
# ENTER =>  53 To Run DMA With Max Transfer Length
# ENTER =>  54 To Run DMA dataflow using FIFOIF
# ENTER =>  55 To Run DMA Burst length tests using FIFOIF
# ENTER =>  56 To Run DMA Auto transfer test using FIFOIF
# ENTER =>  57 To Run Fast Master Back-to-Back Test
# ENTER =>  58 To Run Direct Mode DMA Transfer Test
# ENTER =>  59 To Run Miscellaneous DMA Tests
# ENTER =>  60 To Run Backend Config Space Tests
# ENTER =>  70 To Run MASTER Mode Test
# ENTER =>  71 To Run User DMA  tests
# ENTER =>  98 To Quick Test (All Slots)
# ENTER =>  99 To Run Exhaustive Tests (All Tests/All Slots)
# ENTER =>  S  To Run test 00-99 on slot S ie 112
# ENTER =>  Q  TO EXIT Testbench
```

10.1.1 Customizing the Verification Testbench

The number of core instances in the verification testbench and the configuration of each core can be modified by editing the *coreconfig.vhd* file. It is recommended that customers using the OEM version of Model*Sim* supplied with Libero modify the NSLOTS constant at the top of the *coreconfig.vhd* file to reduce the number of active cores to three. This will decrease simulation time, but test coverage is reduced to 32-bit cores only.

The *usertests.vhd* file contains two example routines that perform Target transactions and Master transfers. These routines can be used as starting points for adding additional tests if required. However, due to the complexity of the verification testbench, Microsemi recommends that it not be modified, and that the simple user testbenches described in the following sections be used as starting points for any user simulations. The verification testbench is provided to demonstrate the core’s operation under multiple conditions.

10.1.2 Files Used in the Verification Testbench

Table 52 lists all the VHDL source files used in the verification testbench and gives a description of their functions. All source files are provided with the RTL release. With the Evaluation release, only some of the source files are provided. All others are pre-compiled into the CorePCIF simulation library.

Table 52 • **Verification Testbench Source Files**

File	Supplied	Function
tb_verif.vhd	Yes	Top level of testbench. Creates a PCI bus and instantiates all the devices connected to the bus.

Downloaded from Arrow.com.

Table 52 • **Verification Testbench Source Files *(continued)***

File	Supplied	Function
coreconfig.vhd	Yes	VHDL package that is used to configure the number of cores and the parameter settings for each of the cores. By default, 12 cores are configured, allowing multiple core implementations to be tested at the same time.
pci_monitor.vhd	RTL only	PCI bus monitor that monitors PCI activity, looking for illegal activity. Also capable of tracing and displaying PCI activity.
pci_target.vhd	RTL only	PCI Target used to generate error conditions when testing the DMA function.
pci_arbiter.vhd	RTL only	PCI arbiter that supports up to 16 Masters used in the testbench.
test_master.vhd	RTL only	PCI Master function used by the testbench to carry out PCI cycles. Also contains the main procedural testbench and user command entry code. It calls the tests provided in the *tests.vhd* package.
fast_master.vhd	RTL only	Second PCI Master function used by the testbench to carry out PCI cycles. This Master is capable of performing PCI transactions at a very fast rate.
tests.vhd	RTL only	VHDL package that contains all the procedures used for performing the tests
usertests.vhd	RTL only	VHDL package that contains some basic test routines that can be used as templates for adding additional tests if required
tb_package.vhd	RTL only	VHDL package that defines all the types and low-level function calls used in the testbench
backend.vhd	RTL only	This backend interface logic implements each of the required backend memory blocks using *bendmem.vhd*, and provides control logic to access the backend interface. It also allows the testbench to control and monitor the backend interface.
bendmem.vhd	RTL only	Implements the actual backend memory block for each configured BAR. It can be configured to operate as a FIFO or as memory.
waveforms.vhd	RTL only	VHDL package that contains all the procedures used for generating the waveforms shown in this handbook
waveform.vhd	RTL only	Monitors the PCI bus and retimes the signals for output to a VCD file for generation of the waveforms shown in this handbook.
tb_components.vhd	RTL only	VHDL package that declares the components used in the testbench
textio.vhd	RTL only	VHDL package that provides the printf function used in the testbench
misc.vhd	RTL only	VHDL package that provides some very low-level type definitions and functions

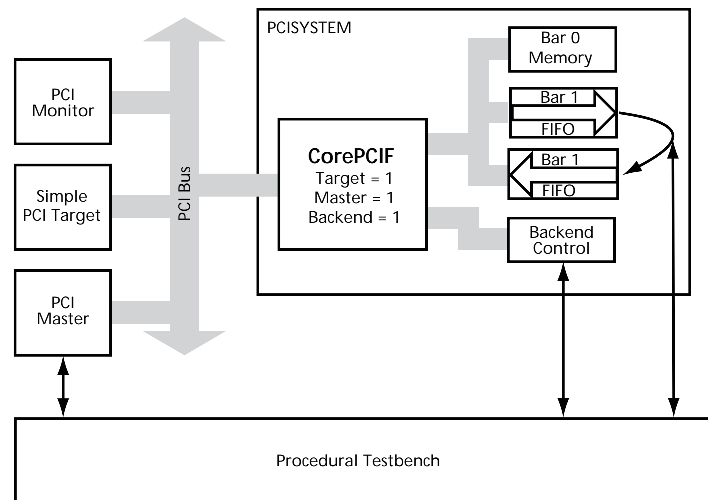
10.2 User Testbench

The user testbenches are intended to act as a starting point for creating a simulation environment for the end-user circuit, and are provided in both VHDL and Verilog. The testbench structure and tests carried out are identical for the VHDL and Verilog testbenches.

The testbench structure is shown in Figure 69. It instantiates a single core that is connected to the PCI bus. The core is instantiated in the PCISYSTEM module. This adds backend memory and FIFOs to the core to create a simple PCI system.

Downloaded from Arrow.com.

Figure 69 • User Testbench



Also attached to the PCI bus are a PCI monitor that displays the PCI bus activity and a simple PCI Target model that is used as a Target when CorePCIF is carrying out DMA activity. The PCI Master module is used by the procedural testbench to carry out PCI cycles. The procedural testbench can also access the CorePCIF backend interface to program the DMA registers.The procedural testbench can also initiate AHB cycles to access the AHB Slave interface within CorePCIF.

The PCISYSTEM module creates a simple PCI system that contains a memory BAR and a second BAR connected to input and output FIFOs. Data from the output FIFO is moved to the input FIFO at a variable rate controlled by the procedural testbench.

The PCISYSTEM design can be synthesized when the Axcelerator, IGLOO/e, ProASIC3/E, or Fusion FPGA family is selected, creating a single-chip PCI system. To synthesize the design, move the *pcisystem*, *fifo*, *memory*, *fifo512x32*, and *ram2k8* files from the CorePCI stimulus directory to the HDL source files directory in Libero IDE. Also supplied is an even simpler design, PCISYSTEM2, that implements just a memory BAR connected to the PCI core. This can be synthesized by copying the *pcisystem2* file as well.

10.2.1 Files Used in the User Testbenches

Table 53 lists all the VHDL and Verilog source files used in the user testbenches and gives a description of their functions. All source files are provided with the RTL release. With the Evaluation release, only some of the source files are provided. All others are pre-compiled into the CorePCIF simulation library.

Table 53 • User Testbench Source Files

File	Supplied	Function
tb_user.vhd tb_user.v	Yes	Top level of testbench. Creates a PCI bus and instantiates all the devices connected to the bus. It also contains the procedural testbench.
pcisystem.vhd pcisystem.v	Yes	Top level of the test design that includes the cores and memory blocks. This is a synthesizable design in some families.
memory.vhd memory.v	Yes	Top-level memory module creating the 8 k words of memory (or 16 k for 64-bit cores) used for BAR 0
fifo.vhd fifo.v	Yes	Top-level FIFO module creating the FIFOs used for BAR 1
ram2k8.vhd ram2k8.v	Yes	Low-level memory block implementing the memory using FPGA memory blocks
fifo512x32.vhd fifo512x32.v	Yes	Low-level FIFO block implementing the FIFO using FPGA FIFO blocks

Downloaded from Arrow.com.

Table 53 • **User Testbench Source Files**

File	Supplied	Function
tb_amba.vhd tb_amba.v	Yes	Top level of testbench. Creates a PCI bus and instantiates all the devices connected to the bus. It also contains the procedural testbench.
coreparameters.vhd coreparameters.v	Yes	Package or include file used to configure the core instantiated in the PCISYSTEM module. The testbench file uses this to decide which tests to run. This file is auto-generated by Libero during the core generation, and the settings will match those set in the Configuration GUI.
pcimaster.vhd pcimaster.v	RTL and Obfuscated only	PCI Master function used by the testbench to carry out PCI cycles
pcitarget.vhd pcitarget.v	RTL and Obfuscated only	Simple PCI Target function that implements a PCI Target capable of responding to memory read and write cycles
pcimonitor.vhd pcimonitor.v	RTL and Obfuscated only	PCI bus monitor that monitors PCI activity, looking for illegal activity. Also capable of tracing and displaying PCI activity.
textio.vhd	RTL only	VHDL package that provides the printf function used in the testbench. Not required for the Verilog version.
misc.vhd	RTL only	VHDL package that provides some very low-level type definitions and functions. Not required for the Verilog version.

10.2.2 Testbench Operation

When the testbench starts, it initially reads the vendor and device IDs from the core and verifies that they are defined by the constants in the *coreconfig* file. It then sets up the PCI configuration space. This sequence is shown in Figure 70.

Downloaded from Arrow.com.

Figure 70 • **User Testbench Startup Sequence**

```
# PCI User Testbench - Microsemi
Group # CorePCIF 4.1 Release July 2017
#
#Basic Core Configuration from coreconfig.vhd
# TARGET              1
# MASTER              1
# BACKEND            1
# PCI_WIDTH          32
# DMA_REG_LOC    2
#
###########################################
# Reading Device & Vendor IDs
# PCI CONFIG Read Slot: 1 AD:00000000
# PCI32 Command CFGRD Started AD : 02000000
# PCI32 CFGRD ADDR : 02000000 DATA : 600011AA BYTES
: 1111
# Device Vendor ID 600011AA
#PCI CONFIG Read Slot: 1 AD:0000002C
# PCI32 Command CFGRD Started AD : 0200002C
# PCI32 CFGRD ADDR : 0200002C DATA : 600011AA
BYTES : 1111
# Subsystem Device Vendor ID 600011AA
###########################################
#Programming Configuration Space
```

While these transfers are being carried out, the PCI monitor function logs all PCI bus transactions.

Once configured, the testbench will perform the sequence of tests in Table 54. If the core configuration, as set in Configuration window, does not support the required function, the test will not be performed.

Table 54 • **User Testbench Test Sequence**

Test	Required Core Parameters	Description
0	Target = 1	PCI device and vendor IDs are verified and the configuration space initialized.
1	Target = 1 Bar0_ENABLE = 1	Single-cycle Target write and read cycle to BAR 0.
2	PCI_Target = 1 Bar0_ENABLE = 1	Burst Target write and read cycle to BAR 0.
3	TARGET = 1 MASTER = 1 BAR0_ENABLE = 1	DMA transfer test initially from BAR 0 to the PCI bus (the simple Target). The DMA access is initiated by the testbench using the PCI Master to write to the DMA registers. A second DMA transfer is then performed to move the data back from the PCI bus to a different location in BAR 0 . Finally, the resultant data is verified.
4	TARGET = 1 MASTER = 1 BACKEND = 1 BAR0_ENABLE = 1	DMA transfer test initially from BAR 0 to the PCI bus (the simple Target). The DMA access is initiated by the testbench writing to the DMA registers using the backend interface. A second DMA transfer is then performed to move the data back from the PCI bus to a different location in BAR 0. Finally, the resultant data is verified.

Downloaded from Arrow.com.

Table 54 • **User Testbench Test Sequence**

Test	Required Core Parameters	Description
5	TARGET = 1 MASTER = 1 BACKEND = 1 BAR1_ENABLE = 2	FIFO test Initially, the testbench, using the PCI Master, fills up the output FIFO by writing data to BAR 1. While data is being written, the clock used to move data from the output to the input FIFOs is disabled; all data remains in the output FIFO. Once all the data is loaded, the testbench (through the backend interface) programs the DMA engine to move all the data from BAR 1 to the PCI Target and re-enables the backend clock to move data between the two FIFOs. As data is moved into the input FIFO, CorePCIF automatically moves the data from the FIFO to the PCI Target using its DMA engine until the DMA transfer is completed. While this process is occurring, the rate at which data is moved between the two FIFOs varies, causing the input FIFO to empty and causing the PCI core to stop the DMA transfer until the FIFO is non-empty. When the DMA transfer is complete, the data in the PCI Target is verified.

Additional verification tests can be run by typing **runall.do** at the Model*Sim* prompt. This will invoke the simulation multiple times using different core configurations (not those set in Configuration window), and will also enable additional tests.

10.2.3 Customizing the User Testbenches

The user testbenches are intended to be customized by the user. First, the PCISYSTEM module should be replaced by the actual PCI design being implemented. Once this is done, the test sequence in the main testbench file can be modified to perform the required configuration and memory cycles. When the simulation is run, the PCI monitor function will display the PCI activity, and the simple PCI Target can be used as a Target if the unit under test implements a PCI Master function.

VHDL User Testbench Procedures and **Verilog User Testbench Procedures** list all the procedure calls using the VHDL and Verilog testbenches. It is recommended that the *testbenchtb_amba.vhd (.v)* file be read carefully to fully understand testbench operation.

Downloaded from Arrow.com.

11 Implementation Hints

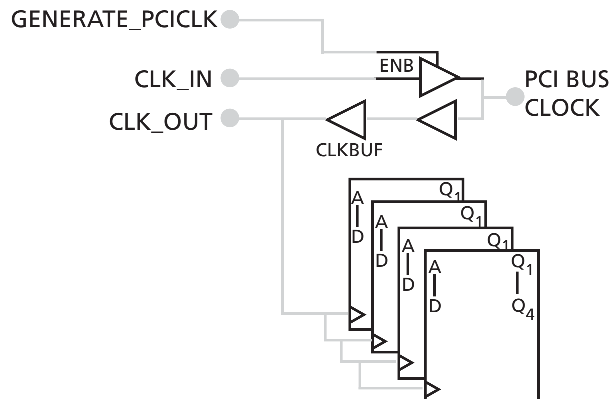
11.1 Clocking

CorePCIF supports generating the PCI clock when the FPGA is also the main bus control function. It is important that the clock networks be configured correctly to allow the core to meet the PCI setup and hold times, as well as to avoid internal clock skew.

If HCLK and the PCI clock are synchronous, Microsemi recommends using the CLK_OUT signal to directly drive the HCLK network, to minimize clock skew.

When generating the PCI clock, the clock source should be connected to the CLK_IN port. This is then routed to the PCI clock pad to drive the PCI bus, and driven back into the core using a global network. This global network on the CLK_OUT port should be used to clock the rest of the FPGA logic running off the PCI clock network (Figure 71).

Figure 71 • **Clock Generation**



11.1.1 Example: Clocking in SmartFusion2

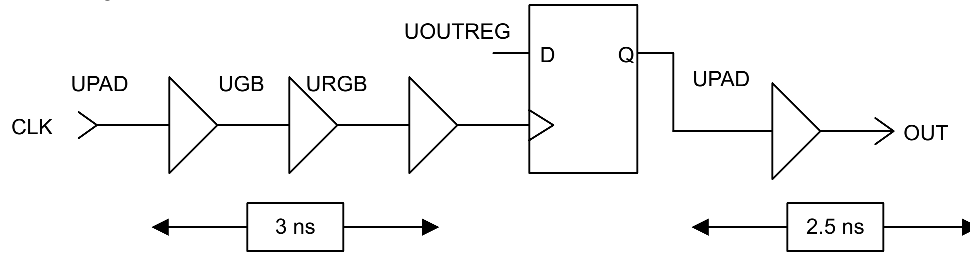
In the SmartFusion2 device, when using 66 MHz configuration, the CorePCIF PCI CLK input signal should be driven by a Fabric CCC (FCCC). You must instantiate the FCC in the top-level design and use the external CLK input as a reference clock. This is to ensure that external setup and clock to out timing are met in the design.

Note: The data paths (for example, AD) use registers to drive the output and you need to use IO-REG combining for these register outputs.

Figure 72 shows the clock-to-out path in SmartFusion2, which has a requirement of 6ns (as part of the PCIF standard. In the SmartFusion2 device, the sum of UPAD, UGB, and URGB on the clock input is around 3ns, if you use the IO bank as shown in **Pin Assignments**. The output pad delay is roughly 2.5ns. This leaves only around 500ps to route and register the output internally. This does not allow enough margin to achieve timing closure in most cases.

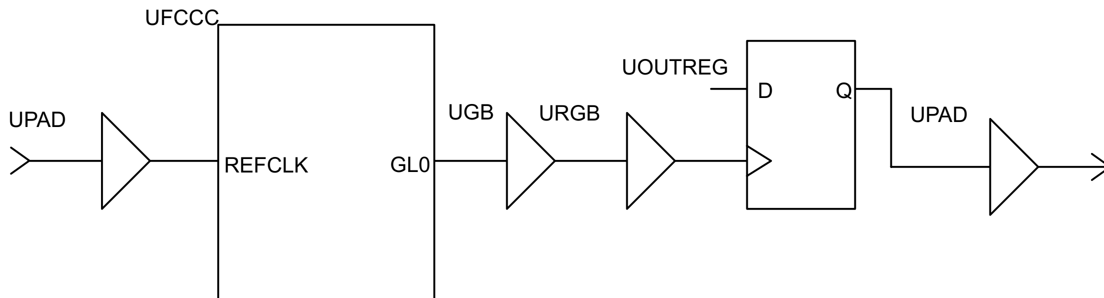
Downloaded from Arrow.com.

Figure 72 • **Clocking in SmartFusion2 with No CCC**



To increase the timing margin, it is advised to use an FCCC in the CLK data path to shift the clock backward, as shown in Figure 73.

Figure 73 • **Clocking in SmartFusion2 with CCC**

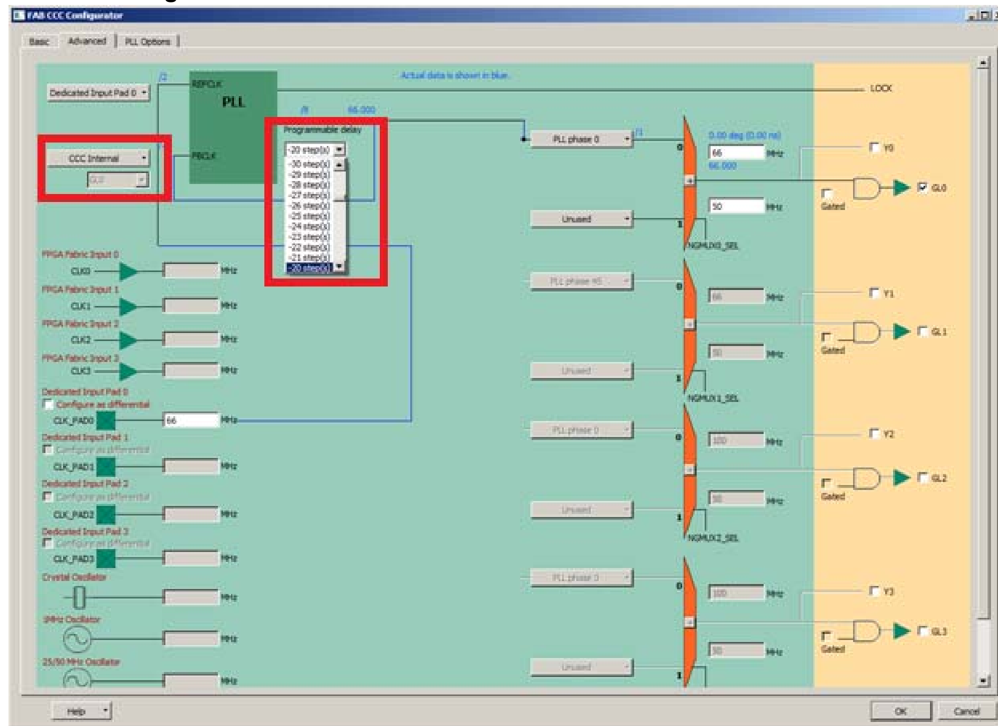


Using an FCCC, the clock can be shifted forward or backward. In this particular case, the clock should be shifted backwards, which will afford a large margin on clk-out path to the user. In the Libero SoC software, this is achieved using the programmable delay element in the FCCC output. Figure 74 shows how to configure the FCCC with a negative delay.

Note: The feedback must be set to CCC Internal. Otherwise, the reference clock and the output clock GL0 will remain in phase.

Downloaded from Arrow.com.

Figure 74 • **FCCC Configuration in SmartFusion2**



You must simply use the FCCC output to drive the CorePCIF design at the top-level, and any other logic in the FPGA required as well (unless the CLK_OUT as described above is used).

11.2 Clock and Reset Networks

The core includes global buffers for both the PCI clock and reset inputs. The buffered versions of these signals are provided on the CLK_OUT and RST_OUTN ports. These should be used for clocking and resetting any additional logic included in the FPGA running of the PCI clock.

The core also uses two additional global resources for internal routing of the high-fanout TRDYN and IRDYN nets, if required. Target cores will require IRDYN to be routed on a global; if the Master function is implemented, the TRDYN net will be routed on a global network.

In SX-A and RTSX-S implementations with both Master and Target functions enabled, the reset network is demoted to a normal buffer tree, as there are only three global resources available in these devices. They are required for the clock and the TRDYN and IRDYN nets.

11.3 Assigning Pin Layout Constraints

You can assign pins manually with the PinEditor tool or import them directly into Designer from the corresponding pin constraint file (PDC file). The pin file will be a PIN, GCF, or PDC file, depending on the FPGA family being used.

11.4 Pin Assignments

To be able to meet the critical PCI setup, hold, and clock-to-out requirements, it is critical that the PCI pin locations be assigned correctly. Two aspects need to be considered:

1. Pin assignments should minimize FPGA place-and-route issues. Pin assignment is extremely important in meeting the PCI setup, hold, and clock-to-out requirements.
2. Pin assignments should minimize PCB layout issues. The PCI specification limits the track lengths allowed on the PCB. Chapter 4 of the PCI specification details the requirements.

The PCI specification recommends that the pin order around the device align exactly with the add-in card (connector) pinout. The additional signals needed in 64-bit versions of the bus continue wrapping around

Downloaded from Arrow.com.

the component in a counterclockwise direction in the same order they appear on the 64-bit connector extension. **PCI Pinout** provides details of the recommended pin order.

Example pin files are provided in the *layout* directory for some of the possible FPGA family, device, and package combinations. These can be adapted to support other device/package combinations.

Each supported FPGA family has different requirements to minimize FPGA layout issues; these are detailed below.

11.4.1 SX-A and RTSX-S Families

The pins should be located around one side of the package in the order specified by the PCI specification. The pins should be located on the same side of the package where CLKA and CLKB are located.

1. Locate TRDYN and IRDYN close to the CLKA and CLKB pins, but do not use these pins.
2. Assign the rest of the PCI pins around the package in the order that will match the add-in connector. Do not use any of the CLK, QCLK, or HCLK pins.
3. Connect the PCI CLK to the HCLK pin.

11.4.2 ProASICPLUS Family

The pins should be located around one side of the package in the order specified by the PCI specification. The pins should be located on the west side of the die (in the pin editor, these pins will be identified by a “W” on the pin), depending on the package type. This may be the left or right side of the package.

1. Locate TRDYN and IRDYN close to the GL inputs.
2. Assign the rest of the PCI pins around the package in the order that will match the add-in connector. Do not use any of the special function pins.
3. Connect the PCI CLK to one of the GL input pins on the opposite side of the package.

11.4.3 Axcelerator and RTAX-S Families

The pins should be located around one side of the package in the order specified by the PCI specification. The pins should be located on the lower side of the package using the bank 4 and bank 5 I/O locations.

1. Locate TRDYN and IRDYN close to the routed clock inputs, but do not use these pins.
2. Assign the rest of the PCI pins around the package in the order that will match the add-in connector. Do not use any of the special function pins.
3. Connect the PCI CLK to an HCLK input pin.

Care should be taken to minimize the number of I/O banks used; the I/O banks used for PCI signals must be set to use PCI electrical levels that may be incompatible with other devices connected to the FPGA. When using large packages, exercise care in making sure that the PCI track lengths can be met with the planned pinout and FPGA location on the PCB. In some cases it may be necessary to move the PCI clock to an RCLK network to reduce the PCB track lengths.

11.4.4 Fusion, IGLOO/e, ProASIC3L, and ProASIC3/E Families

The pins should be located around one side of the package in the order specified by the PCI specification. Initially, identify an I/O bank that contains the global inputs G***.

1. Assign the TRDYN and IRDYN pins to, or close to, two of these global inputs.
2. Assign the rest of the PCI pins around the package in the order that will match the add-in connector. Leave one spare normal I/O pin vacant close to the global pins. Do not use any of the special function pins.
3. For 33 MHz operation, connect the PCI CLK to a global input. For 66 MHz operation, connect the CLK to the I/O pin left vacant close to the global inputs.

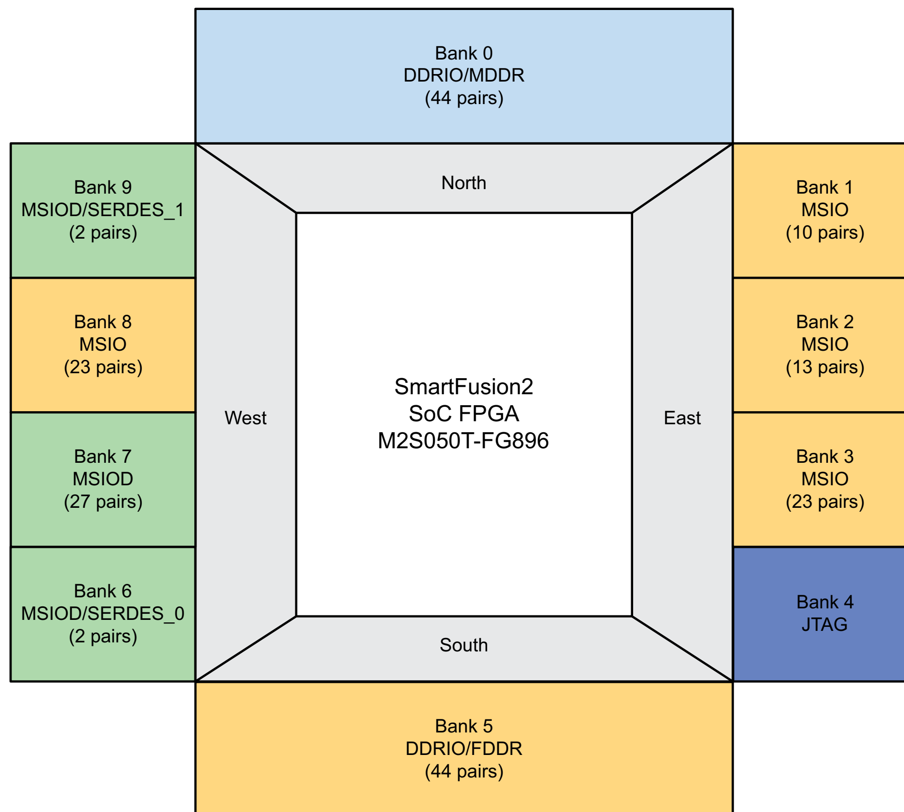
Care should be taken to minimize the number of I/O banks used; the I/O banks used for PCI signals must be set to use PCI electrical levels that may be incompatible with other devices connected to the FPGA.

Downloaded from Arrow.com.

11.4.5 SmartFusion2

There are typically four PCI-capable banks available on most devices, that is, the MSIO banks. For example, on the M2S050T device, Banks 1, 2, 3, and 8 are PCI capable. However, to meet timing and to ease board layout, only Banks 1, 2, and 3 should be used, that is, on the east edge of the package. If using unlocked automatic pin assignment, you need to make sure that bank 8 is not used for PCI IOs.

Figure 75 • **SmartFusion2 M2S050T Device**



Note:

- Do not use bank 8 for PCI I/Os
- Use bank 1, 2, and 3 for PCI I/Os

Alternatively, this may be achieved by manually assigning I/Os to banks 1, 2, and 3. After that, achieving timing should be done either by using the provided PDC file as a reference and modifying as necessary, or by following these steps:

1. Using the following configuration and pin assignment, run an initial Place and Route **using High-Effort, Timing-driven routing**.
   - Use `-iostd` PCI to ensure that I/Os are assigned to PCI compatible pins.
   - Where possible, force the placer to use I/O registers using the `-REGISTER Yes` switch and the `-OUT_REG Yes` or `-IN_REG Yes` depending on whether the register resides in the input or output path for that particular I/O. It is possible that they are both, if both Master and Target modes are enabled.
   - Do not assign PCI I/Os to any pins.
   - Depending on the data width, TRDYN and IRDYN may be best routed through a global net. In 64-bit mode, TRDYN and IRDYN are best routed locally. In 32-bit mode, TRDYN and IRDYN are best routed globally.
2. Lock down the I/O in the **Libero I/O constraints editor** or modify the PDC file by adding the pinname the `-fixed yes` switch to each `set_io` line.
3. Run **SmartTime** to perform static timing analysis.

Downloaded from Arrow.com.

- Adjust the (negative) programmable delay in the FCCC (as described in **Clocking in SmartFusion2**) until External Setup violations go away in Max Delay Analysis, but not so far as to introduce Internal Setup violations.
- Adjust all input delays until hold violations on paths ending at input ports go away in Min Delay Analysis, using the -IN_DELAY x switch in the PDC.

Note: Notes on Synthesis:

To ensure that timing is met, it is important to be able to push as many flip-flops as possible into the I/O registers. To accomplish this, keep the max fanout on output registers at 1 during synthesis. This can be accomplished using the following SDC attributes in the Synthesis constraint file (The instance name need to match with the instance name of your design):

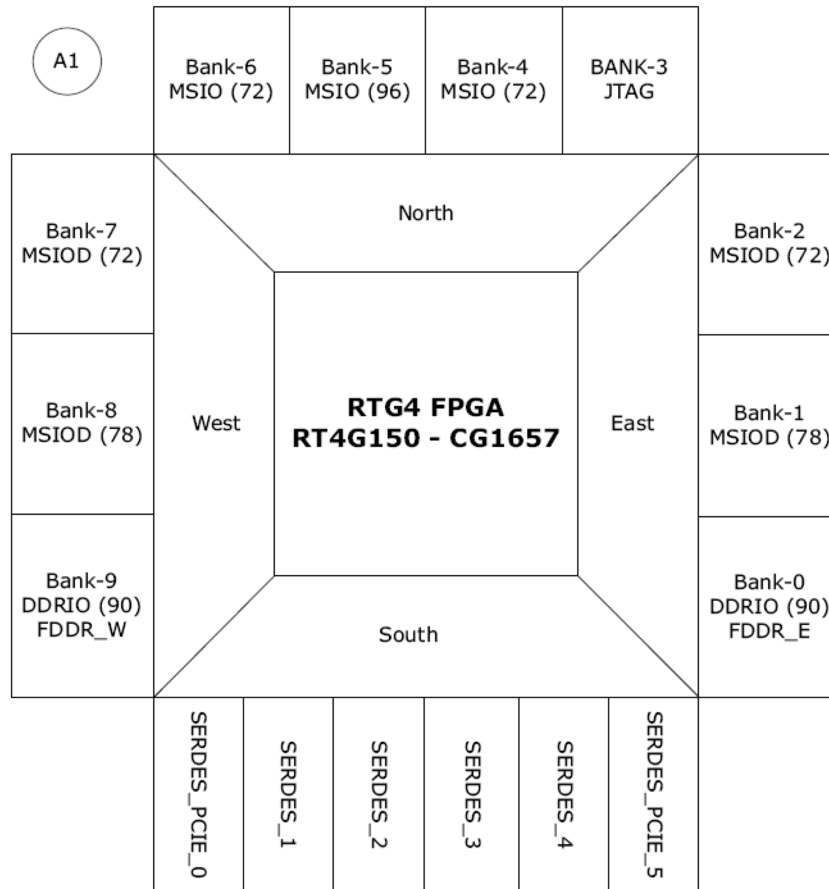
```
define_attribute
{{i:UCORE.MAKE_TARGET.DATAPATHE64.MAKE_DATAPATH_REGISTERS.AD_REGS[31:0]}}
{syn_preserve} {1}
define_attribute {{i:UCORE.MAKE\.UDMA.CBEN_PAD[7:0]}} {syn_replicate} {1}
define_attribute {{i:UCORE.MAKE\.UDMA.CBEN_PAD[7:0]}} {syn_maxfan} {1}
define_attribute
{{i:UCORE.MAKE_TARGET.DATAPATHE.MAKE_DATAPATH_REGISTERS.AD_REGS[31:0]}}
{syn_preserve} {1}
define_attribute {{i:UCORE.MAKE_TARGET.BurstE.UA1\.MAKE_ACK64_OUT.Q_INT}}
{syn_replicate} {1}
define_attribute {{i:UCORE.MAKE\.UDMA.MAKE_REQ64N1.Q_INT}} {syn_replicate}
{1}
define_attribute {{i:UCORE.MAKE\.UDMA.MAKE_REQ64N1.Q_INT}} {syn_maxfan} {1}
define_attribute {{i:UCORE.MAKE_TARGET.BurstE.UA1\.MAKE_ACK64_OUT.Q_INT}}
{syn_maxfan} {1}
define_attribute {{i:UCORE.MAKE_TARGET.BurstE.UM1\.MAKE_IRDY_OUT.Q_INT}}
{syn_replicate} {1}
define_attribute {{i:UCORE.MAKE_TARGET.BurstE.UM1\.MAKE_IRDY_OUT.Q_INT}}
{syn_maxfan} {1}
```

11.4.6 RTG4

Three PCI capable banks are available on most of the RTG4 devices, that is, MSIO banks. For example, on RT4G150-CG1657 device, Banks 4, 5, and 6 are PCI capable. To meet timing and ease board layout consecutive MSIO banks (bank 4, bank 5, and bank 6) should be used.

Downloaded from Arrow.com.

Figure 76 • **RTG4 RT4G150-CG1657 Device**

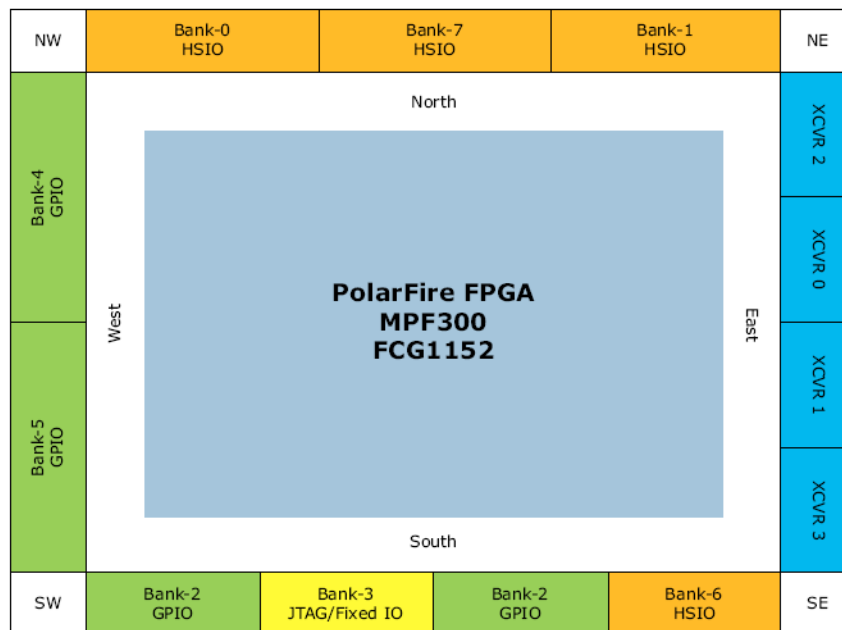


11.4.7 PolarFire

Three PCI capable banks are available on most of the PolarFire devices, that is, GPIO banks. For example, on MPF300-FCG1152 device, Bank 2, 4, and 5 are PCI capable. To meet timing and ease board layout consecutive GPIO banks (bank 4 and bank 5) should be used.

Downloaded from Arrow.com.

Figure 77 • **PolarFire MPF300-FCG1152 Device**



11.4.8 All Families

For 64-bit cores, the PAR64 pin should be located as close as possible to the upper CBEN pins. This creates a non-ideal PCB layout but significantly helps to meet the internal FPGA timing in 66 MHz, 64-bit implementations.

It is recommended that the pinout chosen be verified to check that PCI timing requirements can be met before PCB layout is completed. The core plus loopback database files supplied with the core can be used to verify the pinouts. Load a layout database from the chosen FPGA technology that matches the core function (T, TD, TM, or M; 32- or 64-bit; 33 or 66 MHz) and change the device type and package as required. Then modify the pinout to match your chosen pinout, re-run layout, and verify timing.

11.4.9 Meeting PCI Hold Requirements

The PCI hold time requirements should be checked post-layout. These can easily be found using the Minimum Delay Analysis View in the Timing Analyzer. All the hold times should be less than 0 ns. If any of the PCI inputs violate the hold time requirements, one of the following methods can be used to insert extra delay in the datapath to correct the hold time:

1. Modify the RTL source code, if available, to insert BUFD cells between the IOPAD and the registers violating hold time requirements. This can be done easily in the DEL_BUFF module, which allows the number of delay buffers inserted on each PCI input to be specified. Re-run synthesis and layout.
2. For families that support programmable input delays (Axcelerator, RTAX-S, ProASIC3E, and SmartFusion2 and IGLOOe), the I/O pad can be configured to insert additional delay.[1] This is a good way to correct hold problems on the AD bus; however, adding additional input buffer delays on the control inputs TRDYN, IRDYN, FRAMEN, etc., may cause other endpoints from these inputs to violate the PCI setup times.
3. Export a netlist from Designer. Modify the netlist to insert BUFD cells between the IOPAD and the registers violating hold time requirements. Re-run layout with the incremental layout feature enabled.
4. Using ChipPlanner, move registers that have a hold time violation away from the I/O pad to increase the delay and fix the hold time violation. Re-run layout with the incremental layout feature enabled.

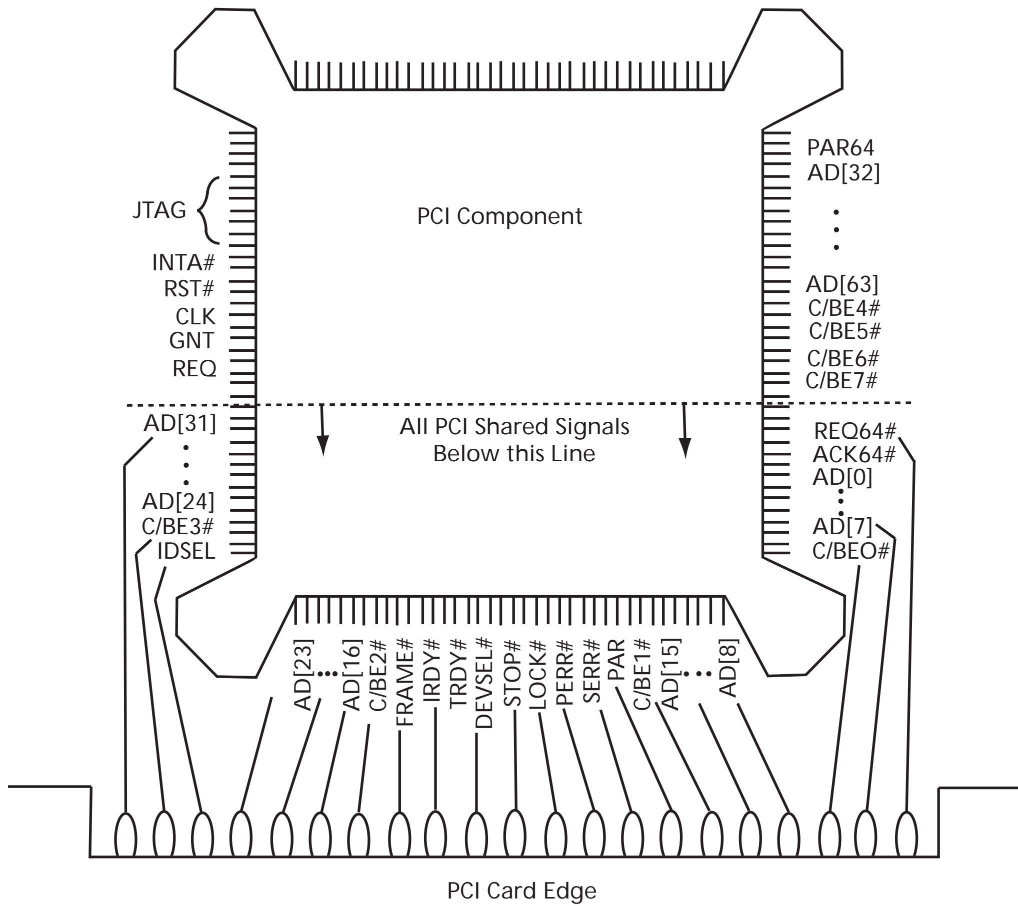
1. Use PinEditor to select the I/O bank. Right-click the colored I/O bank in the GUI to open the Configure I/O Bank dialog box. Once you set the bank delays, you can set the input delays on all PCI pins.

Downloaded from Arrow.com.

12 PCI Pinout

Figure 78 shows the recommended pin ordering around the package.

Figure 78 • **Recommended PCI Pin Ordering**



Downloaded from Arrow.com.

13 Synthesis Timing Constraints

The required timing constraints are given in Table 55.

Table 55 • **Synthesis Timing Constraints**

Frequency (MHz)	PCI Specification (ns)			Synplicity Constraints (ns)		
	Ports	**Setup**	**Clock to Output**	**Period**	**Input Delay**	**Output Delay**
33	AD CBEN DEVSELN FRAMEN IRDYN TRDYN PAR PERRN SERRN STOPN TRDYN PAR64 ACK64N REQ64N	7	11	30	23	19
	IDSEL GNTN	10			20	
	INTAN REQN		11			19
66	AD CBEN DEVSELN FRAMEN IRDYN TRDYN PAR PERRN SERRN STOPN TRDYN PAR64 ACK64N REQ64N	3	6	15	12	9
	IDSEL GNTN	5			10	
	INTAN REQN		6			9

Downloaded from Arrow.com.

14 Place-and-Route Timing Constraints

The required timing constraints are given in Table 56.

Table 56 • **Place-and-Route Timing Constraints**

Frequency (MHz)	PCI Specification (ns)				Designer Constraints (ns)			
	Ports	**Setup**	**Hold**	**Clock to Output**	**Period**	**Input Delay**	**Input Hold**	**Output Delay**
33	AD CBEN DEVSELN FRAMEN IRDYN TRDYN PAR PERRN SERRN STOPN TRDYN PAR64 ACK64N REQ64N	7	0	11	30	23	0	19
	IDSEL GNTN	10	0			20	0	
	INTAN REQN			11				19
66	AD CBEN DEVSELN FRAMEN IRDYN TRDYN PAR PERRN SERRN STOPN TRDYN PAR64 ACK64N REQ64N	3	0	6	15	12	0	9
	IDSEL GNTN	5	0			10	0	
	INTAN REQN			6				9

Note: Timing closure at 66MHz is highly impacted by the other aspects of the design and resource allocation. For highly congested designs it may not be possible in some cases to achieve a 66MHz solution. Priority should be given to closing timing on the CorePCIF before moving onto other areas of the design. To close the timing of CorePCIF, the output delay value should be adjusted with respect to the PCI device (on board) timing constraints at system level.

Downloaded from Arrow.com.

15 Verification Testbench Tests

The verification testbench performs the tests in Table 57.

Table 57 • **Verification Testbench Tests**

Test	Description
01	Simple Read and Write Test
02	All BARs Read/Write Test
03	Byte Enable Test
04	DEVSEL Timing Test
05	Address Parity Error Test
07	Interrupt Test
08	Data Parity Error Test
10	Two Target Test; checks that two targets do not interfere with each other.
11	Target Disconnect and Retry Test
13	Target Abort Test
14	Back-to-Back Transfer Test
15	BAR Overflow Test
16	Memory Read Line, Memory Read Multiple, and Memory Write & Invalidate Test
17	Unaligned Address Transfer Test
18	Target Dataflow Test with variable transfer rates
19	FIFO Interface Test with variable transfer rates
20	BAR Select Test; verifies BAR decode logic.
21	Read Byte Handshake Tests
22	Hot-Swap Interface Tests
23	Configuration Cycle Tests
24	Additional Target Retry/Disconnect Tests
25	Multiple FIFO Tests
26	User Target Routine
27	FIFO Status Register Tests
40	Simple DMA Transfer Test
41	Backend Control during Cycle Test
42	DMA Single Transfer Test
43	DMA Poll Status Test
44	DMA Counter Tests
45	DMA Mega Test; multiple DMA tests at the same time
60	Master Mode Test
47	DMA Single Transfer Test with DMA completion interrupt enabled
48	DMA Poll Status Test with DMA completion interrupt enabled

Downloaded from Arrow.com.

Microsemi Power Matters.

Table 57 • **Verification Testbench Tests *(continued)***

Test	Description
49	DMA Transfer Test with Busy_Master assertion
50	DMA Transfer Test with Stall_Master assertion
51	Byte Enable Test with DMA transfers
52	Byte Enable Test on Backend Registers
53	DMA with Maximum Transfer Length
54	DMA Dataflow with variable transfer rates and FIFO recovery
55	DMA Burst Length Tests with FIFO recovery
56	DMA Auto Transfer Test with FIFO recovery; checks DMA starting and stopping conditions.
57	Fast Master Back-to-Back Test, DMA termination and another Master accessing the core immediately.
58	Direct Mode DMA Transfer Test
59	Miscellaneous DMA Tests
60	Backend Configuration Access Tests
70	Master Mode Test
71	User DMA Routine
98	Quick Test (all slots)
99	Exhaustive Tests (all tests / all slots)
S..n	Test 00–99 on Slot S (e.g., 112)
Q	Quit Testbench

Downloaded from Arrow.com.

16 VHDL User Testbench Procedures

The following is a list of the supported procedure calls in the VHDL user testbench. Microsemi recommends that you examine the *testbench.vhd* file to understand how to use these procedure calls.

```
config_write(SLOT,ADDRESS,DATA_DW ,PCICMD,PCISTAT,MSETUP);
config_write(SLOT,ADDRESS,DATA_INT ,PCICMD,PCISTAT,MSETUP);
config_write(SLOT,ADDRESS,N,DATA(0 to N-1),PCICMD,PCISTAT,MSETUP);
config_read (SLOT,ADDRESS,DATA_INT ,PCICMD,PCISTAT,MSETUP);
config_read (SLOT,ADDRESS,DATA_DW ,PCICMD,PCISTAT,MSETUP);
config_read (SLOT,ADDRESS,N,DATA(0 to N-1),PCICMD,PCISTAT,MSETUP);
memory_write(ADDRESS,DATA_DW ,PCICMD,PCISTAT,MSETUP);
memory_write(ADDRESS,DATA_INT ,PCICMD,PCISTAT,,MSETUP);
memory_write(ADDRESS,DATAH_DW,DATAL_DW,PCICMD,PCISTAT,MSETUP);
memory_write(ADDRESS,DATAH_INT,DATAL_INT,PCICMD,PCISTAT,MSETUP);
memory_write(ADDRESS,N,DATA(0 to N-1),PCICMD,PCISTAT ,MSETUP);
memory_read (ADDRESS,DATA_DW ,PCICMD,PCISTAT,MSETUP);
memory_read (ADDRESS,DATA_INT ,PCICMD,PCISTAT,MSETUP);
memory_read(ADDRESS,DATAH_DW,DATAL_DW,PCICMD,PCISTAT,MSETUP);
memory_read(ADDRESS,DATAH_INT,DATAL_INT,,PCICMD,PCISTAT,MSETUP);
memory_read (ADDRESS,N,DATA(0 to N-1),PCICMD,PCISTAT,MSETUP);
compare_data(ERRCOUNT,"Message",EXP_INT,GOT_INT);
compare_data(ERRCOUNT,"Message",EXP_DWORD,GOT_DWORD);
compare_data(ERRCOUNT,"Msg",EXP_DATA(0 to N-1),GOT_DATA(0 to N-1));
be_write(ADDRESS,DATA_DW , BYTEEN, PCICMD,PCISTAT);
be_write(ADDRESS,DATA_INT, BYTEEN, PCICMD,PCISTAT);
be_read (ADDRESS,DATA_DW , PCICMD,PCISTAT);
be_read (ADDRESS,DATA_INT, PCICMD,PCISTAT);
tb_write_ahb(ADDRESS,COUNT,DATA_DW(0 to N-1),AHB_CONTROL,AHB_STATUS)
tb_write_ahb(ADDRESS,COUNT,DATA_INT(0 to N-1),AHB_CONTROL,AHB_STATUS)
tb_write_ahb(ADDRESS,DATA_DW,AHB_CONTROL,AHB_STATUS)
tb_write_ahb(ADDRESS,DATA_INT,AHB_CONTROL,AHB_STATUS)
tb_read_ahb(ADDRESS,COUNT,DATA_DW(0 to N-1),AHB_CONTROL,AHB_STATUS)
tb_read_ahb(ADDRESS,COUNT,DATA_INT(0 to N-1),AHB_CONTROL,AHB_STATUS)
tb_read_ahb(ADDRESS,DATA_DW,AHB_CONTROL,AHB_STATUS)
tb_read_ahb(ADDRESS,DATA_INT,AHB_CONTROL,AHB_STATUS)
```

The parameters to the above procedure calls are described in Table 58. To simplify the parameters, some predefined types are used. These are defined in the *misc.vhd* package.

```
subtype NIBBLE is std_logic_vector ( 3 downto 0);
subtype DWORD is std_logic_vector (31 downto 0);
type DWORD_ARRAY is array ( INTEGER range <>) of DWORD;
```

Table 58 • **Procedure Call Parameters**

Parameter	Type	Description
SLOT	INTEGER	PCI slot number to use for configuration cycles When 0, will set the eight upper address bits to 01 hex. When 1, will set the eight upper address bits to 02 hex, etc. The testbench connects address bit 25 to the core IDSEL input; therefore, the slot number should be set to 1.
ADDRESS	INTEGER	Address for the PCI cycle
DATA_DW	DWORD	Data word
DATAH_DW	DWORD	Upper 32 bits of a 64-bit data word

Downloaded from Arrow.com.

Table 58 • **Procedure Call Parameters *(continued)***

<table>
<tr><th>Parameter</th><th>Type</th><th colspan="3">Description</th></tr>
<tr><td>DATAL_DW</td><td>DWORD</td><td colspan="3">Lower 32 bits of a 64-bit data word</td></tr>
<tr><td>DATA_INT</td><td>INTEGER</td><td colspan="3">Data word</td></tr>
<tr><td>DATAH_INT</td><td>INTEGER</td><td colspan="3">Upper 32 bits of a 64-bit data word</td></tr>
<tr><td>DATAL_INT</td><td>INTEGER</td><td colspan="3">Lower 32 bits of a 64-bit data word</td></tr>
<tr><td>PCICMD</td><td>TPCICMD</td><td colspan="3">Record used to communicate within the testbench. Allows the procedure to start the PCI Master cycle.</td></tr>
<tr><td>PCISTAT</td><td>TPCISTAT</td><td colspan="3">Record used to communicate within the testbench. Allows the procedure to monitor the PCI Master cycle.</td></tr>
<tr><td rowspan="6">MSETUP</td><td rowspan="6">TMSETUP</td><td colspan="3">Record used to set the Master transfer rates. This contains four fields that may be altered.</td></tr>
<tr><td>Name</td><td>Type</td><td>Description</td></tr>
<tr><td>IRDYRATE0</td><td>INTEGER</td><td>Initial delay from FRAME to IRDY assertion</td></tr>
<tr><td>IRDYRATEN</td><td>INTEGER</td><td>Subsequent delay between IRDY assertions</td></tr>
<tr><td>PCI64</td><td>BOOLEAN</td><td>Indicates whether to request a 64-bit transfer.</td></tr>
<tr><td>ERROR</td><td>TERROR</td><td>Allows errors conditions to be inserted. Should be set to “NONE” for normal operation. Supported error conditions are described in the VHDL source files.</td></tr>
</table>

Downloaded from Arrow.com.

17 Verilog User Testbench Procedures

Following is a list of the supported tasks in the Verilog user testbench. Microsemi recommends that you examine the *testbench.v* file to understand how to use these tasks.

```
// PCI configuration cycles
config_write (SLOT,CADDRESS,COUNT);
config_read (SLOT,CADDRESS,COUNT);

// PCI memory cycles
memory_write (ADDRESS,COUNT,PCI64);
memory_read (ADDRESS,COUNT,PCI64);
compare_data (ERRCOUNT,COUNT);

// Writes and reads to and from the core backend interface
be_write (BADDRESS,WDATA, BYTEEN);
be_read (BADDRESS,RDATA);
// AHB read and write cycles
tb_write_ahb (ADDRESS,COUNT);
tb_read_ahb (ADDRESS,COUNT);
compare_ahb_data(ERRCOUNT,COUNT);
```

The parameters to the above tasks are described in Table 59 and Table 60. Data for the PCI configuration and PCI and AHB memory read and write cycles is passed in the pciwdata and pcirdata, or ahbwdata and ahbrdata, global arrays rather than through the task parameters.

Table 59 • **Global Descriptions**

Globals	Type	Description
pciwdata	reg [31:0] [0:31]	This is an array in which the user sets up the data that will be written before calling the memory_write or config_write tasks. For 64-bit operations, the lower DWORD is specified in the odd addresses and the upper DWORD in the even addresses.
pcirdata	reg [31:0] [0:31]	This is an array by which the memory_read and config_read functions return data. For 64-bit operations, the lower DWORD is specified in the odd addresses and the upper DWORD in the even addresses.

Table 60 • **Parameter Descriptions**

Parameters	Type	Description
SLOT	reg [2:0]	PCI slot number to use for configuration cycles. When 0, will set the eight upper address bits to 01h. When 1, will set the eight upper address bits to 02h, etc. The testbench connects address bit 25 to the core IDSEL input; therefore, the slot number should be set to 1.
CADDRESS	reg [7:0]	Configuration space address
COUNT	reg [7:0]	Number of DWORDs to be written, read, or compared. If 64-bit operation is enabled, this must be an even number. The maximum count is thirty-two 32-bit transfers or sixteen 64-bit transfers.
ADDRESS	reg [31:0]	Memory space address
PCI64	reg	When 1, the testbench will request a 64-bit transfer.
ERRCOUNT	inout reg [31:0]	The compare routine will increment this value if it detects any errors. At the end of a test sequence, it can indicate the total number of errors.
BADDRESS	reg [7:0]	Backend address

Downloaded from Arrow.com.

Table 60 • **Parameter Descriptions**

Parameters	Type	Description
WDATA	reg [31:0]	Data to be written to the core backend
RDATA	output reg [31:0]	Data read from the core backend
BYTEEN	reg [3:0]	Byte enables to backend writes should be set to 4'b1111.

Downloaded from Arrow.com.

18 Ordering Information

18.1 Ordering Codes

CorePCIF can be ordered through your local Microsemi sales representative. It should be ordered using the following number scheme: CorePCIF-XX, where XX is listed in Table 61.

Table 61 • **Ordering Codes**

XX	Description
OM	RTL for Obfuscated RTL – multiple-use license
RM	RTL for RTL source – multiple-use license

Downloaded from Arrow.com.