Multi-Port Memory Controller 2 (MPMC2)

User Guide

UG253 (v2.0) April 6, 2007

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/31/06	1.0	Initial Xilinx release.
08/14/06	1.1	Added description of C_MPMC2_0_RD_FIFO_LATENCY parameter. Corrected RdFIFO_Flush signal name.
10/20/06	1.2	Miscellaneous edits for clarity and to address unsupported features. Changes to several I/O and parameter tables.
04/06/07	2.0	Added CDMAC information from XAPP535, “*High Performance Multi-Port Memory Controller Application Guide*.” Added Error Correction Code (ECC) and Performance Monitoring (PM). Modifications throughout to reflect MPMC2 functionality.

Table of Contents

Appendix A: Communication Direct Memory Access Controller (CDMAC)

Appendix B: Error Correcting Code (ECC)

Appendix C: MPMC2 Performance Monitoring

About This Guide

The second generation Multi-Port Memory Controller (MPMC2) provides an enhanced, highly configurable set of features and capabilities for high-performance systems. This user guide provides example system topologies and covers internal and port interface module architectures. This user guide also provides guidelines for using the Xilinx Platform Studio tool to integrate an MPMC2 processor core (pcore) into a Base System Builder (BSB) project. The guide includes appendixes for the Communication Direct Memory Access Controller (CDMAC), Error Correction Code (ECC), and Performance Monitoring (PM) features. The document is organized as follows:

- "Multi-Port Memory Controller 2"
- Appendix A, "Communication Direct Memory Access Controller (CDMAC)"
- Appendix B, "Error Correcting Code (ECC)"
- Appendix C, "MPMC2 Performance Monitoring"

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File →Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}
Vertical bar |	Separates items in a list of choices	**`lowpwr =`**{**`on`**|**`off`**}

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis . . .	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Platform FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Multi-Port Memory Controller 2

Introduction

This document describes the Multi-Port Memory Controller2 (MPMC2) and provides the following information about MPMC2:

- "Topologies"
- "Memory Controller Architecture"
- "PIM Architecture"
- "I/O Signals and Parameters"
- "Design Implementation"
- "Dragging and Dropping MPMC2 pcores in XPS Version 8.2 or Later"
- "Using the Base System Builder"

This introduction lists the MPMC2 features and supported device families, provides an overview of the Port Interface Modules (PIMs), and lists the available additional resources.

Features

The MPMC2 includes the following features:

- Communications Direct Memory Access Controller (CDMAC) support
- Double Data Rate (DDR) and DDR2 SDRAM memory support
- DIMM support (registered and unbuffered)
- Error Correcting Code (ECC) support
- Parameterizable number of ports (1 to 8)
- Parameterizable number of data bits to memory (8, 16, 32, 64)
- Parameterizable configuration of data path FIFOs
  (Data path configurations and pipeline settings cannot differ across ports)
- Performance Monitoring (PM) support
- User configuration of arbitration algorithms

Supported Device Families

The MPMC2 module supports the following device families:

- Virtex-II Pro (see release notes)
- Virtex-4
- Virtex-5
- Spartan (3/3E/3A) (see release notes)

Note: Some part types are too small to support all parameter configurations.

Overview

MPMC2 is a fully parameterizable memory controller that supports Double Data Rate (DDR and DDR2) memory. MPMC2 provides access to memory for one to 8 ports where each port can be chosen from a set of Port Interface Modules (PIMs) that permit connectivity into PowerPC® 405 processor, MicroBlaze™, CoreConnect, and the MPMC2 Native Port Interface (NPI) structures. MPMC2 also supports the Communications Direct Memory Access Controller (CDMAC) that provides full-duplex, high-bandwidth LocalLink interfaces into memory. Additionally, MPMC2 provides Error Correcting Code (ECC) and Performance Monitoring (PM).

Supported Port Interface Modules (PIMs)

The MPMC2 module connects to the following pre-built PIMs:

- "Communication Direct Memory Access Controller (CDMAC)"
- "PowerPC 405 Processor Data-Side (DSPLB)"
- "PowerPC 405 Processor Instruction-Side Local Bus (ISPLB)"
- "Native Port Interface (NPI)"
- "On-Chip Peripheral Bus (OPB)"
- "Processor Local Bus (PLB)"
- "MicroBlaze Xilinx CacheLink (XCL)"

System Diagram

Figure 1-1 is a diagram of a system that uses an MPMC2 module.

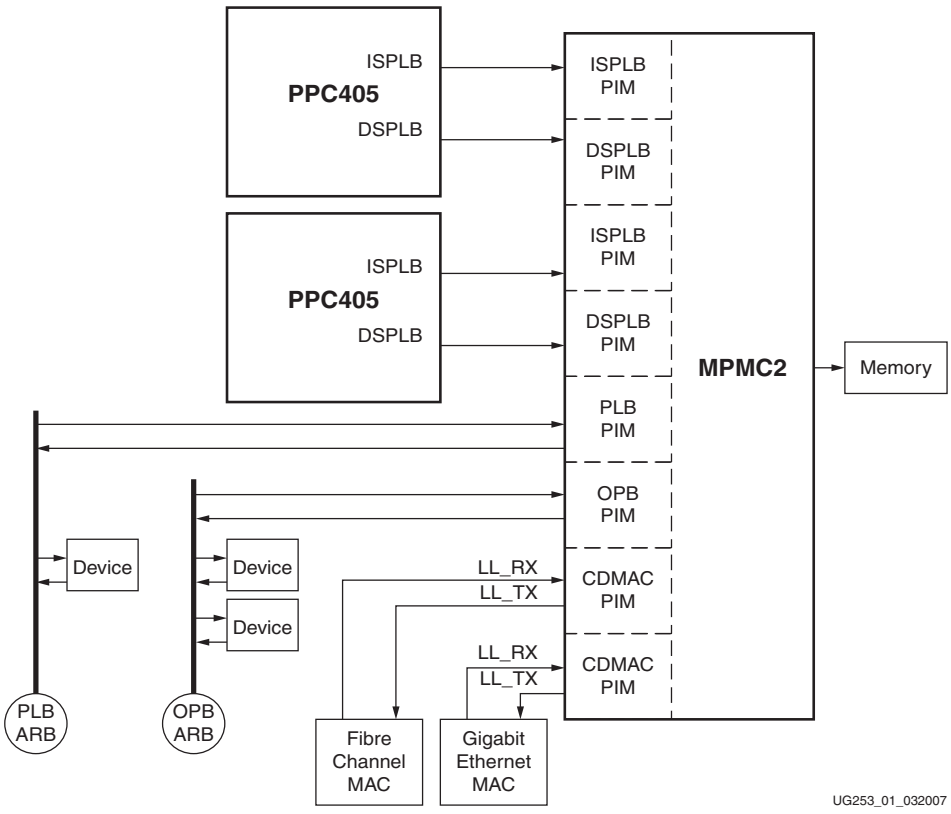


Figure 1-1: **Example PowerPC System**

Additional Resources

- Xilinx main support page: http://www.xilinx.com/support/mysupport.htm
- Xilinx answer browser page: http://www.xilinx.com/xlnx/xlinx.xil_ans_browser.jsp
- Xilinx contact support webcase page:
  http://www.xilinx.com/support/clearexpress/websupport.htm
- Xilinx MPMC2 reference design page: http://www.xilinx.com/mpmc2
- *Multi-Port Memory Controller (MPMC2) IP Configurator GUI*
  .<MPMC2_Install_Directory>/docs/ug245.pdf
- Xilinx memory solutions page: http://www.xilinx.com/memory
- EDK main web page:
  http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
- EDK tutorials and design example page:
  http://www.xilinx.com/ise/embedded/edk_examples.htm
- EDK documentation page contains links to the following documents:
  PowerPC® 405 Processor Block Reference Manual
  LocalLink Interface Specification
  http://www.xilinx.com/ise/embedded/edk_docs.htm
- ISE documentation page:
  http://www.xilinx.com/ise/logic_design_prod/foundation.htm
- Memory website contains links to the following documents:
  Memory Interfaces Data Capture Using Direct Clocking Technique
  MPMC2 Application Note
  http://www.xilinx.com/products/design_resources/mem_corner
- *MicroBlaze Processor Reference Guide*
  http://www.xilinx.com/ise/embedded_design_prod/index.htm
- Spartan-3 generation FPGAs page:
  http://www.xilinx.com/products/silicon_solutions/fpgas/spartan_series/index.htm
- Virtex-4 page:
  http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm
- Virtex-5 page:
  http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/index.htm
- Design resources page:
  http://www.xilinx.com/products/design_resources/proc_central/index.htm
- Development boards page: http://www.xilinx.com/products/devboards/index.htm
- IBM CoreConnect documentation website contains the following documents:
  CoreConnect Device Control Register Bus: Architecture Specification:
  IBMCoreConnect 64-Bit Processor Local Bus: Architecture Specification:
  IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architectural Specification
  http://www-306.ibm.com/chips/techlib/techlib.nsf/pages/main

IBM CoreConnect Documentation and Licensing

The EDK uses the IBM CoreConnect Bus Architecture to build systems with interconnected IP. EDK also integrates with the IBM CoreConnect Toolkit, which provides a number of features, enhancing design productivity and allows you to get the most from the EDK.

To obtain the IBM CoreConnect toolkit, you must be a licensee of the IBM CoreConnect Bus Architecture. Licensing CoreConnect provides access to documentation, Bus Functional Models, Hardware IP, and the toolkit.

- Licensing CoreConnect via the Xilinx website: http://www.xilinx.com/coreconnect
- Licensing CoreConnect from IBM: http://www.ibm.com/chips/products/coreconnect

Topologies

You can use MPMC2 to build different system topologies, some of which are available for the first time. Because of software variables, device bus transactions, memory latency, and speed, each system is capable of different data bandwidths. MPMC2 allows you to quickly build systems with different topologies and compare the performance.

The following sections show examples of the available system topologies:

- "Stand-Alone Topology"
- "Basic Topology"
- "Basic MicroBlaze Topology"
- "Dual-Processor (PowerPC 405)"
- "Shared D-Side Topology"
- "Standard PowerPC 405 CoreConnect Topology"
- "Standard MicroBlaze CoreConnect Topology"
- "Dual-Processor (PowerPC 405 and MicroBlaze) Topology"

Stand-Alone Topology

8 custom devices can be connected to the MPMC2 module via the NPI PIM shown in Figure 1-2. This topology is useful in systems without processors, where multiple devices need access to memory.

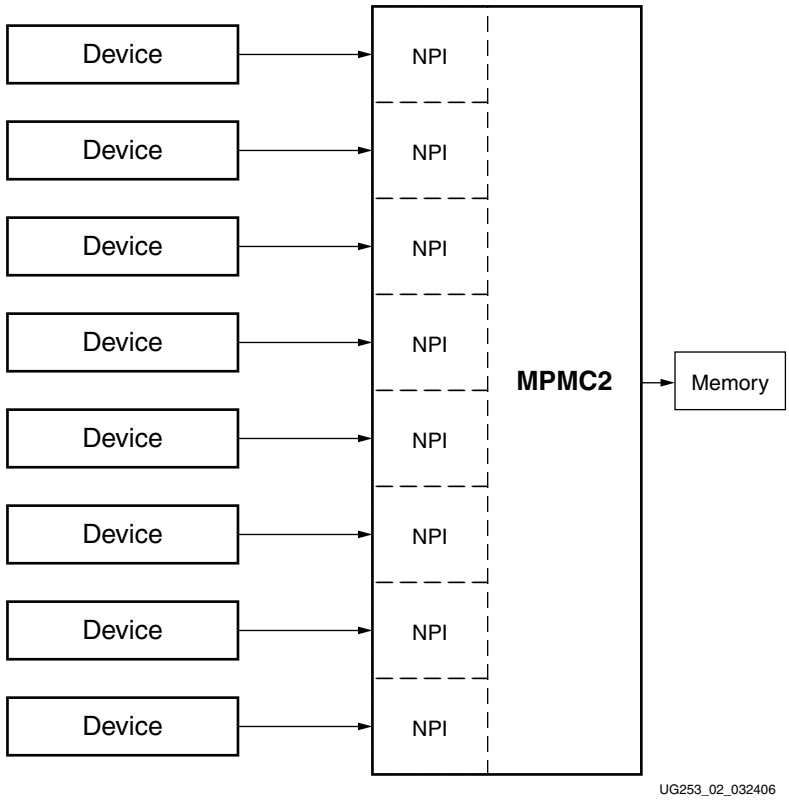


Figure 1-2: **Stand-Alone Topology**

Basic Topology

Figure 1-3 shows a common PowerPC 405 processor system layout. The MPMC2 module provides direct memory access to the processor ISPLB and DSPLB interfaces. A standard PLB port is defined for use with high-speed PLB devices. An OPB port is provided for simpler, slower speed devices.

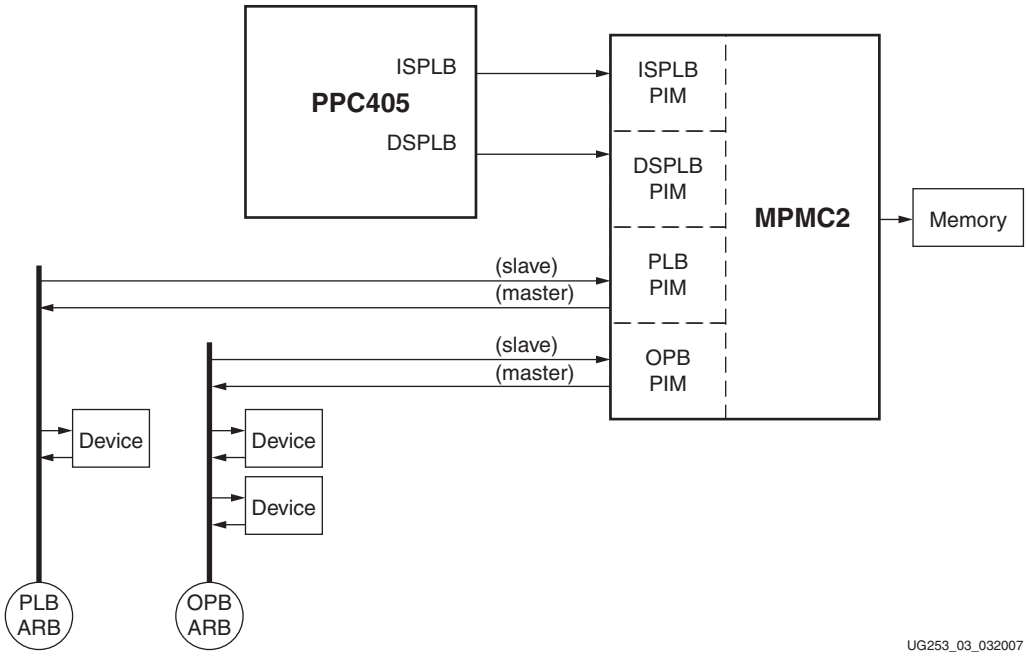


Figure 1-3: **Basic PowerPC 405 Processor Topology**

Basic MicroBlaze Topology

Figure 1-4 shows a common MicroBlaze system layout. The MPMC2 module provides direct memory access to the processor IXCL and DXCL interfaces. A standard PLB port is defined for use with high-speed PLB devices. An OPB port is provided for more simple, slower speed devices. The MicroBlaze processor can also be connected directly to the OPB bus provided by the OPB PIM.

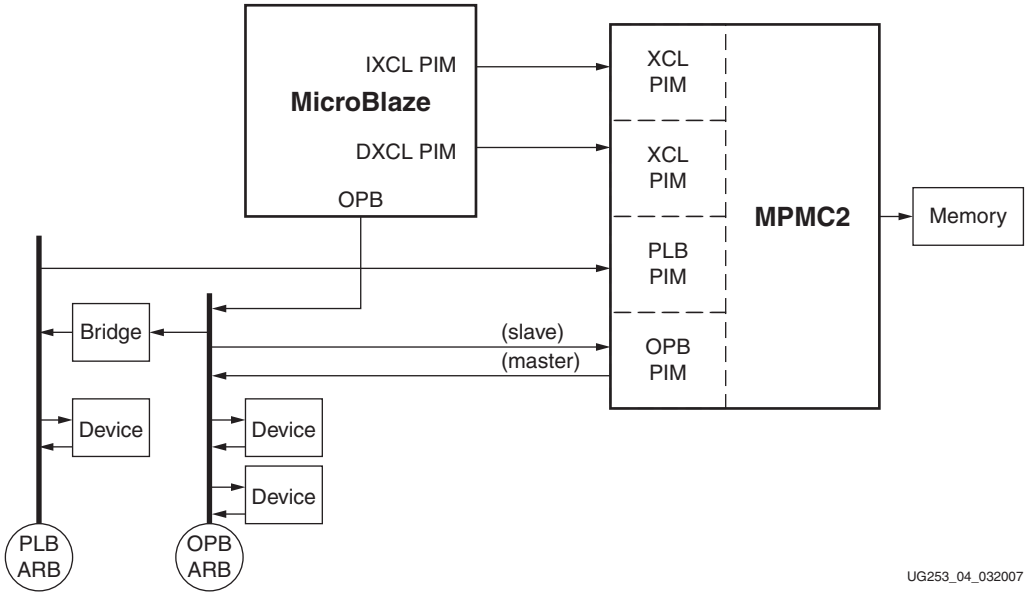


Figure 1-4: **Basic MicroBlaze Topology**

Dual-Processor (PowerPC 405)

Figure 1-5 shows two PowerPC 405 processors connected directly to an MPMC2 module. Both can act as masters on the OPB and PLB ports by designating address ranges that are decoded as master requests.

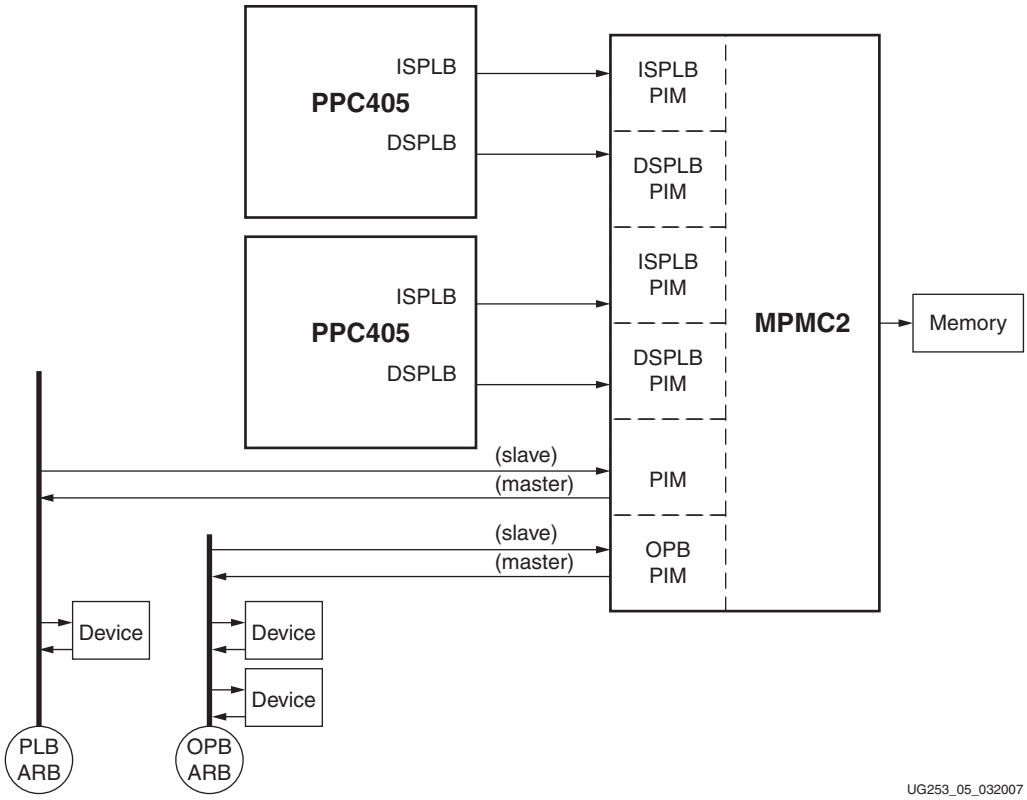


Figure 1-5: **Typical Dual-Processor (PowerPC 405 processor)**

Shared D-Side Topology

Figure 1-6 shows one PowerPC 405 processor sharing its DSPLB port with standard PLB and OPB buses, while the other PowerPC 405 processor has direct access to memory via dedicated DSPLB and ISPLB ports. In this example, one processor can handle device communication leaving the other free to process less device-intensive jobs.

Note: Only the processor that is connected to the DSPLB PIM can access the PLB or OPB ports directly as a master.

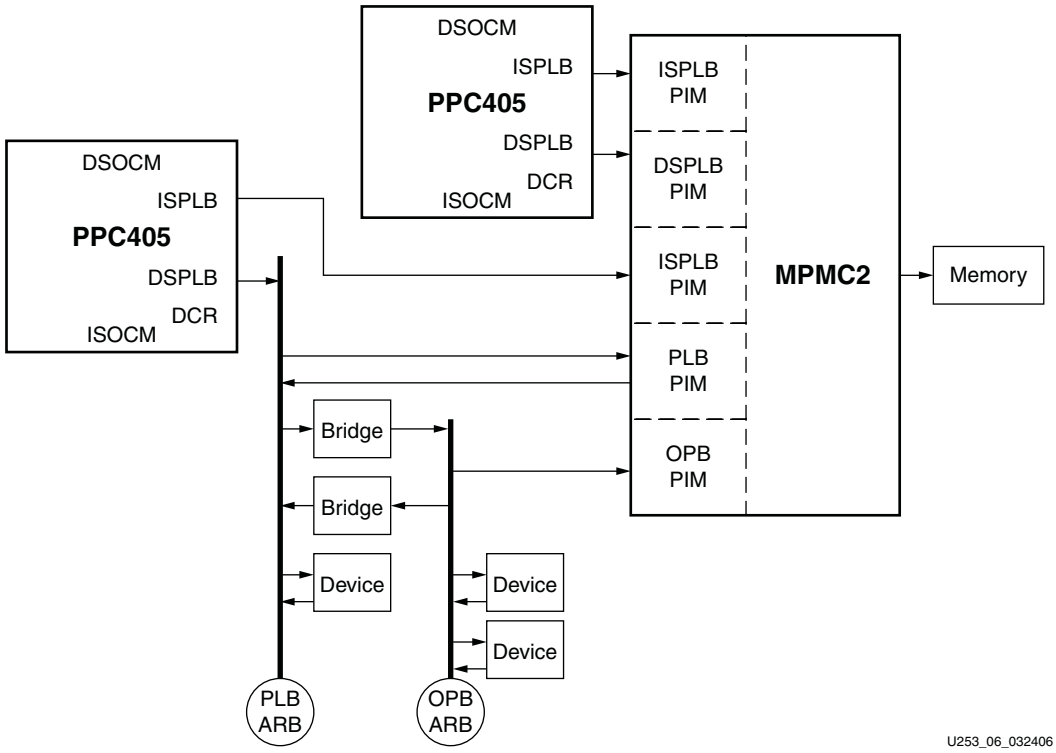


Figure 1-6: **Shared D-Side Topology**

Standard PowerPC 405 CoreConnect Topology

An MPMC2 module easily fits into existing PowerPC 405 CoreConnect-based systems as a single port memory controller as shown in Figure 1-7. This is particularly useful for exploring various system topologies using an existing design-without requiring a lot of modifications.

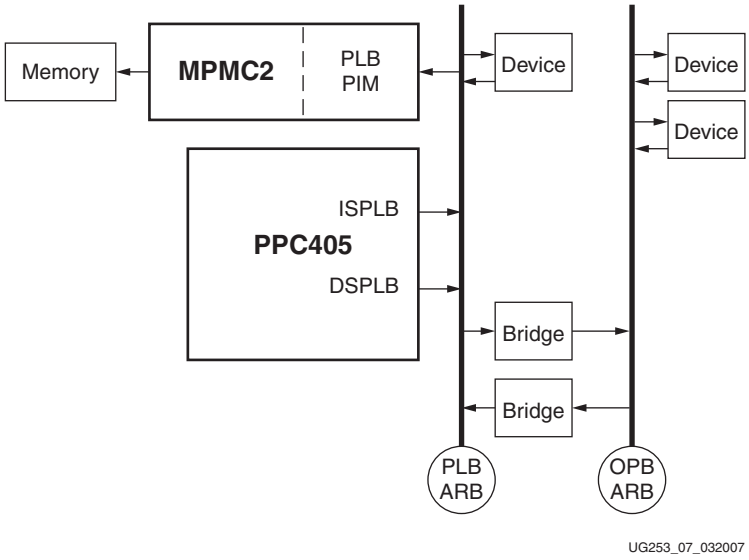


Figure 1-7: **Standard PowerPC 405 CoreConnect Topology**

Standard MicroBlaze CoreConnect Topology

An MPMC2 module also easily fits into existing MicroBlaze CoreConnect-based systems as a single-port memory controller as shown in the Figure 1-8.

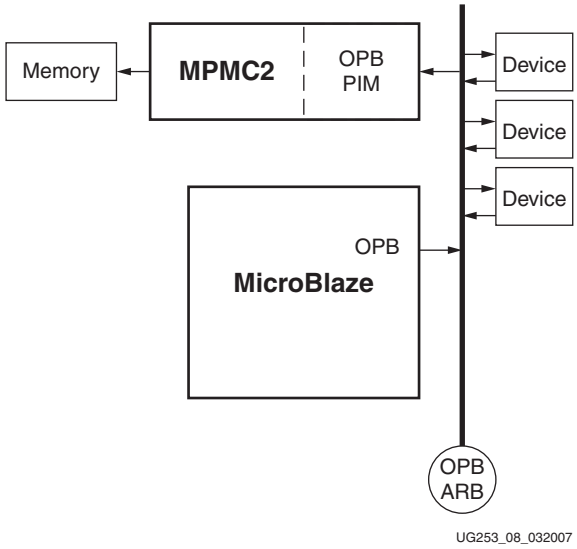


Figure 1-8: **Standard MicroBlaze CoreConnect Topology**

Dual-Processor (PowerPC 405 and MicroBlaze) Topology

In Figure 1-9, a PowerPC 405 processor and MicroBlaze processor are connected to the MPMC2 module for access to memory. The MicroBlaze processor can also be connected directly to the OPB bus.

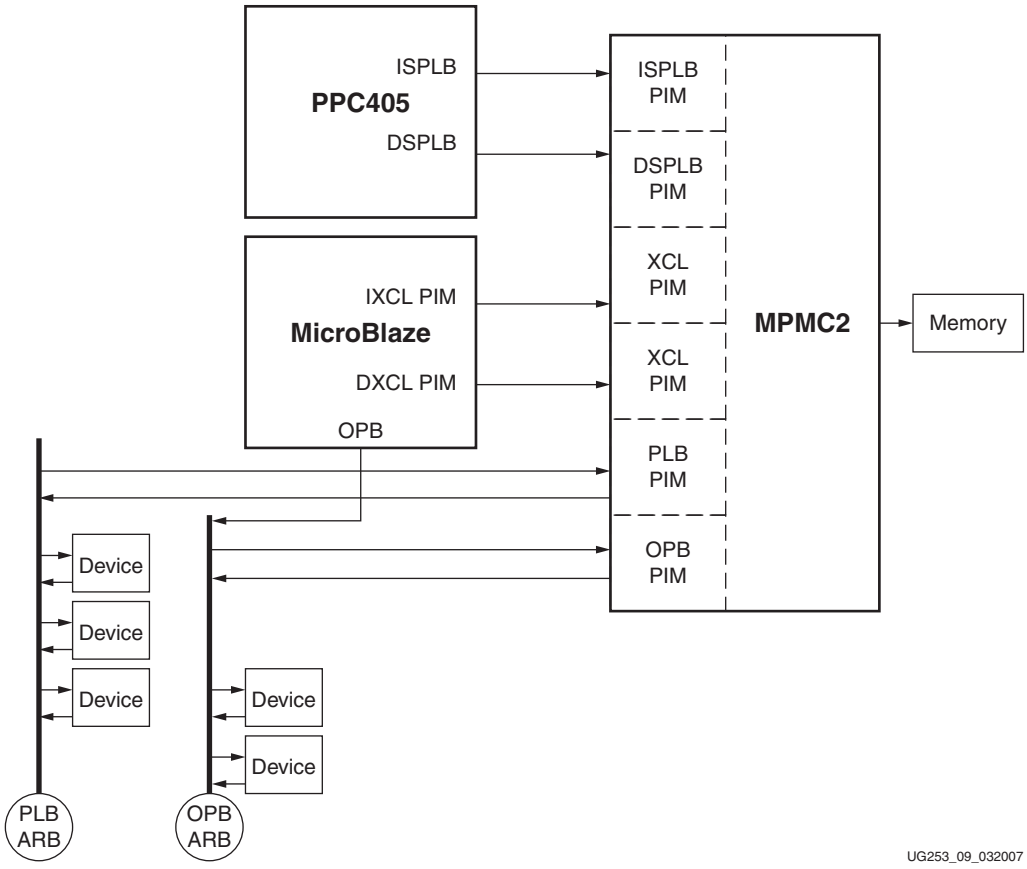


Figure 1-9: **Dual Processor (PowerPC 405 and MicroBlaze) Topology**

Memory Controller Architecture

The MPMC2 module has four major components:

- "Address Path"
- "Data Path"
- "Control Path"
- "Physical (PHY) Interface"

Additionally, the MPMC2 architecture contains:

- "Error Correcting Code (ECC)"that resides between the control path and the PHY interface
- "BRAM Use Models" that describes the data path configuration to and from the PIMs
- "IDELAY Controller" that details IDELAY controller instantiation (on Virtex-4 and Virtex-5)
- "Reset" that provides informations regarding the master reset for the MPMC2 core and PIMS
- "Transaction Ordering and Memory Coherency" that explains how to achieve both within the MPMC2 architectural structure

Figure 1-10 shows the basic internal structure of the MPMC2 module and its major components:

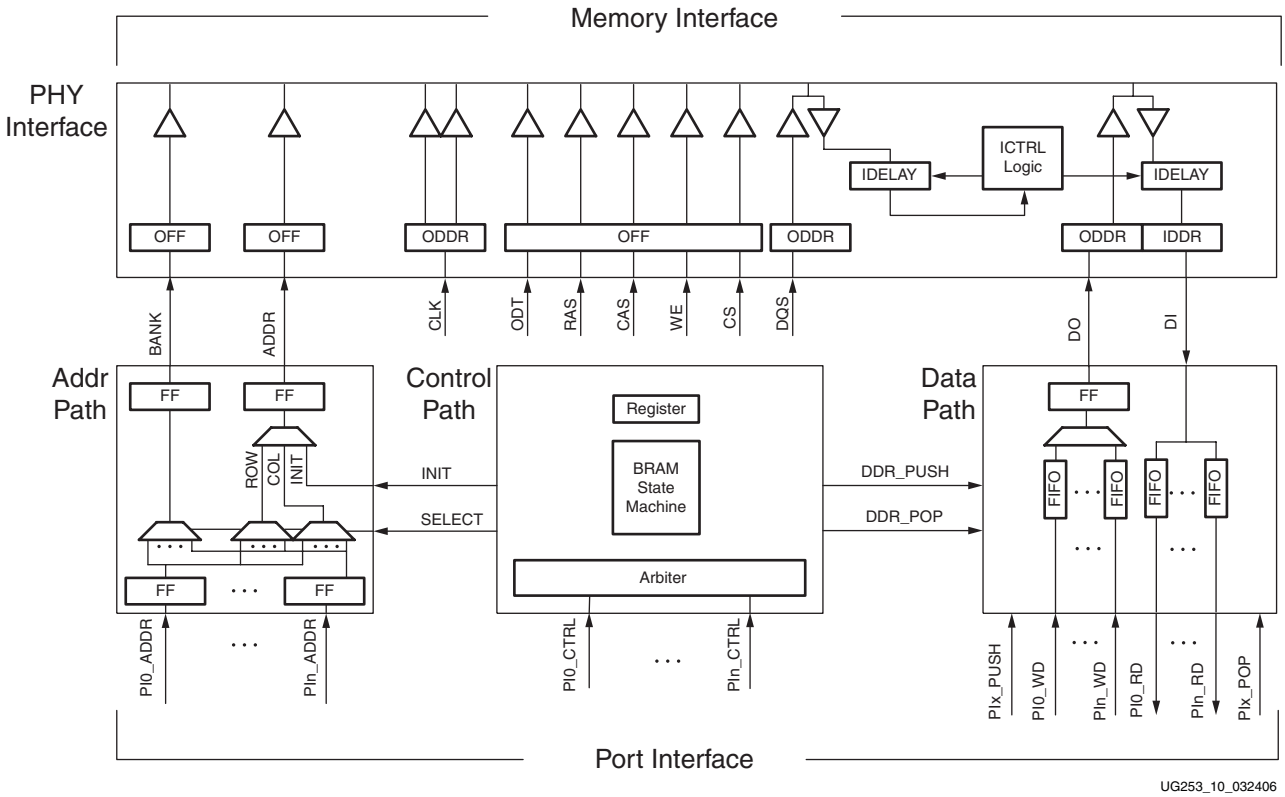


Figure 1-10: **Internal Architecture**

The following subsections describe the major MPMC2 components and additional features.

Address Path

The address path, shown in Figure 1-11, supports 1 to 8 ports of 32-bit addresses. This permits two processors to share memory as if each had separate memories that do not interact with each other. The address path also contains two parameterizable pipeline stages to accommodate for frequency trade-offs. Xilinx recommends keeping both address path pipeline registers. The input address register FIFO depth can be set to allow multiple transactions.

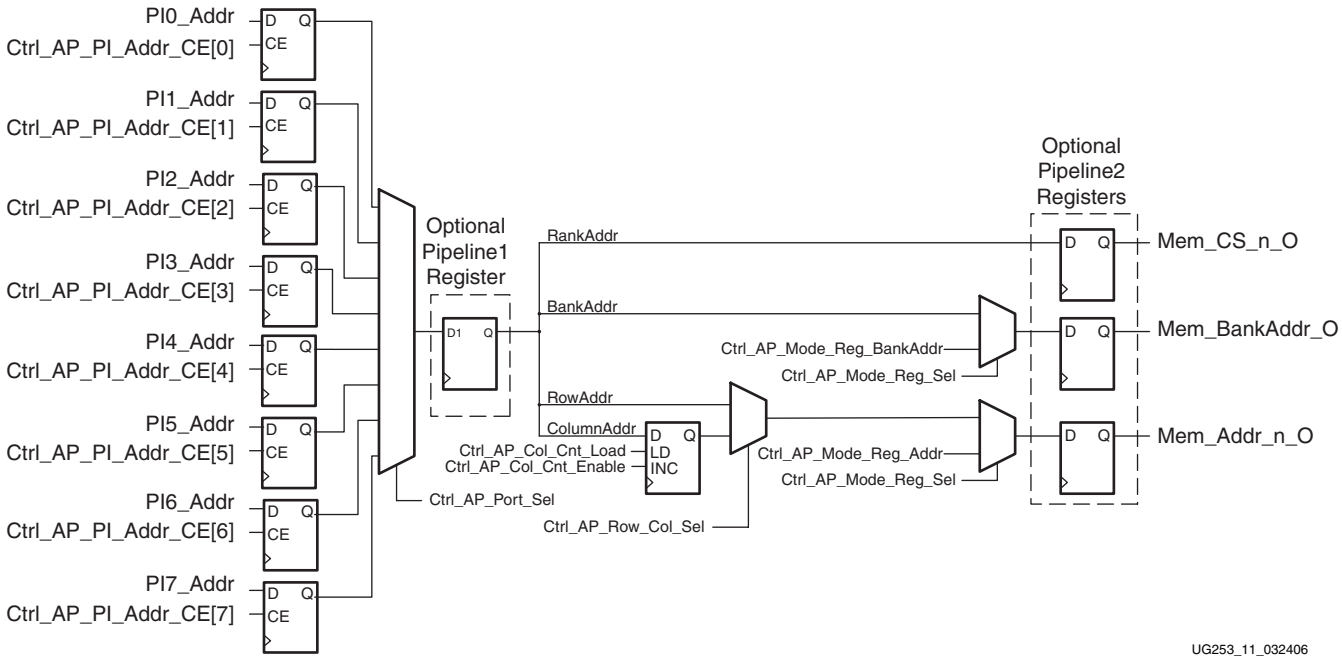


Figure 1-11: **Address Path Implementation**

Data Path

The data path supports memory that is 8, 16, 32, or 64 bits wide. Support for 8 and 16 bits is provided for those who want low-cost memory and high performance. When using 8-bit memories, the 64-word burst transfer type is *not* supported. This affects custom NPI interfaces. Additionally, because the PLB PIM uses this type for unaligned burst transfers, PLB master devices that request unaligned burst transfers are not supported with 8-bit memories.

You can configure each port to support FIFOs implemented in block RAM (BRAM) or 16-bit Shift Register Lookup table (SRL16) primitives.

Note: Either BRAM or SRL primitives can be used, but not both.

You can select the FIFO type, depending on resource availability, memory width, and required frequency of operation. Each port supports read and write data, the write data is not used in the ISPLB PIM. Two separate FIFOs are required, one for the read data and one for the write data. Each FIFO can be configured using the SRL16s or the BRAMs. The number of BRAMs required depends upon the width of the memory data. If the memory is 32-bit DDR, two BRAMs per port per read and write data are required to store the 64-bit data width. Similarly, with 64-bit DDR, four BRAMs are required per each direction of data per port. For 8-bit and 16-bit DDR memory, a 64-bit FIFO is required to match the NPI data width. Currently, 32-bit FIFOs (one BRAM) are not supported, even with 8-bit or 16-bit DDR memory.

These requirements are present because the memory cannot be *stalled*, thus the full width of the single-date rate data coming from the IDDR or going to the ODDR must be maintained. This means that four BRAMs are used in a 32-bit DDR system per port. In a 64-bit system each port can consume 8 BRAMs. See Table 1-1, page 26 for a table of acceptable uses of BRAM in MPMC2.

The output of the data path to the NPI PIM is always 64 bits. Write data from the NPI must be aligned to the size of the requested transfer. Byte enables should be set to demark data to be written.

Read data to the port interface is output target double-word first (word is defined as 32 bits of data and double-word is defined as 64 bits of data) for byte; half-word; word; 4-word, cache-line; and 8-word, cache-line operations and is *aligned* to the transfer size for 32-word burst or 64-word burst operations.

For cache-line transfers, the data path provides a read word address so that the port receiving the data can identify which word it is addressing. For word and burst transfers, the read word address should be ignored. The read FIFOs can be flushed by the port when the port has consumed all the data it needs, thus increasing port performance.

There is no underflow or overflow protection for the read and write data FIFOs. If you are designing custom NPI interfaces and custom PIMs, you must ensure that those PIMs *do not* cause a pop from an empty FIFO (underflow) or push into a full FIFO (overflow). Underflow or overflow of a data FIFO can cause data corruption.

Error Correcting Code (ECC)

MPMC2 supports Error Correcting Code (ECC). The ECC support logic resides between the data path and the physical interface blocks of the MPMC2. The ECC is detailed in Appendix B, "Error Correcting Code (ECC)."

BRAM Use Models

The data path to and from the PIMs is always 64 bits, regardless of the memory width. Figure 1-12 shows how and where the translations are accomplished. The FIFO in each port in both directions can be configured to use SRL16s or BRAMs. The memory data width affects how many BRAMs are required (two or four) and the required frequency for the design to work properly. Some combinations of BRAMs and SRLs are not allowed. Table 1-1, page 26 lists the allowable combinations.

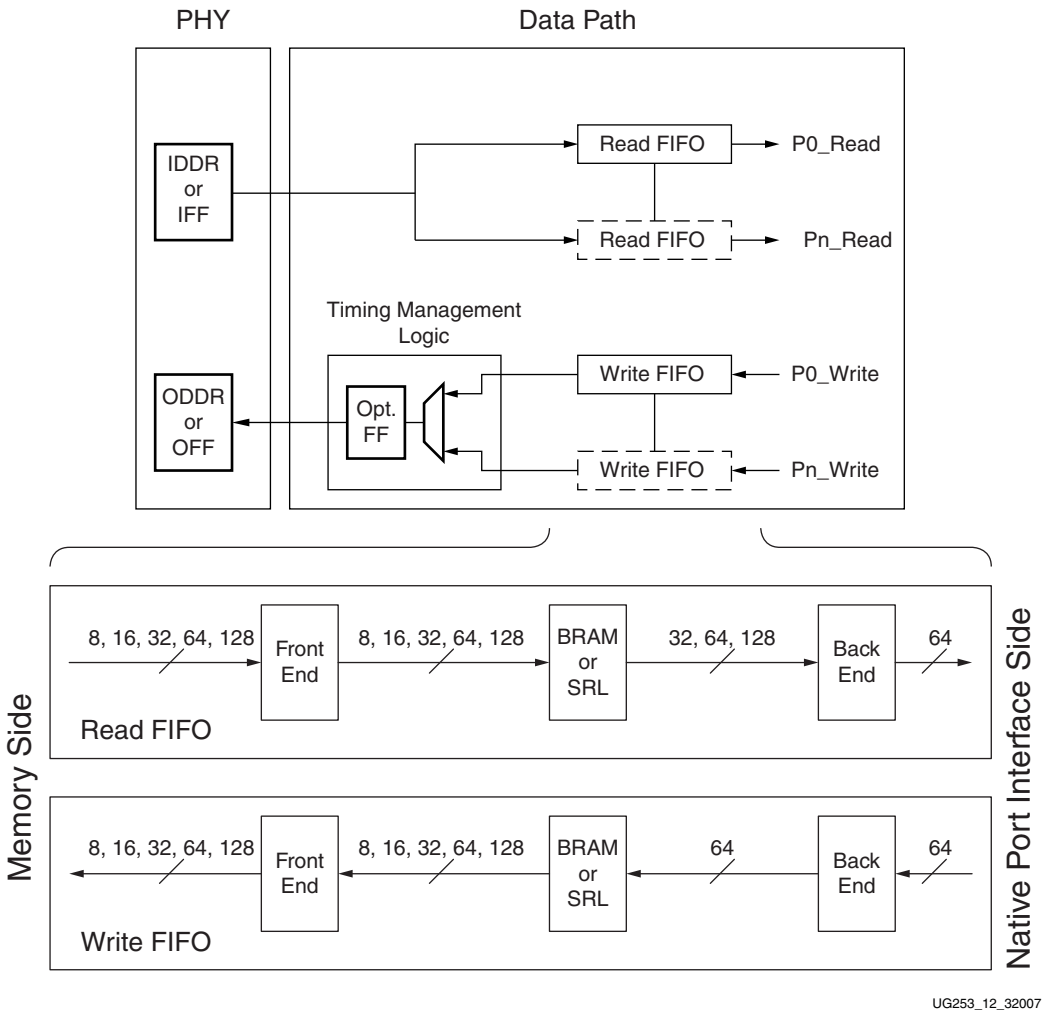


Figure 1-12: **Data Path Implementation**

Table 1-1 shows the allowable combinations for BRAMs and SRLs.

Table 1-1: **MPMC2 Usable Combinations for FIFO Style Per Port and Direction**

Memory			Port			FIFO Configuration	Elements per Read/Write FIFO(4)	Usable(5)
Data Width(1)	Clock Rate(2)	Organization(3)	Data Width	Clock Rate	Organization(3)			
8 bits	1X	8 bits X 128 entries	64 bits	1X	64 bits X 16 entries	SRL16 based	64	No
16 bits	1X	16 bits X 64 entries	64 bits	1X	64 bits X 16 entries	SRL16 based	64	Yes
32 bits	1X	32 bits X 32 entries	64 bits	1X	64 bits X 16 entries	SRL16 based	64	Yes
64 bits	1X	64 bits X 16 entries	64 bits	1X	64 bits X 16 entries	SRL16 based	64	Yes
128 bits	1X	128 bits X 16 entries	64 bits	1X	64 bits X 32 entries	SRL16 based	128	Yes
8 bits	1X	8 bits X 2048 entries	64 bits	2X	64 bits X 256 entries	BRAM based	1	No
16 bits	1X	16 bits X 1024 entries	64 bits	2X	64 bits X 256 entries	BRAM based	1	No
32 bits	1X	32 bits X 512 entries	64 bits	2X	64 bits X 256 entries	BRAM based	1	No
64 bits	2X	64 bits X 256 entries	64 bits	2X	64 bits X 256 entries	BRAM based	1	No
128 bits	N/A	N/A	64 bits	N/A	N/A	BRAM based	1	No
8 bits	1X	8 bits X 4096 entries	64 bits	1X	64 bits X 512 entries	BRAM based	2	No
16 bits	1X	16 bits X 2048 entries	64 bits	1X	64 bits X 512 entries	BRAM based	2	Yes
32 bits	1X	32 bits X 1024 entries	64 bits	1X	64 bits X 512 entries	BRAM based	2	Yes
64 bits	1X	64 bits X 512 entries	64 bits	1X	64 bits X 512 entries	BRAM based	2	Yes
128 bits	2X	128 bits X 256 entries	64 bits	2X	64 bits X 512 entries	BRAM based	2	No
8 bits	N/A	N/A	64 bits	N/A	N/A	BRAM based	4	No
16 bits	N/A	N/A	64 bits	N/A	N/A	BRAM based	4	No
32 bits	N/A	N/A	64 bits	N/A	N/A	BRAM based	4	No
64 bits	N/A	N/A	64 bits	N/A	N/A	BRAM based	4	No
128 bits	1X	128 bits X 512 entries	64 bits	1X	64 bits X 1024 entries	BRAM based	4	Yes

Notes:

1. **Memory Data Width** is the single-data rate version of data coming from the PHY interface. DDR memories have twice their native data width for this parameter. An 8-bit data width corresponds to a 4-bit wide DDR memory, which is not supported.
2. If the clock rate is 2x, `MPMC2_O_Clk0_2x` must be set to 2x, which is the clock rate of `MPMC2_O_Clk0`. Otherwise, `MPMC2_O_Clk0_2x` is unused. Currently, 2x clock rates are unsupported.
3. **Organization** is typically different between memory and port sides of the FIFOs, though the total number of bits in the FIFO is the same.
4. **Number of Elements** refers to the read or write FIFO only. Per-port element usage must add the configuration of both read and write FIFOs.
5. Usable configurations have been tested in hardware and are known to work.

Control Path

The control path logic, in combination with a unique physical (PHY) interface, is what allows the MPMC2 to control any form of memory. Figure 1-13 shows the main sections of the control path: an arbiter and a state machine.

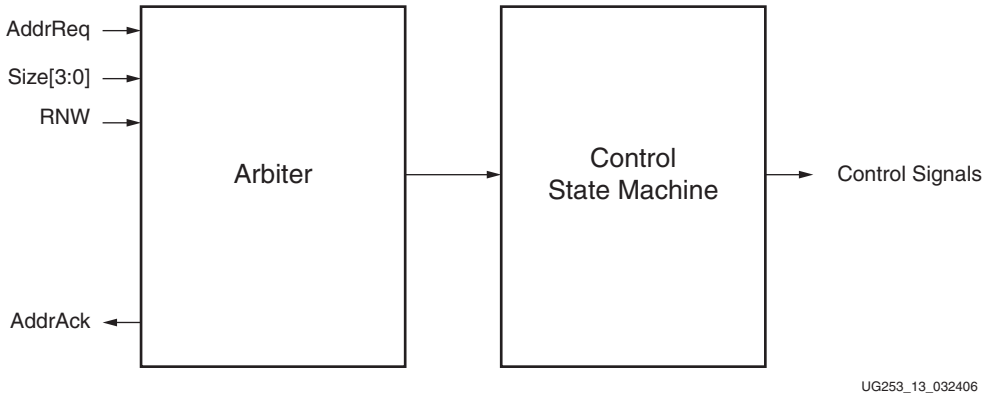


Figure 1-13: **Control Path Top-Level Diagram**

BRAM State Machine

The control path BRAM state machine, shown in Figure 1-14, can be set to any number of values, which permits any kind of control that you require. The state machine comprises several parts that control the data path, address path, PHY interface, and arbiter. The state machine consists of a BRAM (Ctrl BRAM) wherein each of its data outputs is attached to an SRL16. Each SRL16 has a delay parameter that can be set on a per-output basis. The delay parameter indicates the number of cycles that the BRAM data output should be delayed. If the parameter is set to zero, the SRL16 is bypassed. The address input to the BRAM changes the state of the state machine.

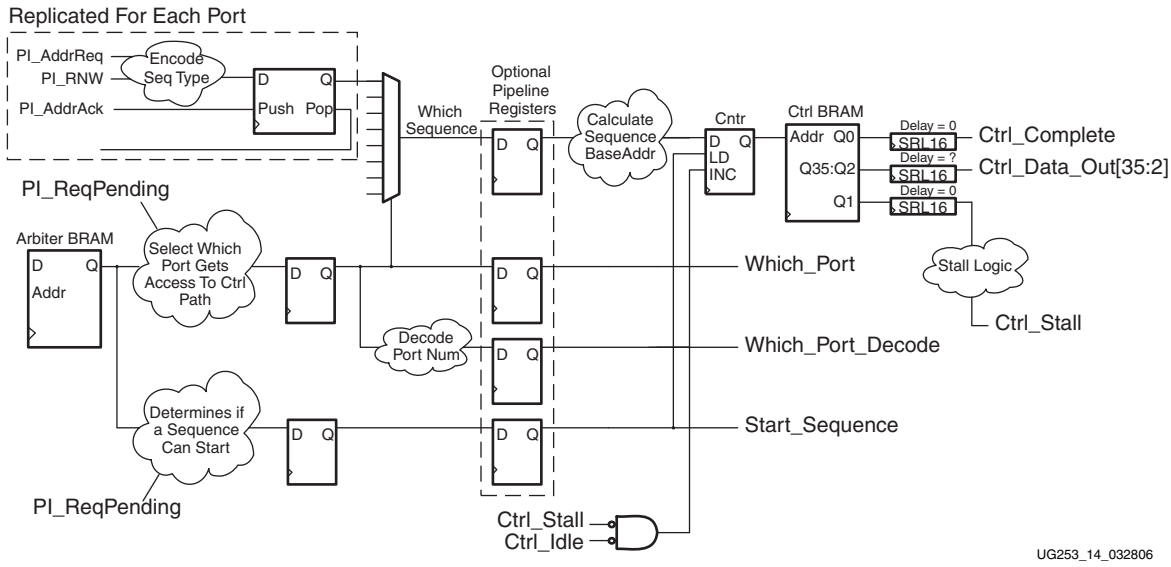


Figure 1-14: **Control Path BRAM State Machine**

The function of a BRAM state machine is to play sequences of events. The BRAM control path allows up to 16 sequences. MPMC2 has 12 sequences; the sequences are:

- Word write; word read
- 4-word, cache-line write; 4-word, cache-line read
- 8-word, cache-line write; 8-word, cache-line read
- 32-word burst write; 32-word burst read
- 64-word burst write; 64-word burst read
- No operation (NOP)
- Memory refresh

In DDR2 and DDR memories the read and write sequences can be divided into three stages: activate, read or write, and precharge. Each stage has specific sequences of events to the data path, address path, PHY interface, and arbiter. The sequences vary depending on the memory type (DDR2 or DDR), memory style (discrete parts, unbuffered DIMM, or registered DIMM), memory configuration (number of rank, bank, row, and column address bits), and clock frequency. The state machine can be configured to support any of these options because the contents of the BRAM can be modified and the SRL16 delays are configurable. The MPMC2 GUI creates `INIT` strings and delay values for the BRAM and SRLs when the pcore is generated.

Note: Changes to the clock frequency require generation of a new MPMC2 pcore.

For more information on creating and modifying MPMC2 configurations, see the *MPMC2 IP Configurator GUI User Guide.* The section "Additional Resources," page 13 contains a link to the document.

Figure 1-15, page 29 shows the memory organization of the control path BRAM state machine. The address must be generated (Figure 1-14, page 27) to *play* the sequence of events from the BRAM. The arbiter provides a prioritized list of port numbers that have access to the control path BRAM state machine.

The control path BRAM state machine contains logic that determines:

- If any of the ports are requesting access to the state machine.
- Which sequence the requesting ports are to play.
- The base address of the selected sequence.

After determining the per port state machine access, sequences, and base addresses, the base address is loaded into the counter that sets the control path BRAM state machine address. Every cycle, the address increments by one unless the sequence calls for a stall (`Ctrl_Stall`) or until the sequence has finished and the state machine is idle. Each sequence has a `Ctrl_Complete` bit that indicates a new sequence base address can be loaded and the next sequence can begin.

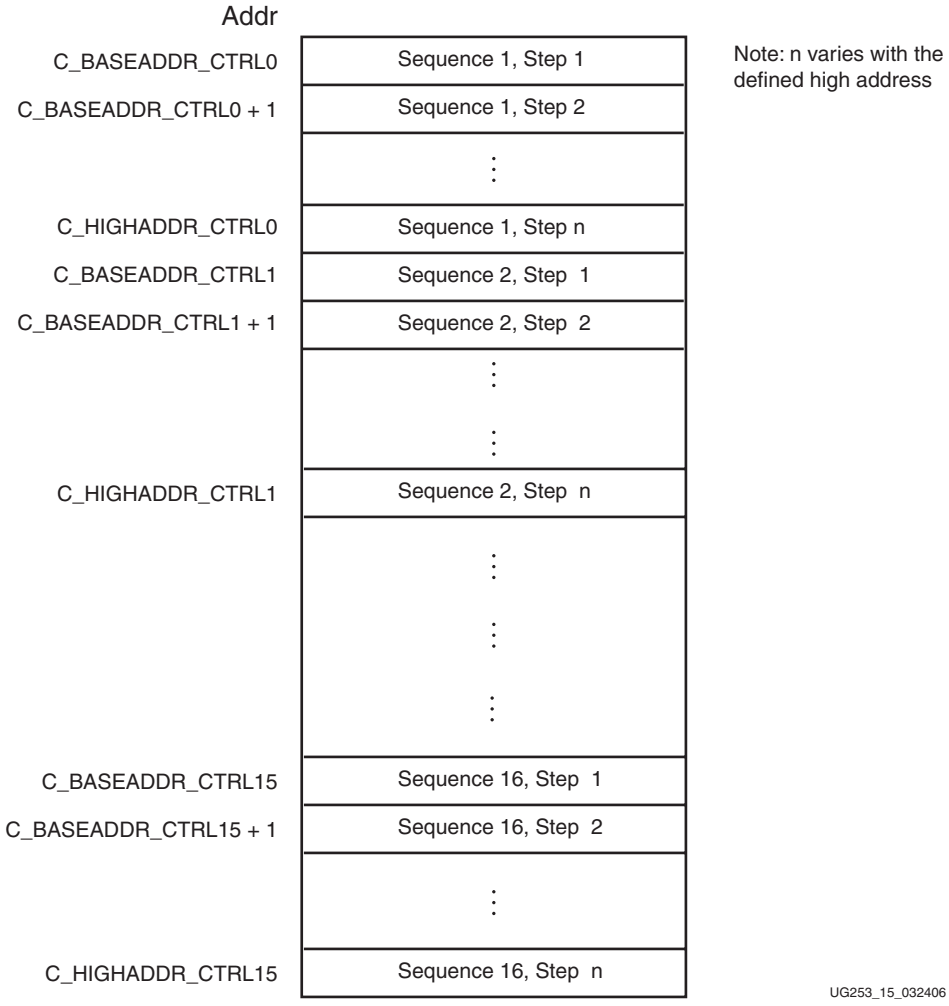


Figure 1-15: **Control Path BRAM State Machine Memory Organization**

Arbiter

The MPMC2 control path arbiter allows for complex arbitration algorithms. The arbitration algorithm is stored in a BRAM. The contents of the BRAM hold a set of encoded port numbers that represent the port priority for a given time slot. These can be configured using the MPMC2 IP Configurator GUI. The arbitration BRAM issues eight 3-bit values on every time slot in an 8-port implementation of the MPMC2.

DOA[0:2] represents the port with the highest priority:

- If this port is requesting access to the control path BRAM state machine, access is granted.
- If it is not requesting access, the port encoded by DOA[3:5] has the opportunity to receive access.

This process continues across all eight priority levels. If a port receives access to the control path BRAM state machine, the arbiter does not advance to the next time slot until `Ctrl_Complete` is asserted. If no port receives access to the state machine, the arbiter immediately advances to the next time slot.

Each algorithm has a base address and a high address. The algorithm number must be set to zero. The arbiter BRAM address increments:

- Every cycle that the `Ctrl_Complete` signal is received
- When the control path BRAM state machine is idle

When the address reaches the high address of the algorithm, it is reset to the base address.

The arbiter also sends address acknowledges to each port and monitors whether a port has a pending request as shown in Figure 1-16.

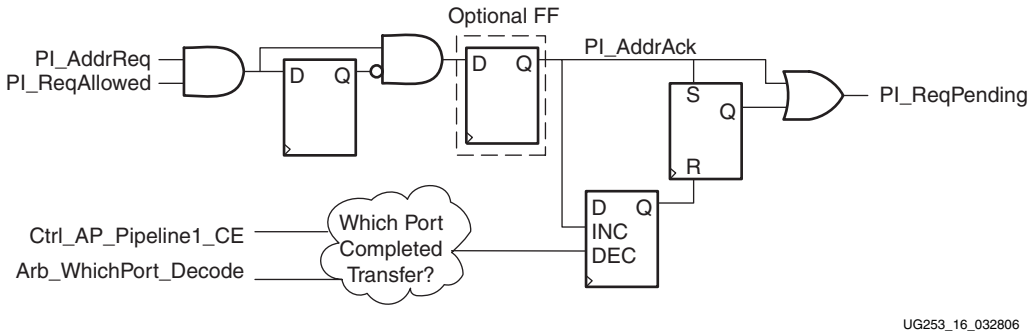


Figure 1-16: **Address Acknowledge Logic**

Physical (PHY) Interface

The MPMC2 PHY interface (Figure 1-10, page 22) is the interface between memory and the MPMC2 address path, control path, and data path. The PHY interface can be configured to support DDR and DDR2 SDRAM memories across Virtex-4, Virtex-5, Virtex-II Pro, and Spartan-3 platforms.

The interface between the MPMC2 data path and memory contains input and output DDR flip-flops, IDELAY logic (to delay the read data in Virtex-4 and Virtex-5 DDR2 and DDR systems only), and data capture logic. The output signals go through an output DDR flip-flop. The physical interface contains logic to perform memory initialization and timing calibration of the memory.

The IDELAY logic aligns the middle of the valid read data to the `CLK0` positive clock edge. This is necessary to accommodate for variations in trace delays on different boards. The IDELAY elements change the time that the read data arrives at the FPGA to align it to the main clock. At the end of the memory initialization sequence, the PHY path issues dummy write and read commands. The IDELAY logic determines the edges of the input data and shifts the input data signals to allow `CLK0` to properly capture the data. The implementation of the PHY interface automatically aligns the read data so that it is pushed into the read data path FIFOs at the appropriate time. The trace lengths for the data signals must be matched to the corresponding data strobe signal and proper FPGA I/O pin selections mush be made to ensure a robust interface.

The MPMC2 PHY interface is built using physical layer code from the Memory Interface Generator (MIG) design. The MIG design has an interface tool that helps guide you through the process of choosing the pinout and generating any necessary constraints files. The MIG design documentation also specifies pinout and board layout guidelines for DDR and DDR2 interfaces.

Note: MPMC2 designs require that you follow MIG pinout and layout guidelines. You must use the MIG Tool to generate pinout constraints for MPMC2 designs.

For Spartan-3 based designs, it is especially important to run the MIG tool to obtain the correct constraints file settings. The Spartan 3-based MIG design uses specially placed internal FPGA elements that must be correctly located in the FPGA.

For information about the MIG physical interface design, please register and download MIG. The section "Additional Resources," page 13 contain a link to the Xilinx Memory webpage where MIG can be found.

More information on the use of IDELAY in DDR and DDR2 applications can also be found in the *Memory Interfaces Data Capture Using Direct Clocking Technique* document. The section "Additional Resources," page 13 contains a link to this document.

IDELAY Controller

For Virtex-4 and Virtex-5 systems, instantiation of IDELAY controllers (IDELAYCTRL) is required to ensure that the IDELAY elements in the MPMC2 physical interface behave properly. The MPMC2 core has a parameter called `C_MPMC2_0_NUM_IDELAYCTRL` that determines the number of IDELAYCTRL elements to instantiate. By default, the MPMC2 instantiates one IDELAYCTRL which causes the ISE™ implementation tools to replicate the IDELAYCTRL blocks across the entire FPGA. The default setting is sufficient for systems where there is only one MPMC2 instance, and no other IP in the system requires the use of IDLELAY elements.

For systems where there are multiple IP cores each using their own IDELAYCTRL element independently, such as two instances of MPMC2 or MPMC2 and PCI in the same system, you must give special consideration to the number of IDELAYCTRLs.

- Instantiate the correct number of IDELAYCTRL blocks for each IP as determined by the clock regions where the associated IDELAY elements are used. For these systems, it is necessary to set the correct value for `C_MPMC2_0_NUM_IDELAYCTRL`.
- Ensure the IDELAYCTRL element is associated with the correct clock region positions in the FPGA and located in the User Constraints File (UCF).

Consult the ISE documentation for more information about IDELAYCTRL. The link to ISE documentation is available in "Additional Resources," page 13.

MPMC2 contains the following ports (see Table 1-3, page 59) that can be used to chain the IDELAY elements between IP blocks:

- `MPMC2_0_Idelayctrl_Rdy_I` is AND'd with internal IDELAYCTRL `RDY` signals inside MPMC2 and signifies that memory initialization can begin.
- `MPMC2_0_Idelayctrl_Rdy_O` port signals that the internal IDELAYCTRL `RDY` signals and the `MPMC2_0_Idelayctrl_Rdy_I` are all high.

This is useful in situations where two IP blocks have I / O that share a common IDELAYCTRL element. You can connect the `MPMC2_0_Idelayctrl_Rdy_O` to the `MPMC2_0_Idelayctrl_Rdy_I` of downstream MPMC2 IP or other IP blocks.

Additionally the `RDY` outputs of upstream IP blocks with IDELAYCTRL can be tied to the `MPMC2_0_Idelayctrl_Rdy_I` input.

Ensure that the `MPMC2_0_Idelayctrl_Rdy_I` and `MPMC2_0_Idelayctrl_Rdy_O` ports are not connected in a circular manner over one or more IP blocks.

`MPMC2_0_Idelayctrl_Rdy_I` and `MPMC2_0_Idelayctrl_Rdy_O` can be left unconnected when not needed. The EDK XPS tool automatically ties `MPMC2_0_Idelayctrl_Rdy_I` to high when it is unconnected.

Reset

The MPMC2 core and each MPMC2 PIM have a reset input. Internally these resets are all OR'ed together to create the master reset for the entire MPMC2 (including PIMs). It is not possible to reset an individual PIM or PORT of the MPMC2 without resetting everything.

The master reset is internally registered and synchronized before being distributed throughout MPMC2. The MPMC2 reset is therefore a fully synchronous reset.

Reset should be held for a minimum of eight cycles of the slowest PIM clock.

Transaction Ordering and Memory Coherency

In the MPMC2 architecture, transactions are executed to memory in the order the transactions are acknowledged with respect to a single port; consequently, on a single port, transactions are completed in the same order as requested.

Across multiple ports of MPMC2 there is no guarantee that the transactions issued by different ports will complete in the request order. You can modify the arbitration algorithms so that a given port is favored over another port. This can be used as a mechanism to influence transaction ordering but will not guarantee a specific order.

MPMC2 allows write transactions to be buffered inside the MPMC2. Because of the buffering, there is an undefined time between when a write transaction has completed over NPI and when the write completes to memory.

Because transaction ordering is not guaranteed across ports, a port doing a read from an address location being written to by another port might read the new or the old memory value. In some applications it is important to know that a write has completed to memory before issuing a read of that location.

There are three methods that can ensure coherency:

- The NPI interface can monitor the write FIFO empty flag. The empty flag is asserted when the write has completed to memory. The design can wait for the empty flag to go high before signaling that a read can be performed.
- The design can take advantage of the fact that transactions complete in order on a given port. After a write to a sensitive part of memory, the device can issue a dummy read and wait for the dummy read to complete and return data. The completion of the dummy read ensures that the previous write has completed to memory.
- The arbitration algorithm can be adjusted. If the port performing the writes can always be set to have higher priority than the ports doing the reads, this should also ensure that the write completes before the read across the two ports.

Note: Using any of these methods to ensure coherency may result in reduced system performance; employ these methods only when necessary.

The DSPLB and PLM PIMs follow the PLB CoreConnect method for handling memory coherency (for example, between PowerPC 405 processor instruction and Data PLB interface). The PLB `BUSY` signal is asserted on writes until the `Write_FIFO_Empty` flag is asserted indicating the write transaction has completed to memory. The processor normally ignores the `BUSY` flag unless an instruction is executed such as `Sync` or Enforce Instruction Execution In Order (`EIEIO`). When the `Sync` or `EIEIO` instruction is executed, the processor waits for PLB `BUSY` to be deasserted before issuing another transaction.

PIM Architecture

The Port Interface Module (PIM) architecture comprises the following interfaces:

- "Communication Direct Memory Access Controller (CDMAC)"
- "PowerPC 405 Processor Data-Side (DSPLB)"
- "PowerPC 405 Processor Instruction-Side Local Bus (ISPLB)"
- "Native Port Interface (NPI)"
- "On-Chip Peripheral Bus (OPB)"
- "Processor Local Bus (PLB)"
- "MicroBlaze Xilinx CacheLink (XCL)"

The following subsections describe the PIMs, followed by a section that contains the PIM signals and parameters.

Communication Direct Memory Access Controller (CDMAC)

The CDMAC PIM:

- Connects a communications direct memory access controller to the MPMC2 pcore. Each CDMAC PIM contains a 32-bit LocalLink interface that can be connected to the LL_xEMACs or customized.
- Provides highly intelligent DMA between communication devices
- Uses a DCR interface to set parameters, and reads all of its descriptors directly from memory with a minimum of intervention by the local processor(s)
- Supports full scatter-gather operations and other features useful to communication devices

Refer to Appendix A, "Communication Direct Memory Access Controller (CDMAC)," for details on the CDMAC functionality.

The following is a code snippet from the CDMAC MHS file.

```
PARAMETER C_CDMAC_3_BASEADDR = 0x00000000
PARAMETER C_CDMAC_3_HIGHADDR = 0x07FFFFFF
PARAMETER C_CDMAC_3_DCR_BASEADDR = 0b01_0100_0000
PARAMETER C_CDMAC_3_DCR_HIGHADDR = 0b01_0111_1111
PARAMETER C_CDMAC_3_MPMC2_TO_PI_CLK_RATIO = 2
BUS_INTERFACE CDMAC_3_S_DCR = dcr_v29_0
BUS_INTERFACE CDMAC_3_LLSRC = tft_local_link0
BUS_INTERFACE CDMAC_3_LLDST = color_local_link0
PORT CDMAC_3_Clk = CLK_100MHz
PORT CDMAC_3_Rst = sys_bus_reset_1 (optional, not required)
PORT CDMAC_3_CDMAC_INT = CDMAC_INT
PORT MPMC2_0_Clk0 = CLK_200MHz
```

CDMAC Clock Requirements

The CDMAC PIM must run at the same frequency as MPMC2 or half the frequency of MPMC2, as specified by `C_CDMAC_*_MPMC2_TO_PI_CLK_RATIO`.

PowerPC 405 Processor Data-Side (DSPLB)

The DSPLB PIM as shown in Figure 1-17:

- Supports PowerPC 405 processor Data-Side PLB requests:
  - one-word reads and writes
  - 4-word, cache-line reads
  - 8-word, cache-line reads and writes
- Supports request aborts
- Serves as an additional master bridge used to bridge the DSPLB to the master ports of the OPB or PLB PIMs
- Connects the PowerPC 405 processor data-side PLB interface directly to the MPMC2 pcore via the `plb_mls1` module (described in the MPMC2 reference designs). The `plb_mls1` pcore is an optimized one master and one slave PLB "arbiter" designed for point-to-point connections between a single master and slave.
- Is a dedicated MPMC2 port that avoids the cost of arbitration present with the typical PLB

The DSPLB PIM provides higher performance because it is designed specifically for DSPLB behavior as defined in the *PowerPC 405 Processor Block Reference Manual*. The section "Additional Resources," page 13 contains a link to the document.

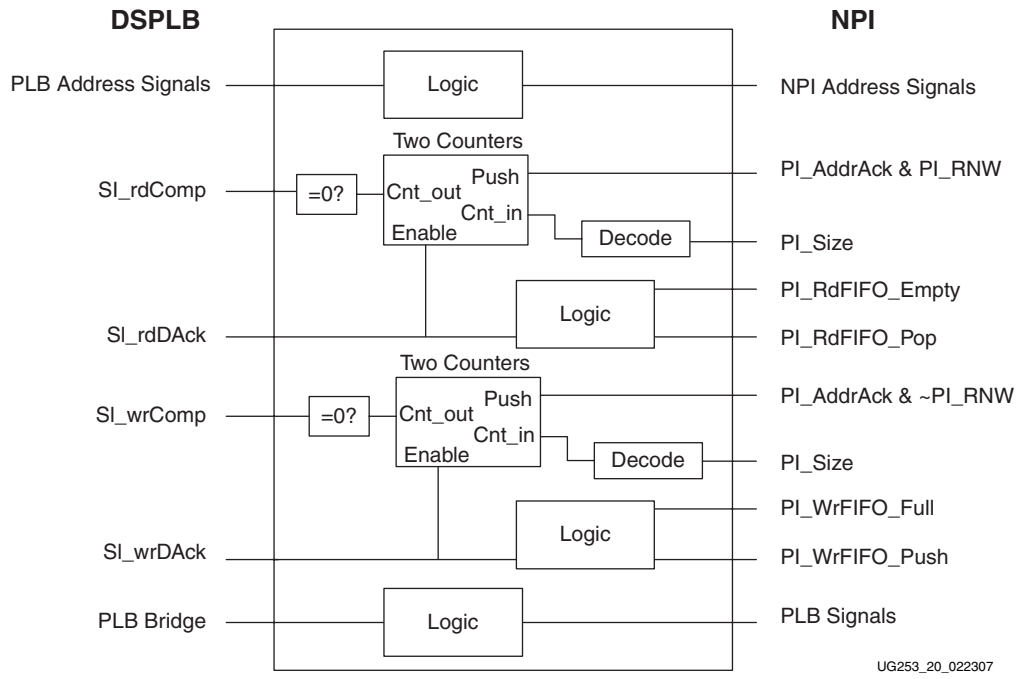


Figure 1-17: **PowerPC 405 DSPLB Block Diagram**

This PIM contains the logic to translate all data-side requests into MPMC2 requests. (The DSPLB PIM has a direct point-to-point connection between DSPLB (or ISPLB) and the MPMC2 port interface.) Therefore, when using this PIM, no other masters or slaves may be present.

The DSPLB can have up to two outstanding read requests and two write requests. The PIM responds to read requests even when the MPMC2 is busy and cannot accept the request. However, read data will not arrive until the request is handled by the MPMC2.

Additionally, the DSPLB PIM acknowledges all address requests, making the assumption that all requests are to valid address spaces.

Write requests are immediately acknowledged. MPMC2 sends a `Complete` signal immediately after pushing the write data. The `BUSY` flag is released once the writer FIFOs become empty, signifying that the write has gone to memory.

DSPLB Bridging Functionality

The DSPLB PIM provides master bridge logic to connect the DSPLB to more than just memory. This added functionality is achieved with minimal additional logic and without the performance penalty of using a full PLB core and arbiter to connect to the MPMC2 and other PLB slaves.

The DSPLB PIM can bridge the DSPLB to other buses using the PLB signals, allowing bus requests to go to either the MPMC2 or another peripheral. The PLB signals are wired out to a special bridge interface similar to PLB. Other MPMC2 PIMs (PLB and OPB PIMs) have slave bridge ports and master bus ports to bring these signals out to a separate bus. You can also connect the DSPLB master bridge port to a second MPMC2 controller.

You can access OPB peripherals by the DSPLB in the following sequence:

1. Through the DSPLB PIM master bridge output port
2. To the OPB PIM MPMC2 bridge and master ports
3. Out to the OPB bus

When the DSPLB bridge is in use, the `C_DSPLB_<portnum>_RNG0_BASEADDR` and `C_DSPLB_<portnum>_RNG0_HIGHADDR` range designates the memory range of the MPMC2. Likewise, the `C_DSPLB_<portnum>_RNG1_BASEADDR` and `C_DSPLB_<portnum>_RNG1_HIGHADDR` range designate the memory range of the first master port of a PLB or OPB PIM. Further, `RNG2` specifies second PLB or OPB PIM, and so on.

Note: If the processor attempts to access a memory address not covered by any C_DSPLB_<*portnum*>_RNG1_<*BASE/HIGH*>_ADDR parameter, the processor will hang waiting for a response. Consider setting address range values that cover all address space if this is of concern.

The following is a code snippet from the DSPLB MHS file.

```
PARAMETER C_DSPLB_1_RNG0_BASEADDR = 0x0000_0000
PARAMETER C_DSPLB_1_RNG0_HIGHADDR = 0x03ff_ffff
PARAMETER C_DSPLB_1_RNG1_BASEADDR = 0x7000_0000
PARAMETER C_DSPLB_1_RNG1_HIGHADDR = 0x7fff_ffff
PARAMETER C_DSPLB_1_RNG2_BASEADDR = 0x8000_0000
PARAMETER C_DSPLB_1_RNG2_HIGHADDR = 0xffff_ffff
PARAMETER C_DSPLB_1_MPMC2_TO_PI_CLK_RATIO = 1
BUS_INTERFACE DSPLB_1 = plb_1
PORT MPMC2_0_Clk0 = CLK_100MHz
```

You can include a maximum of two DSPLB PIMs in an MPMC2. You can also connect a PowerPC 405 ISPLB port to an MPMC2 DSPLB PIM to allow the ISPLB port on the PowerPC 405 processor to access the master bridge to OPB or PLB.

DSPLB Clock Requirements

The DSPLB PIM must run at the same or half the frequency as MPMC2, as specified by `C_DSPLB_*_MPMC2_TO_PI_CLK_RATIO`.

When using two DSPLB PIMs, both DSPLB PIMs must run at the same frequency.

PowerPC 405 Processor Instruction-Side Local Bus (ISPLB)

The ISPLB PIM:

- Supports PowerPC 405 instruction-side PLB requests:
  - one-word reads
  - 4-word, cache-line reads
  - 8-word, cache-line reads
- Supports request aborts

The ISPLB PIM, shown in Figure 1-18, connects the PowerPC 405 instruction-side PLB interface directly to the MPMC2 pcore via the `plb_m1s1` module (described in the MPMC2 reference designs). The `plb_mls1` pcore is an optimized one master and one slave PLB "arbiter" designed for point-to-point connections between a single master and slave.

The ISPLB port is a dedicated MPMC2 port that avoids the cost of arbitration present with the typical PLB. The ISPLB PIM provides higher performance than others because it is designed specifically for ISPLB behavior as defined in the *PowerPC 405 Processor Block Reference Manual*. The section, "Additional Resources," page 13, contains a link to the document.

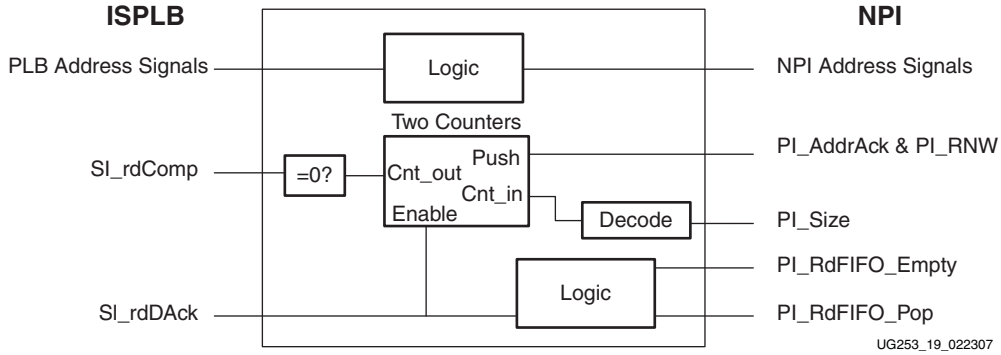


Figure 1-18: **PowerPC 405 ISPLB Block Diagram**

This PIM contains the logic to translate all instruction-side requests into MPMC2 requests. (The ISPLB PIM provides a direct point-to-point connection between the ISPLB and the MPMC2 port interface.) Therefore, when using this PIM no other masters or slaves can be present; this PIM must be the only slave on the PLB bus.

The ISPLB can have up to two outstanding read requests. The PIM responds to these requests even when MPMC2 is busy and cannot accept the request. However, the data does not arrive until the request is handled by the MPMC2. Additionally, the ISPLB PIM acknowledges all address requests, making the assumption that all requests are to valid address spaces.

Note: Unlike the DSPLB, the ISPLB cannot bridge to other ports. If your instruction code is not in OCM or external DDR/DDR2 memory, you might need to connect the PowerPC 405 ISPLB port to a DSPLB PIM (MPMC can have at most two DSPLB PIMs). Alternatively, you can connect the PowerPC 405 ISPLB port to a regular PLB bus that has PLB BRAM and a PLB PIM.

The following is a code snippet from the ISPLB MHS file. The first two parameters are not currently used by the ISPLB PIM logic, but are present due to tool requirements.

```
PARAMETER C_ISPLB_0_BASEADDR = 0x0000_0000
PARAMETER C_ISPLB_0_HIGHADDR = 0x03ff_ffff
PARAMETER C_ISPLB_0_MPMC2_TO_PI_CLK_RATIO = 1
BUS_INTERFACE ISPLB_0 = plb_0
PORT MPMC2_0_Clk0 = CLK_100MHz
```

PowerPC 405 ISPLB Clock Requirements

The ISPLB PIM must run at the same or half the frequency as MPMC2, as specified by `C_ISPLB_*_MPMC2_TO_PI_CLK_RATIO`.

Native Port Interface (NPI)

The NPI PIM:

- Allows you to extend the capabilities of MPMC2 to meet you own design needs.
- Offers a simple interface to memory that can easily be adapted to nearly any protocol.
- Provides address, data, and control signals to enable read and write requests for memory.
- Supports byte; half-word; word; double-word; 4-word; cache-line (four 32-bit words); 8-word, cache-line (eight 32-bit words); 32-word bursts (thirty-two 32-bit words); and 64-word burst (sixty-four 32-bit words) operations.

The interface permits simultaneous push and pull of data from the port FIFOs. I/O Signals and Parameters are specified in Table 1-16, page 71 and Table 1-17, page 73.

The following is a code snippet from the NPI MHS file.

```
PARAMETER C_MPMC2_PIM_0_BASEADDR = 0x0000_0000
PARAMETER C_MPMC2_PIM_0_HIGHADDR = 0x03ff_ffff
BUS_INTERFACE MPMC2_PIM_0 = npi_0
```

MPMC2 has a port called `MPMC2_PIM_<Port_Number>_RdFIFO_Latency`. PIMs use this port to determine when `<PIM_Name>_<Port_Number>_RdFIFO_Data` and `<PIM_Name>_<Port_Number>_RdFIFO_RdWdAddr` are valid in relationship to `<PIM_Name>_<Port_Number>_RdFIFO_Pop`.

With the NPI, the read FIFO signals are brought out to ports, and the peripheral attached to the NPI must accommodate the read FIFO latency. The value is provided as a port.

`MPMC2_PIM_<Port_Number>_RdFIFO_Latency` value behaves as follows:

- When set to 0: Data and `RdWdAddr` are valid in the same cycle as the `Pop` signal.
- When set to 1: Data and `RdWdAddr` are valid in the cycle following the `Pop` signal.
- When set to 2: Data and `RdWdAddr` are valid in two cycles following the `Pop` signal.

All other values are currently invalid and unsupported. This setting does not affect the timing of the `RdFIFO_Empty` signal.

The value of this parameter is set by the pipeline settings and MPMC2 configuration. It cannot be explicitly set nor manually adjusted.

When writing data using the NPI, you mush adhere to the following rules:

- All write data must be pushed into the write FIFOs before it is required by the memory.
  - For safest operation, assert the address request after all data has been pushed into the write FIFOs, as shown in Figure 1-31, page 54. An exception exists for word writes when using a 64-bit memory.
  - For improved system performance, MPMC2 can be analyzed to detect the earliest clock cycle in which the address request can be asserted, in relation to the last write data push. This analysis is best performed in simulation. The key variables in this analysis are the:
    - Number of pipeline stages
    - Width of memory
    - Ability of custom PIMs to stall data

- When using a double-word write transfer with a 64-bit memory:
  - The write data push must occur a minimum of one cycle after the address acknowledge.
  - All write data pushes for previous transfers must be completed before asserting the address request.

A status signal called `MPMC2_PIM_<Port_Number>_InitDone` goes high after the memory initialization is complete. Do not push or pop data from FIFOs when initialization is not complete. FIFO flags are undefined and might contain invalid values during initialization. The NPI PIM can assert `AddrReq` during initialization but the MPMC2 will not assert `AddrAck` until initialization is complete.

Clock Requirements

The NPI PIM must run at the same frequency as MPMC2. Support for any other frequency must be implemented in your custom PIM.

Native Port Interface Timing Diagrams

The following timing diagrams illustrate the functionality of the port interfaces. In the actual design signal names are prefixed with *MPMC2_PIM_* and the port number, but have been omitted in this timing diagram section for readability. The size signals determine the type of transfer to be performed; the types of transfers are decoded in Table 1-2.

Table 1-2: **Size Values Decoded**

Value	Transaction Type
0	Double-word (1)
1	4-word, cache-line (2)
2	8-word, cache-line (3)
3	Reserved
4	32-word burst (4)
5	64-word burst (5)
6–15	Reserved

Footnotes:

1. Address is double-word aligned. The three least significant address bits are ignored.
2. Reads: Address is double-word aligned. The three least significant address bits are ignored.
   Writes: Address is 4-word aligned. The 4 least significant address bits are ignored.
3. Reads: Address in double-word aligned. The three least significant address bits are ignored.
   Writes: Address is 8-word aligned. The five least significant address bits are ignored.
4. Address is 32-word aligned. The seven least significant address bits are ignored.
5. Address is 64-word aligned. The 8 least significant address bits are ignored.

Note: 64-word burst transfers are not supported with 8-bit wide DDR or DDR2 memories.

Double-word Read Transfer

Figure 1-19 shows the timing diagram for a double-word read transfer where C_MPMC2_0_RD_FIFO_LATENCY equals zero. Because the MPMC2 data is aligned to the size of RdFIFO_Data, this figure also applies to byte, half-word, and word transfers.

To initiate a transfer:

- Assert AddrReq with RNW and Addr .
- Align Addr on a double-word boundary.

The MPMC2 then asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another word request is desired.

After AddrAck has been asserted:

- RdFIFO_Empty will be deasserted, indicating that data is valid on RdFIFO_Data and RdWdAddr and can be popped out of the read FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

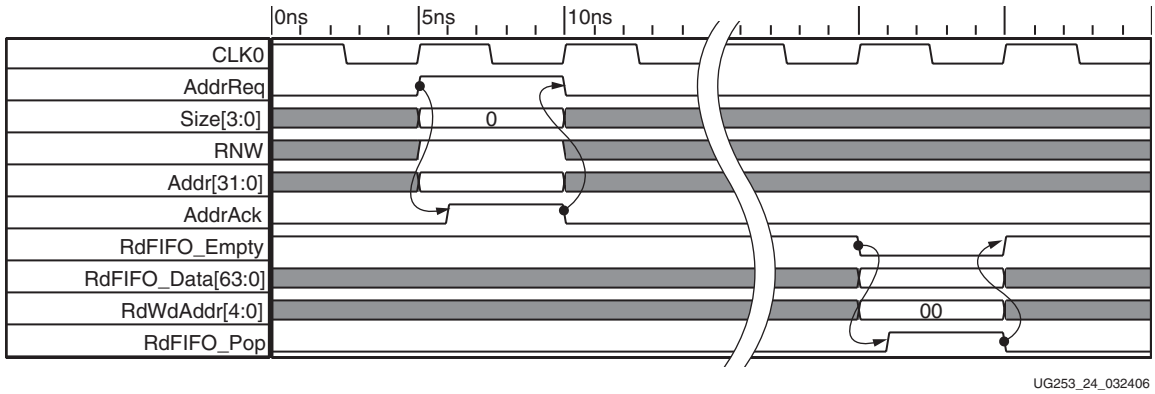


Figure 1-19: **Double-word Read**

4-Word, Cache-Line Read Transfer

Figure 1-20 shows the timing diagram for a 4-word, cache-line read transfer where C_MPMC2_0_RD_FIFO_LATENCY equals zero.

To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr on a double-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck is asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another request is required.

After AddrAck has been asserted:

- RdFIFO_Empty will be deasserted, indicating that data is valid on RdFIFO_Data and RdWdAddr and can be popped out of the read FIFOs.
- RdWdAddr indicates which word is being popped out of the FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

After all data has been popped out of the FIFOs, RdFIFO_Empty is asserted.

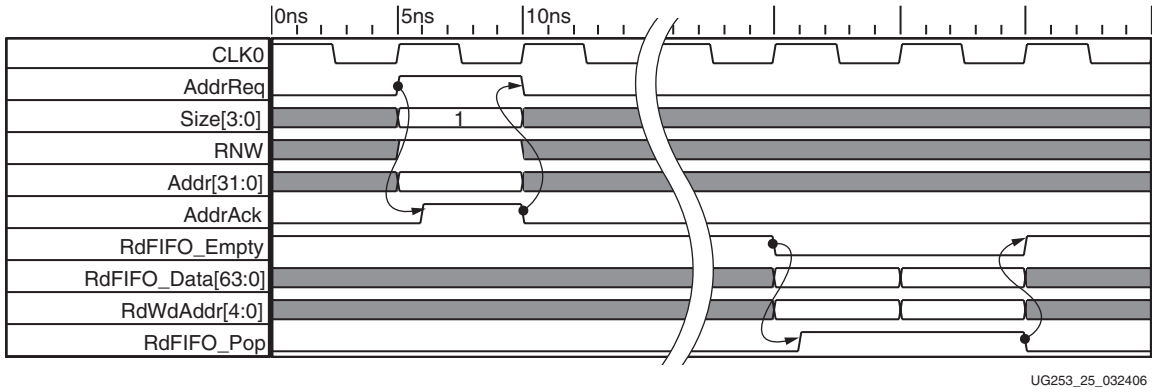


Figure 1-20: **4-Word Cache-Line Read**

Eight-Word, Cache-Line, Read Transfer

Figure 1-21 shows the timing diagram for an 8-word, cache-line read transfer where C_MPMC2_0_RD_FIFO_LATENCY equals zero.

To initiate a transfer:

- Assert AddrReq with RNW and Addr .
- Align Addr on a double-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another request is required.

After AddrAck has been asserted:

- RdFIFO_Empty will be deasserted, indicating that data is valid on RdFIFO_Data and RdWdAddr and can be popped out of the read FIFOs.
- RdWdAddr indicates which word is being popped out of the FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

After all data has been popped out of the FIFOs, RdFIFO_Empty is asserted.

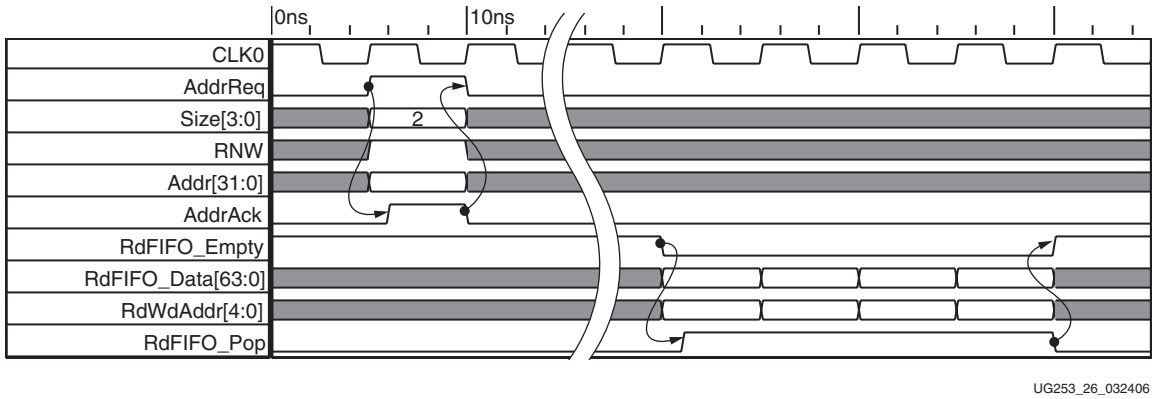


Figure 1-21: **8-Word Cache-Line Read**

32-Word, Burst Read Transfer

Figure 1-22 shows the timing diagram for a 32-word, burst read transfer where C_MPMC2_0_RD_FIFO_LATENCY equals zero.

To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr on a 32-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another request is required.

After AddrAck has been asserted:

- RdFIFO_Empty is deasserted, indicating that data is valid on RdFIFO_Data and can be popped out of the read FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

After all data has been popped out of the FIFOs, RdFIFO_Empty is asserted.

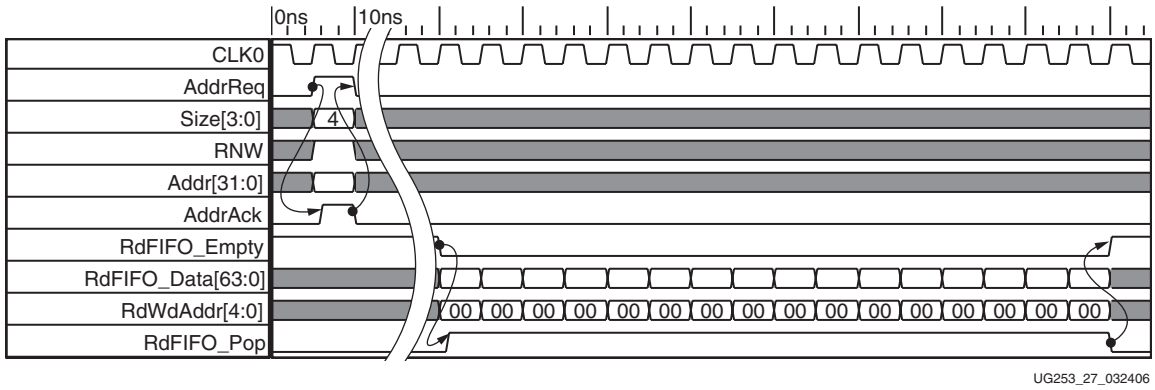


Figure 1-22: **32-Word Burst Read**

4-Word, Cache-Line, Back-to-Back Read Transfer

Figure 1-23 shows the timing diagram for two 4-word, cache-line read transfers that are requested back-to-back, where C_MPMC2_0_RD_FIFO_LATENCY equals zero.

To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr on a double-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq remains asserted.

In the example, the second address acknowledge is delayed by one cycle, but this delay is typically longer.

After the first AddrAck has been asserted:

- RdFIFO_Empty will be deasserted, indicating that data is valid on RdFIFO_Data and RdWdAddr and can be popped out of the read FIFOs.
- RdWdAddr indicates which word of the corresponding transfer is being popped out of the FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

In this example, there is a one cycle delay between data being available from the first transfer to data being available for the second transfer. This delay can be shorter or longer.

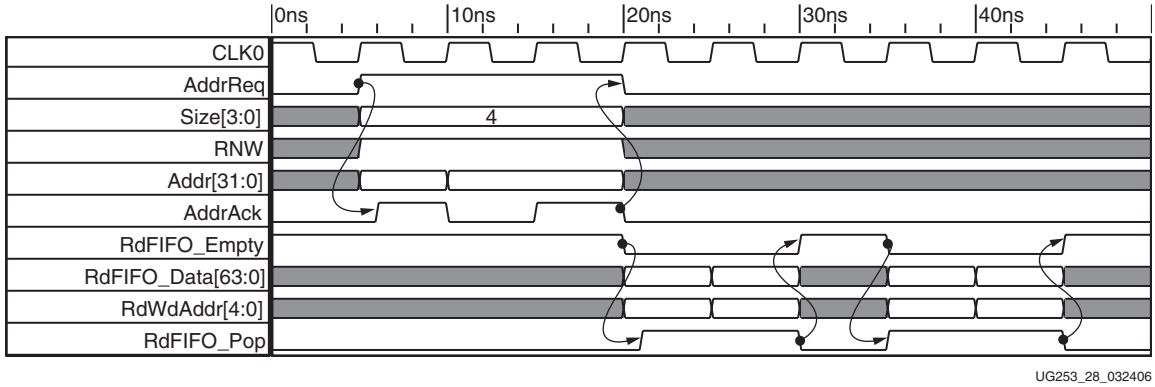


Figure 1-23: **Back-to-Back Read**

4-Word, Cache-Line Read Transfer with Read FIFO Reset

Figure 1-24 shows the timing diagram for a 4-word, cache-line read transfer where the PIM issues a read FIFO reset after the first 64-bits are popped out of the FIFOs, where C_MPMC2_0_RD_FIFO_LATENCY equals zero.

To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr on a double-word boundary.

Size determines the size of the transfer. The MPMC2 will assert AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless there is another request.

After AddrAck has been asserted:

- RdFIFO_Empty is deasserted, indicating that data is valid on RdFIFO_Data and RdWdAddr and can be popped out of the read FIFOs.
- RdWdAddr will indicate which word of is being popped out of the FIFOs.
- RdFIFO_Pop can be asserted any time that RdFIFO_Empty is deasserted.

In this case, the port interface only requires 64-bits of data, so the port interface asserts RdFIFO_Flush with the RdFIFO_Pop signal. This empties the read FIFOs and RdFIFO_Empty returns to being asserted.

If a larger transfer is reset while the data is still being pushed from memory into the read FIFOs, the MPMC2 will drop this data and ensure that it does not go into the read FIFOs.

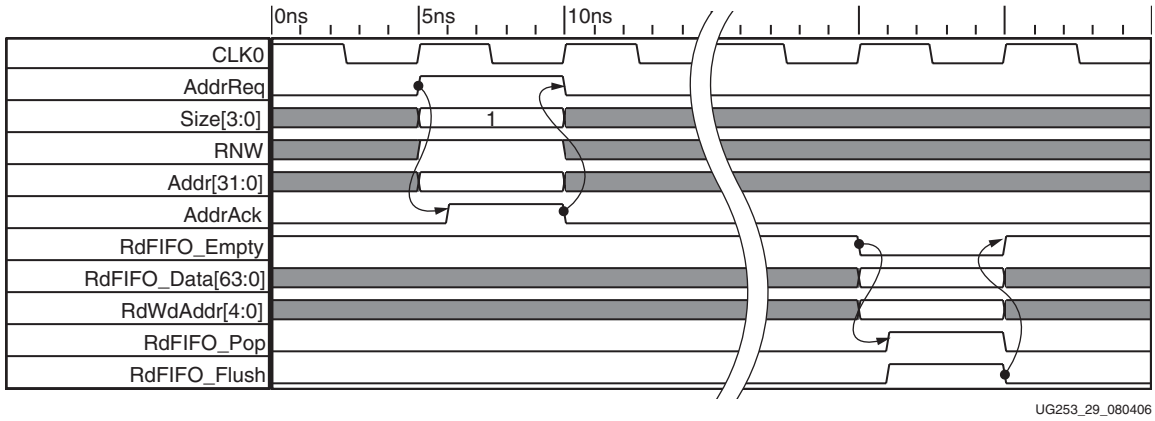


Figure 1-24: **Read FIFO Reset**

Double-Word Write Transfer

Figure 1-25 shows the timing diagram for a double-word write transfer. As the MPMC2 data is aligned to the size of `WrFIFO_Data`, this figure also applies to byte, half-word, and word (32-bit) transfers.

To initiate a transfer:

- Assert `AddrReq` with `RNW` and `Addr`.
- Align `Addr` on a double-word.

Size determines the size of the transfer. The MPMC2 asserts `AddrAck` on the same cycle or on one of the following cycles.

After `AddrAck` has been asserted for one clock cycle, `AddrReq` must be deasserted on the following cycle unless another request is desired.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- For a byte, half-word, word, or double-word write, the data must be aligned to 64-bit boundaries and the byte enables must be set to indicate where the data is located within the 64-bit data.
- `WrFIFO_Data` and `WrFIFO_BE` must be valid while `WrFIFO_Push` is asserted.
- `WrFIFO_Push` is not allowed to be asserted the clock cycle following a cycle where `WrFIFO_AlmostFull` is asserted.
- Refer to the bulleted information, Refer to the bulleted information, "When using a double-word write transfer with a 64-bit memory: on page 41" in "Native Port Interface (NPI)" regarding special cases for 64-bit memories.

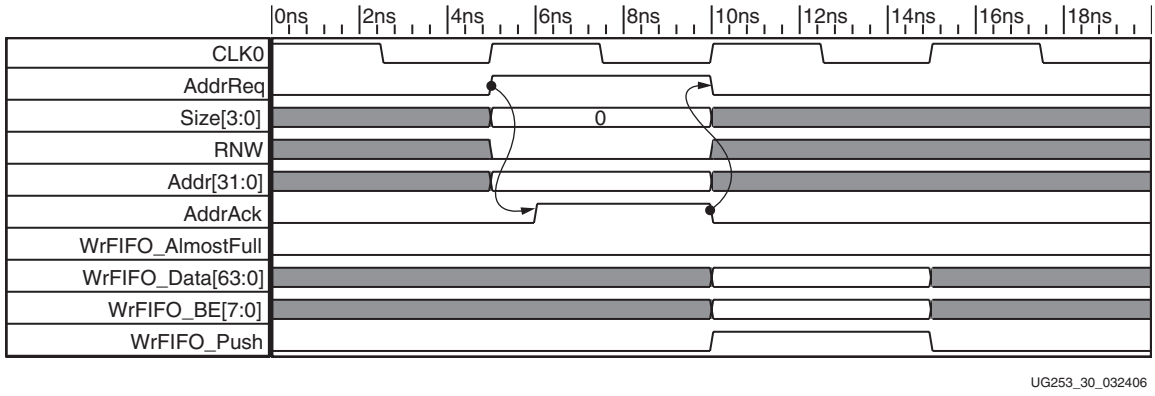


Figure 1-25: **Double-word Write**

4-Word, Cache-Line Write Transfer

Figure 1-26 shows the timing diagram for a 4-word, cache-line write transfer.

To initiate a transfer:

- Assert `AddrReq` with `RNW` and `Addr`.
- Align `Addr` on a 4-word boundary.

Size determines the size of the transfer. The MPMC2 asserts `AddrAck` on the same cycle or on one of the following cycles.

After `AddrAck` has been asserted for one clock cycle, `AddrReq` must be deasserted on the following cycle unless there is another request.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- `WrFIFO_Data` and `WrFIFO_BE` must be valid while `WrFIFO_Push` is asserted.
- `WrFIFO_Push` is not allowed to be asserted the clock cycle following a cycle where `WrFIFO_AlmostFull` is asserted.

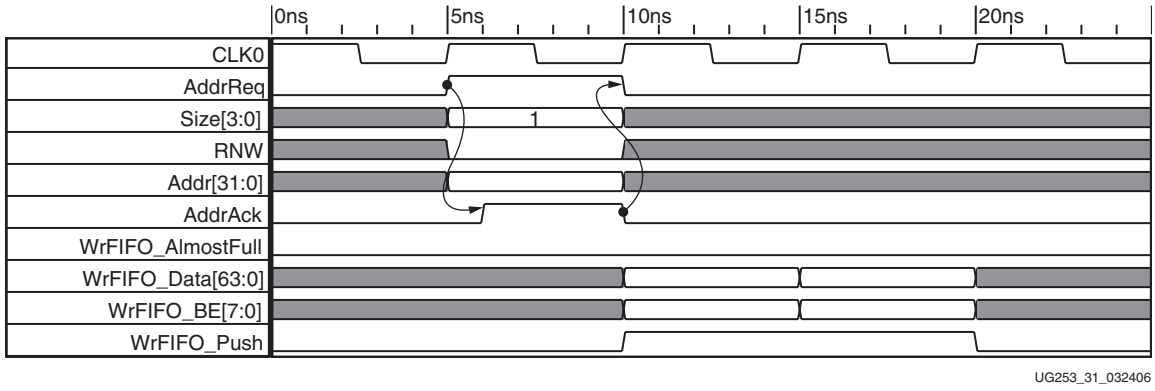


Figure 1-26: **4-Word, Cache-Line Write**

8-Word, Cache-Line Write Transfer

Figure 1-27 shows the timing diagram for an 8-word, cache-line write transfer.

To initiate a transfer:

- `AddrReq` is asserted with `RNW` and `Addr`.
- `Addr` must be aligned on an 8-word boundary.

Size determines the size of the transfer. The MPMC2 asserts `AddrAck` on the same cycle or on one of the following cycles.

After `AddrAck` has been asserted for one clock cycle, `AddrReq` must be deasserted on the following cycle unless another request is desired.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- `WrFIFO_Data` and `WrFIFO_BE` must be valid while `WrFIFO_Push` is asserted.
- `WrFIFO_Push` is not allowed to be asserted the clock cycle following a cycle where `WrFIFO_AlmostFull` is asserted.

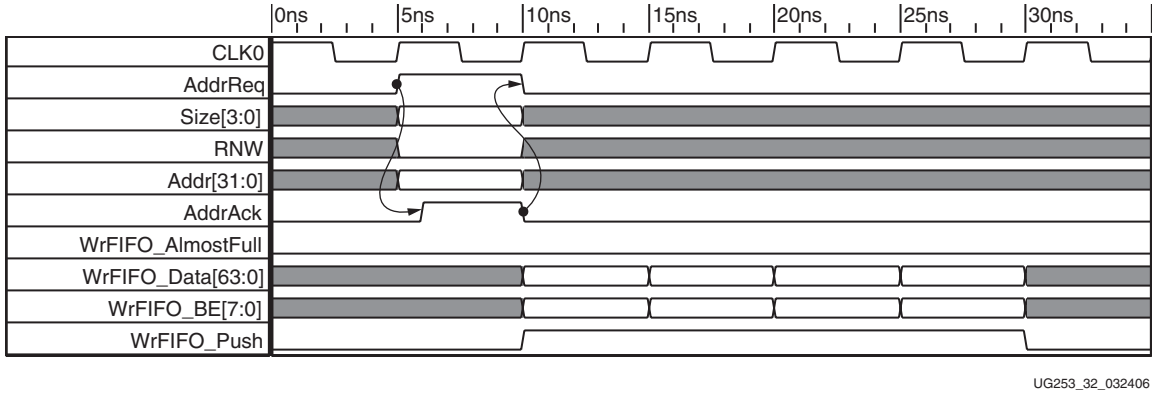


Figure 1-27: **8-Word, Cache-Line Write**

32-Word, Burst Write Transfer

Figure 1-28 shows the timing diagram for a 32-word, burst write transfer.

To initiate a transfer:

- Assert the `AddrReq` with `RNW` and `Addr`.
- Align `Addr` to a 32-word boundary.

Size determines the size of the transfer. The MPMC2 asserts `AddrAck` on the same cycle or on one of the following cycles.

After `AddrAck` has been asserted for one clock cycle, `AddrReq` must be deasserted on the following cycle unless another request is desired.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- `WrFIFO_Data` and `WrFIFO_BE` must be valid while `WrFIFO_Push` is asserted.
- `WrFIFO_Push` is not allowed to be asserted the clock cycle following a cycle where `WrFIFO_AlmostFull` is asserted.

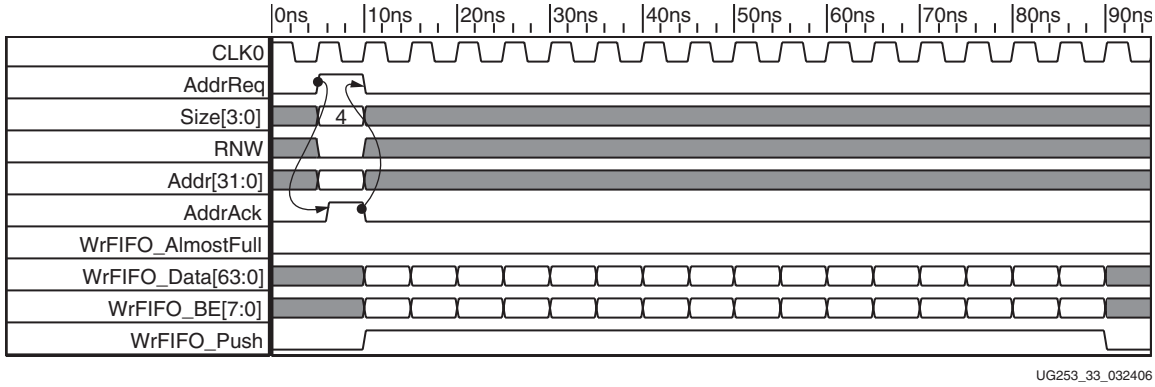


Figure 1-28: **32-Word Burst Write**

4-Word, Cache-Line, Write Back-To-Back Transfers

Figure 1-29 shows the timing diagram for two 4-word, cache-line, write transfers that are requested back-to-back.

To initiate a transfer:

- Assert `AddrReq` with `RNW` and `Addr`.
- Align `Addr` to a 4-word boundary.

Size determines the size of the transfer. The MPMC2 asserts `AddrAck` on the same cycle or on one of the following cycles.

After `AddrAck` has been asserted for one clock cycle, `AddrReq` remains asserted. In Figure 1-29, the second address acknowledge is delayed by one cycle, but the delay can be longer.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- The first two pushes correspond to the first request, and the second two pushes correspond to the second request.
- `WrFIFO_Data` and `WrFIFO_BE` must be valid while `WrFIFO_Push` is asserted.
- `WrFIFO_Push` is not allowed to be asserted the clock cycle following a cycle where `WrFIFO_AlmostFull` is asserted.

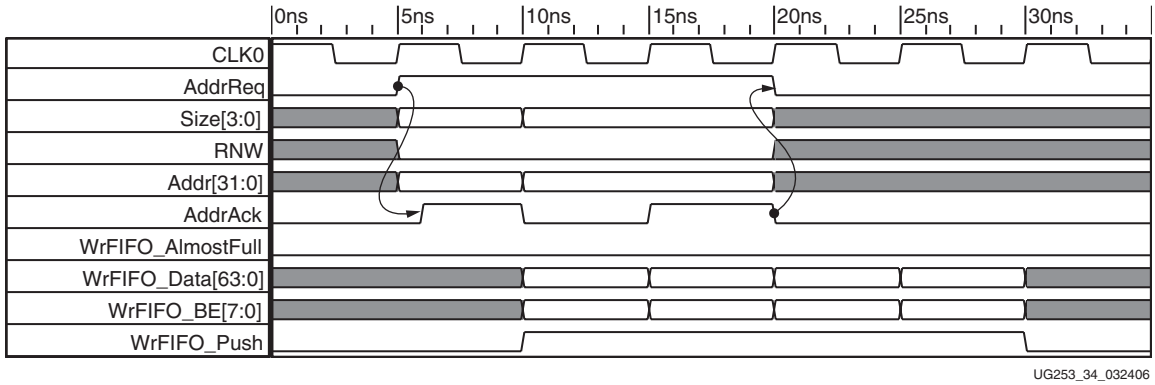


Figure 1-29: **Back-to-Back Write**

4-Word, Cache-Line, Write Transfers with Full Write FIFOs

Figure 1-30 shows the timing diagram for a 4-word, cache-line write transfer where the FIFOs become full halfway through the transfer.

To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr to a 4-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another request is required.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs after the address acknowledge, as is possible when using 32-bit memories or memories with data rates less than or equal to the NPI data rate.

- WrFIFO_Data and WrFIFO_BE must be valid while WrFIFO_Push is asserted.
- WrFIFO_Push is not allowed to be asserted the clock cycle following a cycle where WrFIFO_AlmostFull is asserted.

In the example, the assertion of WrFIFO_AlmostFull requires the second data push to be delayed by one clock cycle.

- MPMC2_PIM_<*PORTNUM*>_WrRIFO_AlmostFull is asserted to indicate that the Write FIFO will be full on the next cycle and MPMC2_PIM_<*PortNum*>_WrFIFO_Push cannot be asserted in the next cycle.
- MPMC2_PIM_<*PORTNUM*>_WrRIFO_AlmostFull is a combinatiorial signal that is asserted in the same cycle as the last MPMC2_PIM_<PortNum>_WrFIFO_Push that will fit into the write FIFO.
- MPMC2_PIM_<*PORTNUM*>_WrRIFO_AlmostFull will remain asserted until there is room in the FIFO for more data to be pushed.

To prevent circular logic, the PIM must register either the assertion of the PMC2_PIM_<*PortNum*>_WrFIFO_Push or MPMC2_PIM_<*PORTNUM*>_WrRIFO_AlmostFull before it is used.

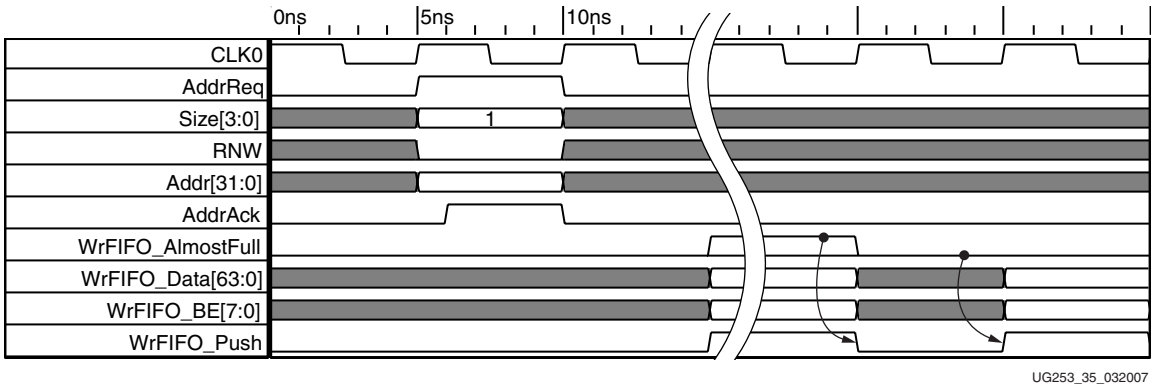


Figure 1-30: **4-Word, Cache-Line Write (Write FIFO Full)**

4-Word, Cache-Line Write Transfer (with Write Data Acks Before Request)

A Performance Improvement option that certain memory widths might require (such as 64-bit)

Figure 1-31 shows the timing diagram for a 4-word, cache-line write transfer with write data acknowledgements (Acks) occurring before the request. To initiate a transfer:

- Assert AddrReq with RNW and Addr.
- Align Addr to a 4-word boundary.

Size determines the size of the transfer. The MPMC2 asserts AddrAck on the same cycle or on one of the following cycles.

After AddrAck has been asserted for one clock cycle, AddrReq must be deasserted on the following cycle unless another word request is required.

The data phase is independent from the address phase (refer to the bulleted information at the beginning of "Native Port Interface (NPI)," page 40). This example shows the data being pushed into the write FIFOs before and with the address acknowledge, which allows for greater system performance.

- WrFIFO_Data and WrFIFO_BE must be valid while WrFIFO_Push is asserted.
- WrFIFO_Push is not allowed to be asserted in the clock cycle following a cycle where WrFIFO_AlmostFull is asserted.

If the logic connected to NPI cannot push the write data fast enough to be consumed by the memory, the NPI must delay the address request until all or enough data is pushed into the write FIFOs. For example, if the PIM is doing a CL8 write and the memory can consume the data twice as fast as the PIM can provide the data, it is safe for the PIM to push in 4 words, assert the address request, and continue pushing the data in at the same rate until all 8 words are in the FIFOs. If the PIM takes address and data pipelining into account, it might be able to adjust the push by a couple of cycles forward or backward as needed.

The MPMC2 core does not throttle writes to memory if the write FIFO is empty. The MPMC2 logic is based on the assumption that it can pop from the write FIFO at the data rate provided by memory.

Another consideration with performing an address request before all data is pushed is whether the FIFO type is SRL- or BRAM-based:

- BRAM FIFOs are large and deep such that they will not get full.
- In an SRL FIFO, it often reaches a full condition, especially with burst transfers.

When attempting to calculate how early address request can be asserted, you must take into account that the write FIFO almost full flag could be asserted after the request, which would affect these calculations. You can also choose to conservatively assert the address request after the write data is pushed in to prevent a write FIFO underrun.

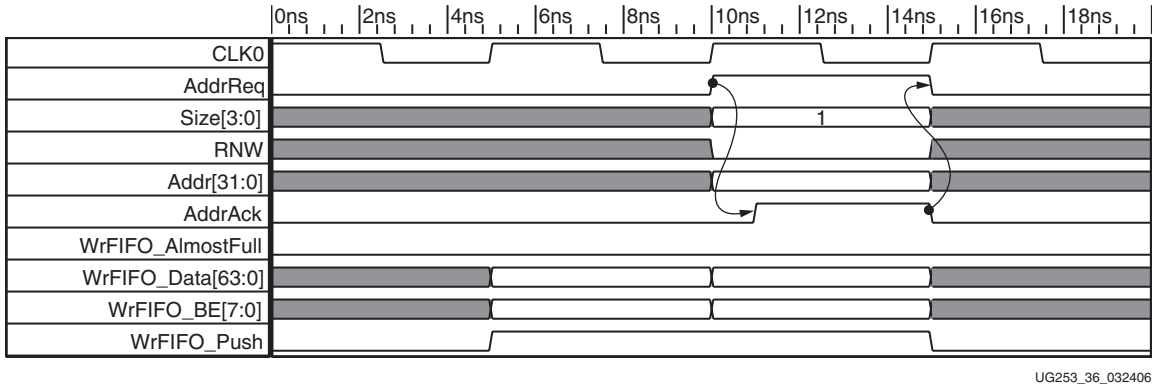


Figure 1-31: **4-Word, Cache-Line Write (Write Data Ack Occurs Before Address Req)**

On-Chip Peripheral Bus (OPB)

The OPB PIM, which is shown in Figure 1-32:

- Supports PowerPC 405 processor OPB requests for one-word reads and writes
- Provides support for sequential burst word transactions (on the slave interface)
- Serves as the master bridge from a PowerPC 405 processor to allow another PowerPC 405 processor to initiate transactions on OPB bus
- Connects OPB devices to the MPMC2 pcore
- Contains logic to respond to any slave side transactions provided by external arbiters
- Can initiate master transactions on the bus from the DSPLB PIMs, allowing for processors to talk directly to any OPB device. See the *IBM CoreConnect 64-Bit On-Chip Peripheral Bus: Architecture Specification* for more information. "Additional Resources," page 13 contains a link to the document.

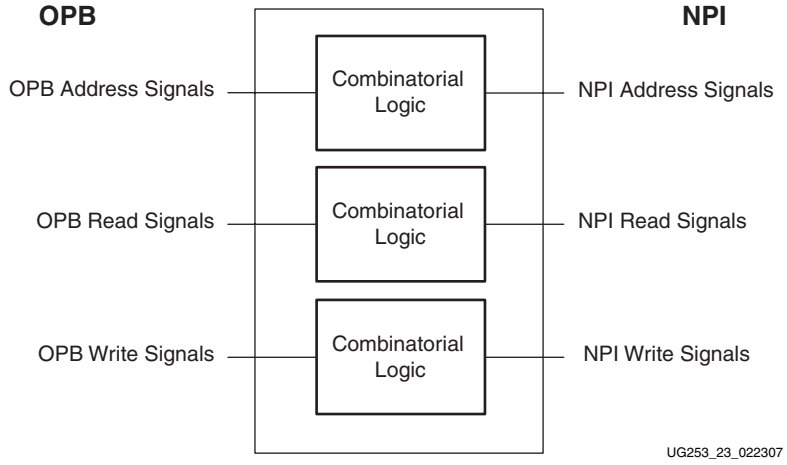


Figure 1-32: **OPB Block Diagram**

The following is a code snippet from the OPB MHS file.

```
PARAMETER C_OPB_0_BASEADDR = 0x10000000
PARAMETER C_OPB_0_HIGHADDR = 0x13FFFFFF
PARAMETER C_OPB_0_MPMC2_TO_PI_CLK_RATIO = 1
PARAMETER C_OPB_0_BRIDGE_TO_PI_CLK_RATIO = 1
BUS_INTERFACE OPB_S_0 = opb
PORT MPMC2_0_Clk0 = CLK_100MHz
```

OPB Clock Requirements

The OPB PIM must run at the same or half the frequency as MPMC2, as specified by `C_OPB_*_MPMC2_TO_PI_CLK_RATIO`.

Processor Local Bus (PLB)

The PLB, which is shown in Figure 1-33:

- Supports PowerPC 405 processor PLB requests:
  - one-word reads and writes
  - 4-word, cache-line reads and writes
  - 8-word, cache-line reads and writes
- Supports request aborts
- Provides additional support for burst reads and writes

  Note: When using an 8-bit memory, master devices that request unaligned burst transfers are not supported. The PLB PIM does not support indeterminate length bursts. It only supports fixed length bursts of 2-16 data beats.
- Serves as the master bridge from the PowerPC 405 processor to allow that processor to initiate transactions on PLB bus
- Connects PLB devices to the MPMC2 pcore
- Contains logic to respond to any slave-side transactions provided by external arbiters
- Can initiate master transactions on the bus from the DSPLB PIMs, allowing processors to communicate directly to any PLB device

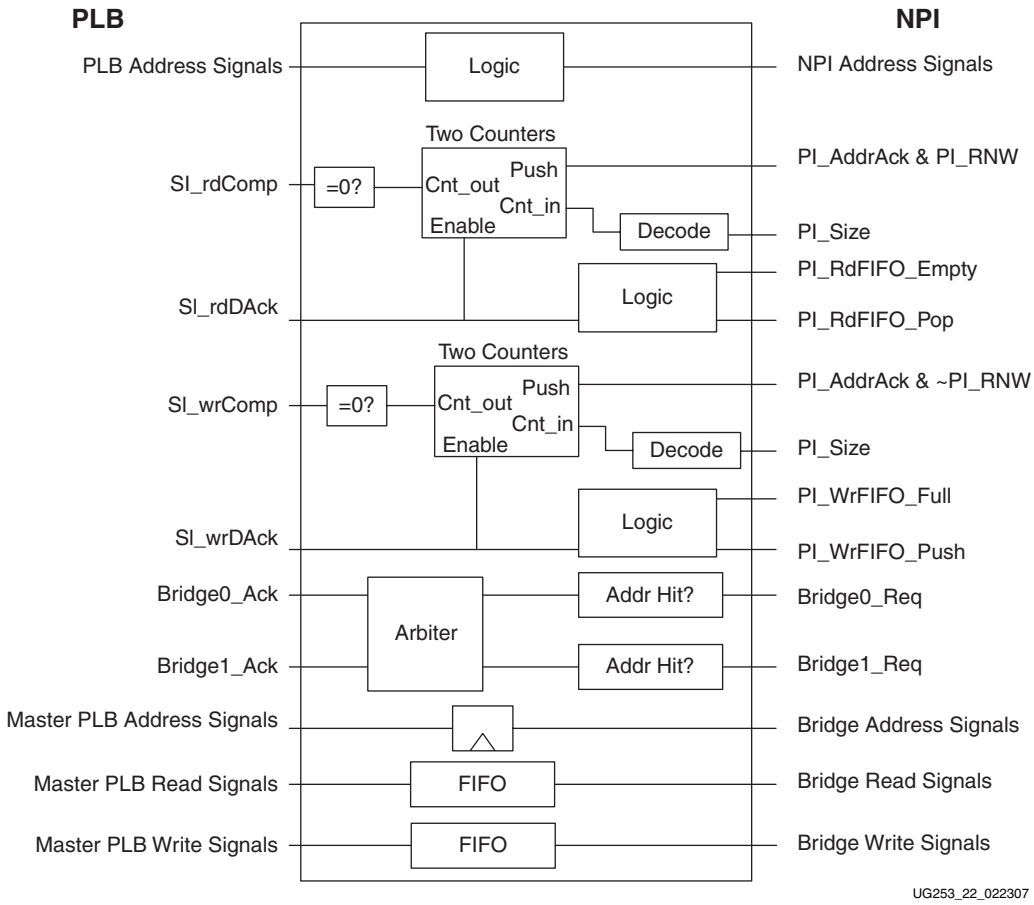


Figure 1-33: **PLB Block Diagram**

The following is a code snippet from the PLB MHS file.

```
PARAMETER C_PLB_2_BASEADDR = 0x0000_0000
PARAMETER C_PLB_2_HIGHADDR = 0x03ff_ffff
PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 1
PARAMETER C_PLB_2_BRIDGE_TO_PI_CLK_RATIO = 1
BUS_INTERFACE PLB_M_2 = plb_2
BUS_INTERFACE PLB_S_2 = plb_2
PORT MPMC2_0_Clk0 = CLK_100MHz
```

PLB Clock Requirements

The PLB PIM must run at the same or one-half the frequency as MPMC2, as specified by `C_PLB_*_MPMC2_TO_PI_CLK_RATIO`.

MicroBlaze Xilinx CacheLink (XCL)

The XCL PIM as shown in Figure 1-34:

- Supports XCL requests:
  - one-word writes
  - 4-word or 8-word, cache-line reads (user-configurable parameter)
- Connects an XCL interface to the MPMC2 pcore
- Contains the logic to respond to any slave-side XCL transaction emitted by the MicroBlaze soft processor

For additional details on XCL, see the *MicroBlaze Processor Reference Guide*." "Additional Resources," page 13 contains a link to the document.

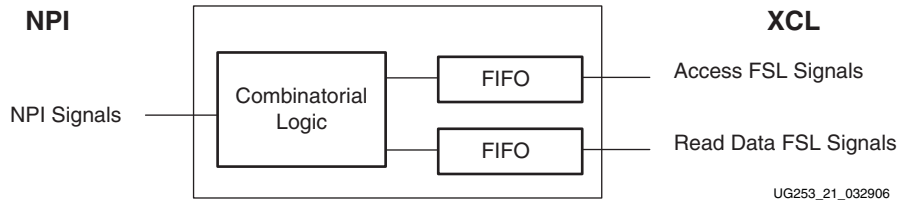


Figure 1-34: **MicroBlaze XCL Block Diagram**

The following is a code snippet from the XCL MHS file.

```
PARAMETER C_XCL_0_BASEADDR = 0x10000000
PARAMETER C_XCL_0_HIGHADDR = 0x13FFFFFF
BUS_INTERFACE XCL_0 = xcl_0
PORT XCL_0_Rst = sys_bus_reset_1 (optional, not required)
PORT MPMC2_0_Clk0 = CLK_100MHz
```

For instruction-side connected Microblaze XCL PIMs, you can set a parameter to disable write support to reduce logic utilization and to improve timing.

MicroBlaze XCL Clock Requirements

The XCL PIM runs at an integer ratio of the MPMC2. This clock ratio is automatically detected during reset and can be an integer ratio between 1:1 and 16:1.

The XCL PIM clock must be synchronous and rising edge-aligned to the MPMC2 memory clock.

I/O Signals and Parameters

System Signals and Parameters

Table 1-3 shows the system I/O signals. See "System Clocks on page 80" in Table 1-25, page 80 for more information.

Table 1-3: **System I/O Signals**

Signal Name	Direction	Init Status	Description
MPMC2_0_Clk_200MHz	Input	x	200 MHz clock input.
MPMC2_0_Clk_2X	Input	x	2x version of system clock. See Table 1-1, page 26, for more information. Reserved. Can be left unconnected
MPMC2_0_Clk_Cal	Input	x	Reserved. Can be left unconnected
MPMC2_0_Clk_Mem	Input	x	Reserved. Can be left unconnected
MPMC2_0_Clk0	Input	x	System clock input.
MPMC2_0_Clk90	Input	x	System clock input, phase shifted by 90 degrees.
MPMC2_0_Rst	Input	x	System reset input.
MPMC2_SlowestClk	Input	x	Reserved. Can be left unconnected
ECC_Intr	Output	0	ECC Interrupt; (level sensitive) 1=Interrupt asserted 0=No Interrupt
MPMC2_0_Idelayctrl_Rdy_I(1)	Input	set to 1 automatically if unconnected	This signal is AND'd with internal IDELAYCTRL RDY signals to indicate that memory initialization can begin
MPMC2_0_Idelayctrl_Rdy_O(1)	Output	0	1= All internal IDELAYCTRL RDY signals and MPMC2_0_Idelayctrl_Rdy_I are all high
MPMC2_0_InitDone	Output	0	Indicates memory initialization is complete

Note:

1. Signals are only applicable on Virtex-4 and Virtex-5 architectures.

Table 1-4 shows the system design parameters.

Table 1-4: **System Design Parameters**

Parameter	Allowable Values	Description
C_MPMC2_0_MAX_REQ_PENDING(1,2)	1–15	Indicates number of requests MPMC2 can queue per port
C_MPMC2_0_REQ_PENDING_CNTR_WIDTH(1,2)	1–4	Must be large enough to count to the value specified by C_MPMC2_0_MAX_REQ_PENDING
C_MPMC2_0_NUM_IDELAYCTRL	0,1,N(5)	Number of IDELAYCTRL elements to instantiate (Virtex-4 and Virtex-5 only). (Default = 1)
C_INCLUDE_ECC_SUPPORT	0,1	1 = ECC Enabled 0 = ECC DISABLED (default)
C_ECC_DEFAULT_ON(3)	0,1	1 = ECC enable upon reset (Default Value) 0 = ECC disabled upon reset (must write to ECCCR to enable ECC)

Table 1-4: **System Design Parameters *(Continued)***

Parameter	Allowable Values	Description
C_INCLUDE_ECC_TEST(3)	0,1	1 = Enable ECC test support 0 = No ECC test support (Default Value)
C_ECC_BASEADDR(3)	10-bit number	DCR base address of ECC registers
C_ECC_HIGHADDR(3)	10-bit number	DCR high address of ECC registers = base address + 0x00F
C_ECC_SEC_THRESHOLD(3)	1-4095	Specifies single-bit data error interrupt threshold counter value (Default = 1)
C_ECC_DEC_THRESOLD(3)	1-4095	Specifies double-bit data error interrupt threshold counter value (Default = 1)
C_ECC_PEC_THRESHOLD(3)	1-4095	Specifies parity field bit error interrupt threshold counter value (Default = 1)
C_PM_DCR_BASEADDR(4)	10-bit number	DCR base address of Performance Monitor registers
C_PM_DCR_HIGHADDR(4)	10-bit number	DCR high address of Performance Monitor = base address + 0x00F
C_PM_DC_CNTR(4)	8-bit vector	8-bit vector to enable 48-bit dead cycle counters on a per port basis 1= enable dead cycle counters (default) 0 = disable dead cycle counters MSB = port 8 LSB = port 1
C_PM_GC_CNTR(4)	0,1	1 = enable global cycle counters (default) 0 = disable global cycle counters
C_PM_SHIFT_BY(4)	0, 1, 2,3	Specifies the size of the histogram bins used by the Performance Monitors. See Appendix C, "MPMC2 Performance Monitoring," for more information about the performance monitors and the C_PM_SHIFT_BY parameter.

Notes:

1. Parameter values other than the default values are currently untested.
2. All the parameter default values are set by the MPMC2 GUI.
3. Parameter valid only when `C_INCLUDE_ECC_SUPPORT = 1`.
4. Parameter valid only when Performance Monitors are instantiated.
5. N can range up to the number IDELAYCTRL elements in the FPGA device.

MPMC2 Parameter and Port Dependencies

Table 1-5: **MPMC2 Parameter and Port Dependencies**

Parameter	Affects	Relationship Description
C_MPMC2_0_MEM_ADDR_WIDTH	MPMC2_0_DDR2_Addr_O MPMC2_0_DDR_Addr_O	Width of address to memory
C_MPMC2_0_MEM_BANKADDR_WIDTH	MPMC2_0_DDR2_BankAddr_O MPMC2_0_DDR_BankAddr_O	Width of bank address to memory
C_MPMC2_0_MEM_CE_WIDTH	MPMC2_0_DDR2_CE_O MPMC2_0_DDR_CE_O	Number of clock enable outputs
C_MPMC2_0_MEM_CLK_WIDTH	MPMC2_0_DDR2_Clk_O MPMC2_0_DDR2_Clk_n_O MPMC2_0_DDR_Clk_O MPMC2_0_DDR_Clk_n_O	Number of clock/inverted clock pair outputs
C_MPMC2_0_MEM_CS_N_WIDTH	MPMC2_0_DDR2_CS_n_O MPMC2_0_DDR_CS_n_O	Number of chip select output
C_MPMC2_0_MEM_DATA_WIDTH	MPMC2_0_DDR2_DQ_I MPMC2_0_DDR2_DQ_O MPMC2_0_DDR2_DQ_T MPMC2_0_DDR_DQ_I MPMC2_0_DDR_DQ_O MPMC2_0_DDR_DQ_T	Width of data at memory interface
C_MPMC2_0_MEM_DM_WIDTH	MPMC2_0_DDR2_DM_I MPMC2_0_DDR2_DM_O MPMC2_0_DDR2_DM_T MPMC2_0_DDR_DM_I MPMC2_0_DDR_DM_O MPMC2_0_DDR_DM_T	Width of data mask at memory interface
C_MPMC2_0_MEM_DQS_WIDTH	MPMC2_0_DDR2_DQS_I MPMC2_0_DDR2_DQS_O MPMC2_0_DDR2_DQS_T MPMC2_0_DDR2_DQS_n_I MPMC2_0_DDR2_DQS_n_O MPMC2_0_DDR2_DQS_n_T MPMC2_0_DDR_DQS_I MPMC2_0_DDR_DQS_O MPMC2_0_DDR_DQS_T	Width of data strobe bits at memory interface
C_MPMC2_0_MEM_ODT_WIDTH	MPMC2_0_DDR2_ODT_O	Number of ODT bits to memory
C_NUM_PORTS	MPMC2_PIM_<*PortNum*>_***	Determines the number of ports attached to the MPMC2

Table 1-5: **MPMC2 Parameter and Port Dependencies *(Continued)***

Parameter	Affects	Relationship Description
C_MPMC2_PIM_<*PortNum*> _ADDR_WIDTH	MPMC2_PIM_<*PortNum*>_Addr	Width of the address at each port
C_MPMC2_PIM_<*PortNum*> _BE_WIDTH	MPMC2_PIM_<*PortNum*>_WrDataBE	Number of byte enables at each port
C_MPMC2_PIM_<*PortNum*> _DATA_WIDTH	MPMC2_PIM_<*PortNum*>_WrData MPMC2_PIM_<*PortNum*>_RdData	Width of the data at each port
C_MPMC2_PIM_<*PortNum*> _RDWDADDR_WIDTH	MPMC2_PIM_<*PortNum*>_RdWdAddr	Number of bits required to encode the maximum transaction size

Memory Signals and Parameters

Table 1-6: **DDR I/O Signals**

Signal Name(1)	Direction	Init Status	Description
MPMC2_0_DDR_Addr_O[C_MPMC2_0_MEM_ADDR_WIDTH -1:0]	Output	x	Row/Column address
MPMC2_0_DDR_BankAddr_O[C_MPMC2_0_MEM_BANKADDR_WIDTH-1:0]	Output	x	Bank address
MPMC2_0_DDR_CAS_n_O	Output	1	Command input
MPMC2_0_DDR_CE_O[C_MPMC2_0_MEM_CE_WIDTH-1:0]	Output	0	1 = Clock enabled
MPMC2_0_DDR_Clk_n_O[C_MPMC2_0_MEM_CLK_WIDTH-1:0]	Output	1	Inverted clock to memory
MPMC2_0_DDR_Clk_O[C_MPMC2_0_MEM_CLK_WIDTH-1:0]	Output	0	Clock to memory
MPMC2_0_DDR_CS_n_O[C_MPMC2_0_MEM_CS_N_WIDTH-1:0]	Output	1	0 = Chip select enabled
MPMC2_0_DDR_DM_O [C_MPMC2_0_MEM_DM_WIDTH+C_MPMC2_0_ECC_DM_WIDTH-1:0]	Output	x	Data mask outputs
MPMC2_0_DDR_DQ [C_MPMC2_0_MEM_DATA_WIDTH+C_MPMC2_0_ECC_DATA_WIDTH-1:0]	InOut	x	Data
MPMC2_0_DDR_DQS [C_MPMC2_0_MEM_DQS_WIDTH+C_MPMC2_0_ECC_DQS_WIDTH-1:0]	InOut	x	Data strobe
MPMC2_0_DDR_RAS_n_O	Output	1	Command input
MPMC2_0_DDR_WE_n_O	Output	1	Command input
MPMC2_0_DDR_DQS_DIV_O(2)	Output	x	Timing loop signal
MPMC2_0_DDR_DQS_DIV_I(2)	Input	x	Timing loop signal

Notes:

1. For detailed signal descriptions, refer to device-specific data sheets.
2. Spartan 3 generation devices only.

Table 1-7: **DDR2 I / 0 Signals**

Signal Name(1)	Direction	Init Status	Description
MPMC2_0_DDR2_Addr_O [C_MPMC2_0_MEM_ADDR_WIDTH-1:0]	Output	x	Row and column address
MPMC2_0_DDR2_BankAddr_O [C_MPMC2_0_MEM_BANKADDR_WIDTH-1:0]	Output	x	Bank address
MPMC2_0_DDR2_CAS_n_O	Output	1	Command input
MPMC2_0_DDR2_CE_O [C_MPMC2_0_MEM_CE_WIDTH-1:0]	Output	0	1 = Clock enabled
MPMC2_0_DDR2_Clk_n_O [C_MPMC2_0_MEM_CLK_WIDTH-1:0]	Output	1	Inverted clock to memory
MPMC2_0_DDR2_Clk_O [C_MPMC2_0_MEM_CLK_WIDTH-1:0]	Output	0	Clock to memory
MPMC2_0_DDR2_CS_n_O [C_MPMC2_0_MEM_CS_N_WIDTH-1:0]	Output	1	0 = Chip select enabled
MPMC2_0_DDR2_DM_O [C_MPMC2_0_MEM_DM_WIDTH +C_MPMC2_O_ECC_DM_DATA_WIDTH-1:0]	Output	x	Data mask outputs
MPMC2_0_DDR2_DQ [C_MPMC2_0_MEM_DATA_WIDTH +C_MPMC2_O_ECC_DATA_WIDTH-1:0]	InOut	x	Data
MPMC2_0_DDR2_DQS_n [C_MPMC2_0_MEM_DATA_WIDTH +C_MPMC2_O_ECC_DQS_WIDTH-1:0]	Input	x	Inverted data strobe. Valid only when `C_MPMC_0_MEM_DQSN_ENABLE =1`
MPMC2_0_DDR2_DQS [C_MPMC2_0_MEM_DQS_WIDTH +C_MPMC2_O_ECC_DQS_WIDTH-1:0]	Input	x	Read data strobe inputs
MPMC2_0_DDR2_DQS_DIV_O(2)	Output	x	Timing loop signal
MPMC2_0_DDR2_DQS_DIV_I(2)	Input	x	Timing loop signal

Notes:

1. For detailed signal descriptions, refer to device-specific data sheets.
2. Spartan-3 generation devices only.

Memory Parameters

Table 1-8: **Memory Parameters**

Parameter[1]	Allowable Values	Description
C_MPMC2_0_MEM_BANKADDR_WIDTH	2, 3	Width of bank address to memory
C_MPMC2_0_MEM_CAS_LATENCY0	Any integer	CAS Latency of memory 0[2]
C_MPMC2_0_MEM_CE_WIDTH	Any integer	Number of clock enable outputs
C_MPMC2_0_MEM_CLK_WIDTH	Any integer	Number of clock/inverted clock pair outputs
C_MPMC2_0_MEM_CS_N_WIDTH	Multiple of NUM_DIMMs x NUM_RANKS	Number of chip select outputs
C_MPMC2_0_MEM_DATA_WIDTH	8, 16, 32, 64	Width of data at memory interface
C_MPMC2_0_MEM_DM_WIDTH	0, 4, 8, 16	Width of data mask at memory interface
C_MPMC2_0_MEM_DQS_WIDTH	2, 4, 8, 16	Width of data strobe bits at memory interface
C_MPMC2_0_MEM_ODT_WIDTH	Any integer	Number of ODT bits to memory
C_MPMC2_0_MEM_DQSN_ENABLE	0, 1	1= Memory has differential DQS (default) 0 = Memory has single-ended DQS. This parameter is only valid for DDR2 memory

Notes:

1. All the parameter default values are set by the MPMC2 GUI.
2. Non-integer CAS latencies (such as 1.5, 2.5, and so on) are not supported.

CDMAC Signals and Parameters

The following tables list the CDMAC I/O signals and design parameters.

Table 1-9: **CDMAC PIM I/O Signal Description**

Signal Name	Direction	Init Status	Description
CDMAC_<*PortNum*>_CLK	Input	x	CDMAC clock
CDMAC_<*PortNum*>_CRST	Input	x	CDMAC reset
CDMAC_<*PortNum*>_CDMAC_INT	Output	0	CDMAC interrupt
CDMAC_<*PortNum*>_DCR_ABus	Input	x	DCR address bus
CDMAC_<*PortNum*>_DCR_DBusIn	Input	x	DCR data bus input
CDMAC_<*PortNum*>_DCR_Write	Input	x	DCR write request
CDMAC_<*PortNum*>_DCR_Read	Input	x	DCR read request
CDMAC_<*PortNum*>_DCR_Ack	Output	0	DCR acknowledge
CDMAC_<*PortNum*>_DCR_DBusOut	Output	x	DCR data bus output
CDMAC_<*PortNum*>_TX_D	Output	x	Data bus. Valid while CDMAC_TX_Src_Rdy and CDMAC_TX_Dst_Rdy are asserted
CDMAC_<*PortNum*>_TX_Rem	Output	x	TX remainder. Data mask for last word of header, payload, or footer
CDMAC_<*PortNum*>_TX_SOF	Output	1	TX start of frame. Active-Low
CDMAC_<*PortNum*>_TX_EOF	Output	1	TX end of frame. Active-Low
CDMAC_<*PortNum*>_TX_SOP	Output	1	TX start of payload. Active-Low
CDMAC_<*PortNum*>_TX_EOP	Output	1	TX end of payload. Active-Low
CDMAC_<*PortNum*>_TX_Src_Rdy	Output	1	TX source ready. Active-Low. Indicates CDMAC has valid data on the TXn LocalLink outputs
CDMAC_<*PortNum*>_TX_Dst_Rdy	Input	x	TX Destination ready. Active-Low. Indicates connecting device is ready to receive data
CDMAC_<*PortNum*>_RX_D	Input	x	RX Data bus. Valid while RXn_Src_Rdy and RXn_Dst_Rdy are asserted
CDMAC_<*PortNum*>_RX_Rem	Input	x	RX remainder. Data mask for last word of header, payload, or footer
CDMAC_<*PortNum*>_RX_SOF	Input	x	RX start of frame. Active-Low
CDMAC_<*PortNum*>_RX_EOF	Input	x	RX end of frame. Active-Low
CDMAC_<*PortNum*>_RX_SOP	Input	x	RX start of payload. Active-Low
CDMAC_<*PortNum*>_RX_EOP	Input	x	RX end of payload. Active-Low
CDMAC_<*PortNum*>_RX_Src_Rdy	Input	x	RX source ready. Active-Low. Indicates connecting device has valid data on the RXn LocalLink outputs

Table 1-9: **CDMAC PIM I/O Signal Description *(Continued)***

Signal Name	Direction	Init Status	Description
CDMAC_<*PortNum*>_CLK	Input	x	CDMAC clock
CDMAC_<*PortNum*>_CRST	Input	x	CDMAC reset
CDMAC_<*PortNum*>_CDMAC_INT	Output	0	CDMAC interrupt
CDMAC_<*PortNum*>_DCR_ABus	Input	x	DCR address bus
CDMAC_<*PortNum*>_DCR_DBusIn	Input	x	DCR data bus input
CDMAC_<*PortNum*>_DCR_Write	Input	x	DCR write request
CDMAC_<*PortNum*>_DCR_Read	Input	x	DCR read request
CDMAC_<*PortNum*>_DCR_Ack	Output	0	DCR acknowledge
CDMAC_<*PortNum*>_DCR_DBusOut	Output	x	DCR data bus output
CDMAC_<*PortNum*>_TX_D	Output	x	Data bus. Valid while CDMAC_TX_Src_Rdy and CDMAC_TX_Dst_Rdy are asserted
CDMAC_<*PortNum*>_TX_Rem	Output	x	TX remainder. Data mask for last word of header, payload, or footer
CDMAC_<*PortNum*>_TX_SOF	Output	1	TX start of frame. Active-Low
CDMAC_<*PortNum*>_TX_EOF	Output	1	TX end of frame. Active-Low
CDMAC_<*PortNum*>_TX_SOP	Output	1	TX start of payload. Active-Low
CDMAC_<*PortNum*>_TX_EOP	Output	1	TX end of payload. Active-Low
CDMAC_<*PortNum*>_TX_Src_Rdy	Output	1	TX source ready. Active-Low. Indicates CDMAC has valid data on the TXn LocalLink outputs
CDMAC_<*PortNum*>_TX_Dst_Rdy	Input	x	TX Destination ready. Active-Low. Indicates connecting device is ready to receive data
CDMAC_<*PortNum*>_RX_D	Input	x	RX Data bus. Valid while RXn_Src_Rdy and RXn_Dst_Rdy are asserted
CDMAC_<*PortNum*>_RX_Rem	Input	x	RX remainder. Data mask for last word of header, payload, or footer
CDMAC_<*PortNum*>_RX_SOF	Input	x	RX start of frame. Active-Low
CDMAC_<*PortNum*>_RX_EOF	Input	x	RX end of frame. Active-Low
CDMAC_<*PortNum*>_RX_SOP	Input	x	RX start of payload. Active-Low
CDMAC_<*PortNum*>_RX_EOP	Input	x	RX end of payload. Active-Low
CDMAC_<*PortNum*>_RX_Src_Rdy	Input	x	RX source ready. Active-Low. Indicates connecting device has valid data on the RXn LocalLink outputs
CDMAC_<*PortNum*>_RX_Dst_Rdy	Output	1	RX Destination ready. Active-Low. Indicates CDMAC is ready to receive data
CDMAC_<*PortNum*>_DCR_Read	Input	x	DCR read request

Table 1-10: **CDMAC PIM Design Parameters**

Parameter	Allowable Values	Description
C_CDMAC_<*PortNum*>_DCR_HIGHADDR	10-bit number	CDMAC DCR high address
C_CDMAC_<*PortNum*>_COMPLETED_ERR_TX	0,1	Enable/disable completed error on TX engine
C_CDMAC_<*PortNum*>_COMPLETED_ERR_RX	0,1	Enable/disable completed error on RX engine
C_CDMAC_<*PortNum*>_INSTANTIATE_TIMER_TX	0,1	Enable/disable TX interrupt interval timer
C_CDMAC_<*PortNum*>_INSTANTIATE_TIMER_RX	0,1	Enable/disable RX interrupt interval timer
C_CDMAC_<*PortNum*>_PRESCALAR	8-bit number	Clock divide factor to interrupt interval timer clock
C_CDMAC_<*PortNum*>_MPMC2_TO_PI_CLK_RATIO	1,2	MPMC2 to CDMAC clock ratio MPMC2:PIM 1 for 1:1 2 for 2:1

Note: All the parameter default values are set by the MPMC2 GUI.

DSPLB Signals and Parameters

The follow tables list DSPLB I / O signals and design parameters.

Table 1-11: **DSPLB PIM I/O Signal Description**

Signal Name	Direction	Init Status	Description
DSPLB_<*PortNum*>_PLB_SlClk	Input	x	PLB clock
DSPLB_<*PortNum*>_PLB_SlRst	Input	x	PLB reset
DSPLB_<*PortNum*>_PLB_abort	Input	x	PLB abort bus request indicator
DSPLB_<*PortNum*>_PLB_ABus [0:C_PLB_AWIDTH-1]	Input	x	PLB address bus
DSPLB_<*PortNum*>_PLB_BE [0:(C_PLB_DWIDTH/8)-1]	Input	x	PLB byte enables
DSPLB_<*PortNum*>_PLB_busLock	Input	x	PLB bus lock
DSPLB_<*PortNum*>_PLB_compress	Input	x	PLB compressed data transfer indicator
DSPLB_<*PortNum*>_PLB_guarded	Input	x	PLB guarded transfer indicator
DSPLB_<*PortNum*>_PLB_lockErr	Input	x	PLB lock error indicator
DSPLB_<*PortNum*>_PLB_masterID [0:C_PLB_MID_WIDTH-1]	Input	x	PLB current master indicator
DSPLB_<*PortNum*>_PLB_MSize [0:1]	Input	x	PLB master data bus size
DSPLB_<*PortNum*>_PLB_ordered	Input	x	PLB synchronize transfer indicator
DSPLB_<*PortNum*>_PLB_PAValid	Input	x	PLB primary address valid indicator
DSPLB_<*PortNum*>_PLB_rdBurst	Input	x	PLB burst read transfer indicator
DSPLB_<*PortNum*>_PLB_rdPrim	Input	x	PLB secondary to primary read request indicator
DSPLB_<*PortNum*>_PLB_RNW	Input	x	PLB read not write

Table 1-11: **DSPLB PIM I/O Signal Description *(Continued)***

Signal Name	Direction	Init Status	Description
DSPLB_<*PortNum*>_PLB_SAValid	Input	x	PLB secondary address valid indicator
DSPLB_<*PortNum*>_PLB_size [0:3]	Input	x	PLB transfer size
DSPLB_<*PortNum*>_PLB_type [0:2]	Input	x	PLB transfer type
DSPLB_<*PortNum*>_PLB_wrBurst	Input	x	PLB burst write transfer indicator
DSPLB_<*PortNum*>_PLB_wrDBus [0:C_PLB_DWIDTH-1]	Input	x	PLB write data bus
DSPLB_<*PortNum*>_PLB_wrPrim	Input	x	PLB secondary to primary write request indicator
DSPLB_<*PortNum*>_PLB_pendReq	Input	x	PLB pending bus request indicator
DSPLB_<*PortNum*>_PLB_pendPri [0:1]	Input	x	PLB pending request priority
DSPLB_<*PortNum*>_PLB_reqPri [0:1]	Input	x	PLB current request priority
DSPLB_<*PortNum*>_Sl_addrAck	Output	0	PLB slave address acknowledge
DSPLB_<*PortNum*>_Sl_MErr	Output	0	PLB slave error
DSPLB_<*PortNum*>_Sl_MBusy	Output	0	PLB slave busy
DSPLB_<*PortNum*>_Sl_rdBTerm	Output	0	PLB slave read burst terminate
DSPLB_<*PortNum*>_Sl_rdComp	Output	0	PLB slave read transfer complete indicator
DSPLB_<*PortNum*>_Sl_rdDAck	Output	0	PLB slave read data acknowledge
DSPLB_<*PortNum*>_Sl_rdDBus [0:C_PLB_DWIDTH-1]	Output	x	PLB slave read data bus
DSPLB_<*PortNum*>_Sl_rdWdAddr [0:3]	Output	x	PLB slave read word address
DSPLB_<*PortNum*>_Sl_rearbitrate	Output	0	PLB slave rearbitrate
DSPLB_<*PortNum*>_Sl_SSize [0:1]	Output	0	PLB slave data bus size
DSPLB_<*PortNum*>_Sl_wait	Output	0	PLB slave wait indicator
DSPLB_<*PortNum*>_Sl_wrBTerm	Output	0	PLB slave write burst terminate
DSPLB_<*PortNum*>_Sl_wrComp	Output	0	PLB slave write transfer complete indicator
DSPLB_<*PortNum*>_Sl_wrDAck	Output	0	PLB slave write data acknowledge

Table 1-12: **Data-Side PIM (DSPLB) Design Parameters**

Parameter(2)	Allowable Values	Description
C_DSPLB_<*PortNum*>_MPMC2_TO_PI_CLK_RATIO(1)	1, 2	MPMC2 to DSPLB clock ratio. MPMC2:PIM 1 for 1:1 2 for 2:1

Notes:

1. All ISPLB/DSPLB PIM ratio parameters must be set the same.
2. All the parameter default values are set by the MPMC2 GUI.

Error Correction Code (ECC) DCR I / O Signals

Table 1-13 lists the I/O signals for the DCR interface used for ECC control and status registers.

Note: This interface is only applicable if C_INCLUDE_ECC_SUPPORT = 1.

Table 1-13: **ECC I/O Signal Description (Port Interface = ECC_S_DCR)**

Signal Name	Direction	Init Status	Description
ECC_DCR_ABus[0:9]	Input	x	DCR address bus.
ECC_DCR_DBusIn[0:31]	Input	x	DCR data bus input.
ECC_DCR_Write	Input	x	DCR write request
ECC_DCR_Read	Input	x	DCR read request.
ECC_DCR_Ack	Output	0	DCR acknowledge
ECC_DCR_DBusOut[0:31]	Output	x	DCR data bus output
ECC_DCR_Clk	Input	x	DCR Clock

ISPLB Signals and Parameters

The following tables list the ISPLB signals and design parameters

Table 1-14: **ISPLB I/O Signals**

Signal Name	Direction	Init Status	Description
ISPLB_<*PortNum*>_PLB_SlClk	Input	x	PLB clock
ISPLB_<*PortNum*>_PLB_SlRst	Input	x	PLB reset
ISPLB_<*PortNum*>_PLB_abort	Input	x	PLB abort bus request indicator
ISPLB_<*PortNum*>_PLB_ABus [0:C_PLB_AWIDTH-1]	Input	x	PLB address bus
ISPLB_<*PortNum*>_PLB_BE [0:(C_PLB_DWIDTH/8)-1]	Input	x	PLB byte enables
ISPLB_<*PortNum*>_PLB_busLock	Input	x	PLB bus lock
ISPLB_<*PortNum*>_PLB_compress	Input	x	PLB compressed data transfer indicator
ISPLB_<*PortNum*>_PLB_guarded	Input	x	PLB guarded transfer indicator
ISPLB_<*PortNum*>_PLB_lockErr	Input	x	PLB lock error indicator
ISPLB_<*PortNum*>_PLB_masterID [0:C_PLB_MID_WIDTH-1]	Input	x	PLB current master indicator
ISPLB_<*PortNum*>_PLB_MSize [0:1]	Input	x	PLB master data bus size
ISPLB_<*PortNum*>_PLB_ordered	Input	x	PLB synchronize transfer indicator
ISPLB_<*PortNum*>_PLB_PAValid	Input	x	PLB primary address valid indicator
ISPLB_<*PortNum*>_PLB_rdBurst	Input	x	PLB burst read transfer indicator

Table 1-14: **ISPLB I/O Signals *(Continued)***

Signal Name	Direction	Init Status	Description
ISPLB_<*PortNum*>_PLB_rdPrim	Input	x	PLB secondary to primary write request indicator
ISPLB_<*PortNum*>_PLB_RNW	Input	x	PLB read not write
ISPLB_<*PortNum*>_PLB_SAValid	Input	x	PLB secondary address valid indicator
ISPLB_<*PortNum*>_PLB_size [0:3]	Input	x	PLB transfer size
ISPLB_<*PortNum*>_PLB_type [0:2]	Input	x	PLB transfer type
ISPLB_<*PortNum*>_PLB_wrBurst	Input	x	PLB burst write transfer indicator
ISPLB_<*PortNum*>_PLB_wrDBus [0:C_PLB_DWIDTH-1]	Input	x	PLB write data bus
ISPLB_<*PortNum*>_PLB_wrPrim	Input	x	PLB secondary to primary write request indicator
ISPLB_<*PortNum*>_PLB_pendReq	Input	x	PLB pending bus request indicator
ISPLB_<*PortNum*>_PLB_pendPri [0:1]	Input	x	PLB pending request priority
ISPLB_<*PortNum*>_PLB_reqPri [0:1]	Input	x	PLB current request priority
ISPLB_<*PortNum*>_Sl_addrAck	Output	0	PLB slave address acknowledge
ISPLB_<*PortNum*>_Sl_MErr	Output	0	PLB slave error
ISPLB_<*PortNum*>_Sl_MBusy	Output	0	PLB slave busy
ISPLB_<*PortNum*>_Sl_rdBTerm	Output	0	PLB slave read burst terminate
ISPLB_<*PortNum*>_Sl_rdComp	Output	0	PLB slave read transfer complete indicator
ISPLB_<*PortNum*>_Sl_rdDAck	Output	0	PLB slave read data acknowledge
ISPLB_<*PortNum*>_Sl_rdDBus [0:C_PLB_DWIDTH-1]	Output	x	PLB slave read data bus
ISPLB_<*PortNum*>_Sl_rdWdAddr [0:3]	Output	x	PLB slave read word address
ISPLB_<*PortNum*>_Sl_rearbitrate	Output	0	PLB slave rearbitrate
ISPLB_<*PortNum*>_Sl_SSize [0:1]	Output	0	PLB slave data bus size
ISPLB_<*PortNum*>_Sl_wait	Output	0	PLB slave wait indicator
ISPLB_<*PortNum*>_Sl_wrBTerm	Output	0	PLB slave write burst terminate
ISPLB_<*PortNum*>_Sl_wrComp	Output	0	PLB slave write transfer complete indicator
ISPLB_<*PortNum*>_Sl_wrDAck	Output	0	PLB slave write data acknowledge

ISPLB Design Parameters

Table 1-15: **Instruction-Side PLB (ISPLB) PIM Design Parameters**

Parameter[2]	Allowable Values	Description
C_ISPLB_<*PortNum*>_MPMC2_TO_PI_CLK_RATIO [1]	1, 2	MPMC2 to ISPLB clock ratio. MPMC2:PIM 1 for 1:1 2 for 2:1

Notes:

1. All ISPLB/DSPLB PIM ratio parameters must be set the same.
2. All the parameter default values are set by the MPMC2 GUI.

NPI Signals and Parameters

The following tables show the NPI PIM I/O signals and parameters.

Table 1-16: **NPI PIM Signals**

Signal Name	Direction	Init Status	Description
MPMC2_PIM_<*PortNum*>_Addr [C_MPMC2_PIM_<*PortNum*>_ADDR_WIDTH-1:0]	Input	x	Start address. Valid with MPMC2_PIM_<*PortNum*>_AddrReq.
MPMC2_PIM_<*PortNum*>_AddrAck	Output	0	1 = Acknowledge MPMC2_PIM_<*PortNum*>_AddrReq. Valid for one clock cycle.
MPMC2_PIM_<*PortNum*>_AddrReq	Input	0	1 = Request memory transfer. Held until MPMC2_PIM_<*PortNum*>_AddrAck is asserted. Can have back to back requests.
MPMC2_PIM_<*PortNum*>_RdFIFO_Data [C_MPMC2_PIM_<*PortNum*>_DATA_WIDTH-1:0]	Output	0	Read data. Valid when MPMC2_PIM_<*PortNum*>_RdFIFO_Empty not asserted.
MPMC2_PIM_<*PortNum*>_RdFIFO_Empty	Output	0	1 = Read FIFOs are empty 0 =Read FIFPOs are not empty and at least 64-bits of data is available. If memory is very narrow in width, it waits until 64-bits of read data has been accumulated before being deasserted
MPMC2_PIM_<*PortNum*>_RdFIFO_Pop	Input	0	1 = Indicates MPMC2_PIM_<*PortNum*>_RdFIFO_Data has been processed and the next MPMC2_PIM_<*PortNum*>_RdFIFO_Data should be available on the next cycle. Valid for one clock cycle.
MPMC2_PIM_<*PortNum*>_RdFIFO_RdWdAddr [C_MPMC2_PIM_<*PortNum*>_RDWDADDR_WIDTH-1:0]	Output	0	Indicates which word of the transfer is being displayed on MPMC2_PIM_<*PortNum*>_RdFIFO_Data
MPMC2_PIM_<*PortNum*>_RNW	Input	x	1 = Request Read Transfer 0 = Request Write Transfer Valid with MPMC2_PIM_<*PortNum*>_AddrReq.

Table 1-16: **NPI PIM Signals *(Continued)***

Signal Name	Direction	Init Status	Description
MPMC2_PIM_<*PortNum*>_Size[3:0]	Input	0	0 = Word/half-word/byte 1 = 4-Word Cache-Line 2 = 8-Word Cache-Line 4 = 32-Word Burst 5 = 64-Word Burst Valid when MPMC2_PIM_<*PortNum*>_AddrReq is valid
MPMC2_PIM_<*PortNum*>_WrFIFO_AlmostFull	Output	0	1 = Write FIFO will be full on the next cycle and MPMC2_PIM_<PortNum>_WrFIFO_Push cannot be asserted in the next cycle Combinatorial signal that is asserted in same cycle as the last MPMC2_PIM_<PortNum>_WrFIFO_Push that will fit into the Write FIFO Signal will remain asserted until there is room in the FIFO for more data To prevent circular logic, the PIM must register the assertion of this signal or MPMC2_PIM_<PortNum>_WrFIFO_Push before using
MPMC2_PIM_<*PortNum*>_WrFIFO_BE [C_MPMC2_PIM_<*PortNum*>_BE_WIDTH-1:0]	Input	x	Write data byte enables. Valid with MPMC2_PIM_<*PortNum*>_WrFIFO_Push
MPMC2_PIM_<*PortNum*>_WrFIFO_Data [C_MPMC2_PIM_<*PortNum*>_DATA_WIDTH-1:0]	Input	x	Write data. Valid with MPMC2_PIM_<*PortNum*>_WrFIFO_Push
MPMC2_PIM_<*PortNum*>_WrFIFO_Push	Input	x	1 = Indicates MPMC2_PIM_<*PortNum*>_WrFIFO_Data is valid and should be pushed into the MPMC2. Valid for one clock cycle
MPMC2_PIM_<*PortNum*>_RdFIFO_Data_Available	Output	0	Reserved
MPMC2_PIM_<*PortNum*>_RdFIFO_Flush	Input	x	1 = Indicates read FIFOs should be flushed
MPMC2_PIM_<*PortNum*>_WrFIFO_Flush	Input	x	1 = Indicates write FIFOs should be flushed
MPMC2_PIM_<PortNum>_WrFIFO_Empty	Output	1	1 = Write FIFO is completely empty 0 = Write FIFO is not empty
MPMC2_PIM_<PortNum>_InitDone	Output	0	1 = Memory initialization completed 0 = Memory initialization not completed

Table 1-16: **NPI PIM Signals *(Continued)***

Signal Name	Direction	Init Status	Description
MPMC2_PIM_<PortNum>_RD_FIFO_LATENCY[1:0]	Output	0, 1, or 2	Number of clock cycles of latency between MPMC2_PIM_<PortNum>_RdFIFO_Pop and valid data on MPMC2_PIM_<PortNum>_RdFIFO_Data and MPMC2_PIM_<PortNum>_RdFIFO_RdWdAddr See "Native Port Interface (NPI)," page 40 for more information.
MPMC2_PIM_<PortNum>_RdModWr	Input	X	1 = Write transaction may require read-modify-write operation for ECC or for memory without data masks (default) 0 = Write transaction is aligned across memory size or ECC word size so read-modify-write is not required Note: This signal is applicable only when qualified by MPMC2_PIM_<PortNum>_AddrReq

Table 1-17: **NPI PIM Design Parameters**

Parameter	Allowable Values	Description
C_MPMC2_PIM_<*PortNum*>_ADDR_WIDTH	32	Port address width
C_MPMC2_PIM_<*PortNum*>_BASEADDR	0x00000000–0xFFFFFFFF	The memory base address that MPMC2 will respond to for port <*PortNum*>
C_MPMC2_PIM_<*PortNum*>_BE_WIDTH	8	Port byte enable width
C_MPMC2_PIM_<*PortNum*>_DATA_WIDTH	64	Port data width
C_MPMC2_PIM_<*PortNum*>_HIGHADDR	0x00000000–0xFFFFFFFF	The memory high address that MPMC2 will respond to for port <*PortNum*>
C_MPMC2_PIM_<*PortNum*>_OFFSET	0x00000000–0xFFFFFFFF	Amount by which the base/high address should be offset. Not currently supported.

Note: All the parameter default values are set by the MPMC2 GUI.

OPB Signals and Parameters

The following tables list the On-Chip Peripheral Bus (OPB) PIM I / O signals and design parameters.

Table 1-18: **OPB PIM I/O Signals**

Signal Name	Direction	Init Status	Description
OPB_<*PortNum*>_CLK	Input	x	OPB clock
OPB_<*PortNum*>_RST	Input	x	OPB reset
OPB_<*PortNum*>_M_ABus [0:C_OPB_AWIDTH-1]	Output	x	OPB master address bus
OPB_<*PortNum*>_M_BE [0:3]	Output	0	OPB master byte enables
OPB_<*PortNum*>_M_busLock	Output	0	OPB master bus lock

Table 1-18: **OPB PIM I/O Signals *(Continued)***

Signal Name	Direction	Init Status	Description
OPB_<*PortNum*>_M_DBus [0:C_OPB_DWIDTH-1]	Output	x	OPB master write data bus
OPB_<*PortNum*>_M_DBusEn	Output	0	OPB master write data bus enable
OPB_<*PortNum*>_M_request	Output	0	OPB master request
OPB_<*PortNum*>_M_RNW	Output	0	OPB master read, not write
OPB_<*PortNum*>_M_select	Output	0	OPB master select
OPB_<*PortNum*>_M_seqAddr	Output	0	OPB master sequential address
OPB_<*PortNum*>_OPB_errAck	Input	x	OPB master error acknowledge
OPB_<*PortNum*>_OPB_MGrant	Input	x	OPB master grant
OPB_<*PortNum*>_OPB_retry	Input	x	OPB master retry
OPB_<*PortNum*>_OPB_timeout	Input	x	OPB timeout
OPB_<*PortNum*>_OPB_xferAck	Input	x	OPB master transfer acknowledge
OPB_<*PortNum*>_OPB_pendReqn	Input	x	OPB pending request
OPB_<*PortNum*>_OPB_RdDBus [0:C_OPB_DWIDTH-1]	Input	x	OPB master read data bus
OPB_<*PortNum*>_Sl_DBus [0:C_OPB_DWIDTH-1]	Output	x	OPB slave read data bus
OPB_<*PortNum*>_Sl_DBusEn	Output	0	OPB slave read data bus enable
OPB_<*PortNum*>_Sl_errAck	Output	0	OPB slave error acknowledge
OPB_<*PortNum*>_Sl_retry	Output	0	OPB slave retry
OPB_<*PortNum*>_Sl_toutSup	Output	0	OPB timeout suppress
OPB_<*PortNum*>_Sl_xferAck	Output	0	OPB slave transfer acknowledge
OPB_<*PortNum*>_OPB_ABus [0:C_OPB_AWIDTH-1]	Input	x	OPB slave address bus
OPB_<*PortNum*>_OPB_BE [0:3]	Input	x	OPB slave byte enables
OPB_<*PortNum*>_OPB_WrDBus [0:C_OPB_DWIDTH-1]	Input	x	OPB slave write data bus
OPB_<*PortNum*>_OPB_RNW	Input	x	OPB slave read not write
OPB_<*PortNum*>_OPB_select	Input	x	OPB slave select
OPB_<*PortNum*>_OPB_seqAddr	Input	x	OPB slave sequential address

Table 1-19: **OPB PIM Design Parameters**

Parameter	Allowable Values	Description
C_OPB_<*PortNum*>_ASEADDR	32-bit number	Base address for OPB bridge
C_OPB_<*PortNum*>_HIGHADDR	32-bit number	High address for OPB bridge
C_OPB_<*PortNum*>_MPMC2_TO_PI_CLK_RATIO	1, 2	MPMC2 to OPB clock ratio MPMC2:PIM 1 for 1:1 2 for 2:1
C_OPB_<*PortNum*>_BRIDGE_TO_PI_CLK_RATIO	1, 2	DSPLB to OPB clock ratio MPMC2:PIM 1 for 1:1 2 for 2:1

Note: All the parameter default values are set by the MPMC2 IP Configurator Interface

Performance Monitor (PM) I / O Signals

Table 1-20: **Performance Monitor I/O Signals (Port Interface = PM_S_DCR)**

Signal Name	Direction	Init Status	Description
PM_DCR_ABus[0:9]	Input	x	DCR address bus
PM_DCR_DBusIn[0:31]	Input	x	DCR data bus input
PM_DCR_Write	Input	x	DCR write request
PM_DCR_Read	Input	x	DCR read request
PM_DCR_Ack	Output	0	DCR acknowledge
PM_DCR_DBusOut[0:31]	Output	x	DCR data bus output
PM_DCR_ABus[0:9]	Input	x	DCR address bus

PLB Signals and Parameters

Table 1-21: **PLB PIM I/O Signals**

Signal Name	Direction	Init Status	Description
PLB_<*PortNum*>_CLK	Input	x	PLB clock
PLB_<*PortNum*>_RST	Input	x	PLB reset
PLB_<*PortNum*>_M_abort	Output	0	PLB master abort bus request indicator
PLB_<*PortNum*>_M_ABus [0:C_PLB_AWIDTH-1]	Output	x	PLB master address bus
PLB_<*PortNum*>_M_BE [0:7]	Output	0	PLB byte enables
PLB_<*PortNum*>_M_busLock	Output	0	PLB bus lock
PLB_<*PortNum*>_M_compress	Output	0	PLB compressed data transfer indicator
PLB_<*PortNum*>_M_guarded	Output	0	PLB guarded transfer indicator
PLB_<*PortNum*>_M_lockErr	Output	0	PLB lock error indicator
PLB_<*PortNum*>_M_mSize [0:1]	Output	1	PLB master data bus size
PLB_<*PortNum*>_M_ordered	Output	0	PLB synchronize transfer indicator
PLB_<*PortNum*>_M_priority [0:1]	Output	1	PLB transfer priority
PLB_<*PortNum*>_M_rdBurst	Output	0	PLB burst read transfer indicator
PLB_<*PortNum*>_M_request	Output	0	PLB master transfer request
PLB_<*PortNum*>_M_RNW	Output	0	PLB read not write
PLB_<*PortNum*>_M_size [0:3]	Output	0	PLB transfer size
PLB_<*PortNum*>_M_type [0:2]	Output	0	PLB transfer type
PLB_<*PortNum*>_M_wrBurst	Output	0	PLB burst write transfer indicator
PLB_<*PortNum*>_M_wrDBus[0:C_PLB_DWIDTH-1]	Output	x	PLB write data bus
PLB_<*PortNum*>_PLB_MAddrAck	Output	0	PLB master address acknowledge
PLB_<*PortNum*>_PLB_MBusy	Output	0	PLB master busy
PLB_<*PortNum*>_PLB_MErr	Output	0	PLB master error
PLB_<*PortNum*>_PLB_pendReq	Input	x	PLB pending bus request indicator
PLB_<*PortNum*>_PLB_pendPri [0:1]	Input	x	PLB pending request priority
PLB_<*PortNum*>_PLB_MRdBTerm	Input	x	PLB master read burst terminate
PLB_<*PortNum*>_PLB_MRdDAck	Input	x	PLB master read data acknowledge
PLB_<*PortNum*>_PLB_MRdDBus [0:C_PLB_DWIDTH-1]	Input	x	PLB master read data bus
PLB_<*PortNum*>_PLB_MRdWdAddr [0:3]	Input	x	PLB master read word address
PLB_<*PortNum*>_PLB_MRearbitrate	Input	x	PLB master rearbitrate
PLB_<*PortNum*>_PLB_reqPri [0:1]	Input	x	PLB current request priority
PLB_<*PortNum*>_PLB_MSSize [0:1]	Input	x	PLB master data bus size

Table 1-21: **PLB PIM I/O Signals *(Continued)***

Signal Name	Direction	Init Status	Description
PLB_<*PortNum*>_PLB_MWrBTerm	Input	x	PLB master write burst terminate
PLB_<*PortNum*>_PLB_MWrDAck	Input	x	PLB master write data acknowledge
PLB_<*PortNum*>_PLB_SlClk	Input	x	PLB reset
PLB_<*PortNum*>_PLB_SlRst	Input	x	PLB clock
PLB_<*PortNum*>_PLB_abort	Input	x	PLB abort bus request indicator
PLB_<*PortNum*>_PLB_ABus [0:C_PLB_AWIDTH-1]	Input	x	PLB address bus
PLB_<*PortNum*>_PLB_BE [0:7]	Input	x	PLB byte enables
PLB_<*PortNum*>_PLB_busLock	Input	x	PLB bus lock
PLB_<*PortNum*>_PLB_compress	Input	x	PLB compressed data transfer indicator
PLB_<*PortNum*>_PLB_guarded	Input	x	PLB guarded transfer indicator
PLB_<*PortNum*>_PLB_lockErr	Input	x	PLB lock error indicator
PLB_<*PortNum*>_PLB_masterID [0:C_PLB_MID_WIDTH-1]	Input	x	PLB current master indicator
PLB_<*PortNum*>_PLB_MSize	Input	x	PLB master data bus size
PLB_<*PortNum*>_PLB_ordered	Input	x	PLB synchronize transfer indicator
PLB_<*PortNum*>_PLB_PAValid	Input	x	PLB primary address valid indicator
PLB_<*PortNum*>_PLB_rdBurst	Input	x	PLB burst read transfer indicator
PLB_<*PortNum*>_PLB_rdPrim	Input	x	PLB secondary to primary read request indicator
PLB_<*PortNum*>_PLB_RNW	Input	x	PLB read not write
PLB_<*PortNum*>_PLB_SAValid	Input	x	PLB secondary address valid indicator
PLB_<*PortNum*>_PLB_size [0:3]	Input	x	PLB transfer size
PLB_<*PortNum*>_PLB_type [0:2]	Input	x	PLB transfer type
PLB_<*PortNum*>_PLB_wrBurst	Input	x	PLB burst write transfer indicator
PLB_<*PortNum*>_PLB_wrDBus [0:C_PLB_DWIDTH-1]	Input	x	PLB write data bus
PLB_<*PortNum*>_PLB_wrPrim	Input	x	PLB secondary to primary write request indicator
PLB_<*PortNum*>_Sl_addrAck	Output	0	PLB slave address acknowledge
PLB_<*PortNum*>_Sl_MErr [0:(C_PLB_NUM_MASTERS-1)]	Output	0	PLB slave error
PLB_<*PortNum*>_Sl_MBusy [0:(C_PLB_NUM_MASTERS-1)]	Output	0	PLB slave busy
PLB_<*PortNum*>_Sl_rdBTerm	Output	0	PLB slave read burst terminate
PLB_<*PortNum*>_Sl_rdComp	Output	0	PLB slave read transfer complete indicator

Table 1-21: **PLB PIM I/O Signals *(Continued)***

Signal Name	Direction	Init Status	Description
PLB_<*PortNum*>_Sl_rdDAck	Output	0	PLB slave read data acknowledge
PLB_<*PortNum*>_Sl_rdDBus [0:C_PLB_DWIDTH-1]	Output	x	PLB slave read bus
PLB_<*PortNum*>_Sl_rdWdAddr [0:3]	Output	x	PLB slave read word address
PLB_<*PortNum*>_Sl_rearbitrate	Output	0	PLB slave rearbitrate
PLB_<*PortNum*>_Sl_SSize [0:1]	Output	1	PLB slave data bus size
PLB_<*PortNum*>_Sl_wait	Output	0	PLB slave wait indicator
PLB_<*PortNum*>_Sl_wrBTerm	Output	0	PLB slave terminate write burst transfer
PLB_<*PortNum*>_Sl_wrComp	Output	0	PLB slave write transfer complete indicator
PLB_<*PortNum*>_Sl_wrDAck	Output	0	PLB slave write data acknowledge

Table 1-22: **PLB PIM Design Parameters**

Parameter	Allowable Values	Description
C_PLB_<*PortNum*>_RNG<*Relative_PortNum*>_BASEADDR	32-bit number	Base address for PLB bridge
C_PLB_<*PortNum*>_RNG<*Relative_PortNum*>_HIGHADDR	32-bit number	High address for PLB bridge
C_PLB_<*PortNum*>_MPMC2_TO_PI_CLK_RATIO	1, 2	MPMC2 to PLB clock ratio. MPMC2:PIM 1 for 1:1; 2 for 2:1
C_PLB_<*PortNum*>_BRIDGE_TO_PI_CLK_RATIO	1, 2	DSPLB to PLB clock ratio. MPMC2:PIM 1 for 1:1; 2 for 2:1

Note: All the parameter default values are set by the MPMC2 GUI.

XCL Signals and Parameters

The following tables list the XCL I/O signals and design parameters.

Table 1-23: **XCL PIM I/O Signals**

Signal Name	Direction	Init Status	Description
XCL_<*PortNum*>_CLK	Input	x	FSL input clock
XCL_<*PortNum*>_RST	Input	x	FSL reset
XCL_<*PortNum*>_Access_FSL_M_Clk	Input	x	FSL master clock
XCL_<*PortNum*>_Access_FSL_M_Write	Input	x	FSL master write enable signal
XCL_<*PortNum*>_Access_FSL_M_Data [0:C_XCL_<*PortNum*>_DATA_WIDTH-1]	Input	x	FSL master data bus
XCL_<*PortNum*>_Access_FSL_M_Control	Input	x	FSL master control signal
XCL_<*PortNum*>_Access_FSL_M_Full	Output	0	FSL master FIFO full
XCL_<*PortNum*>_Read_Data_FSL_S_Clk	Input	x	FSL slave input clock
XCL_<*PortNum*>_Read_Data_FSL_S_Read	Input	x	FSL slave read acknowledge
XCL_<*PortNum*>_Read_Data_FSL_S_Data	Output	x	FSL slave read data
XCL_<*PortNum*>_[0:C_XCL_<*PortNum*>_DATA_WIDTH-1]	Output	x	FSL slave data bus
XCL_<*PortNum*>_Read_Data_FSL_S_Control	Output	0	FSL slave control signal
XCL_<*PortNum*>_Read_Data_FSL_S_Exists	Output	0	FSL FIFO contains valid data

Table 1-24: **XCL Design Parameters**

Parameter	Allowable Values	Description
C_XCL_<*PortNum*>_PI_RDDATA_PIPELINE	0,1	0 = no extra pipeline on the read data path 1 = add extra pipeline register on the read data path to improve Fmax timing, but increases latency (default)
C_XCL_<*PortNum*>_PI_READONLY	0,1	0 = Read and Write capable XCL interface for data side XCL (default) Note: Compatible with instruction-side XCL, but might not be optimal. 1 = Read-only XCL interface optimized for instruction side XCL connection Note: `C_ALLOW_ICACHE_WR` parameter on Microblaze must be set to zero.
C_XCL_<*PortNum*>_PI__CACHE_LINE_LEN	4, 8	Cacheline length used by MicroBlaze processor 4 = 4-word cacheline size (default value) 8 = 8-word cacheline size

Design Implementation

Clock and Reset

The MPMC2 requires some system clock inputs, which are listed in Table 1-25, plus clock inputs for each PIM, as shown in the Figure 1-35, page 81.

Table 1-25: **System Clocks**

Clock Name	Description
MPMC2_0_Clk0(1)	The MPMC2 system clock with which the address path, control path, and data path operate. Also used to generate the memory clocks
MPMC2_0_Clk90	The MPMC2_0_Clk0 phase-shifted by 90 degrees.
MPMC2_0_Clk_200MHz	Required for Virtex-4 and Virtex-5 based designs. It is used to calibrate the IDELAYs and must be set to 200 MHz. Non Virtex-4 and Virtex-5 designs can tie the port to ground.
MPMC2_0_Clk0_2X	Reserved. This input can be tied to ground or left unconnected.
MPMC2_0_Clk_Cal	Reserved. This input can be tied to ground or left unconnected.
MPMC2_0_Clk_Mem	Reserved. This input can be tied to ground or left unconnected.
MPMC2_SlowestClk	Reserved. This input can be tied to ground or left unconnected.

Note:

1. Each PIM has at least one clock input. These clocks should be in phase with MPMC2_0_Clk0, but depending on the PIM specification could have a frequency that is an integer multiple.

Figure 1-35 is an example of how you can set up the clocking structure. You do not always require two digital clock managers (DCMs). In the example, a 100 MHz clock input is supplied and is used to generate a 200 MHz base clock. When two DCMs are used, a `dcm_por_reset` pcore can be used as shown. This core ensures that all elements are properly sequenced so that memory initialization sequences are performed after the correct application of clock and reset to the memory. Following best practices, no DCM has more than four global clock buffers (GCBs) used at a time. The `util_bus_split` logic splits the `sys_bus_reset` signal to reduce fanout. The `sys_bus_reset_1` signal is intended for the MPMC2, while the `sys_bus_reset_0` is for the rest of the system.

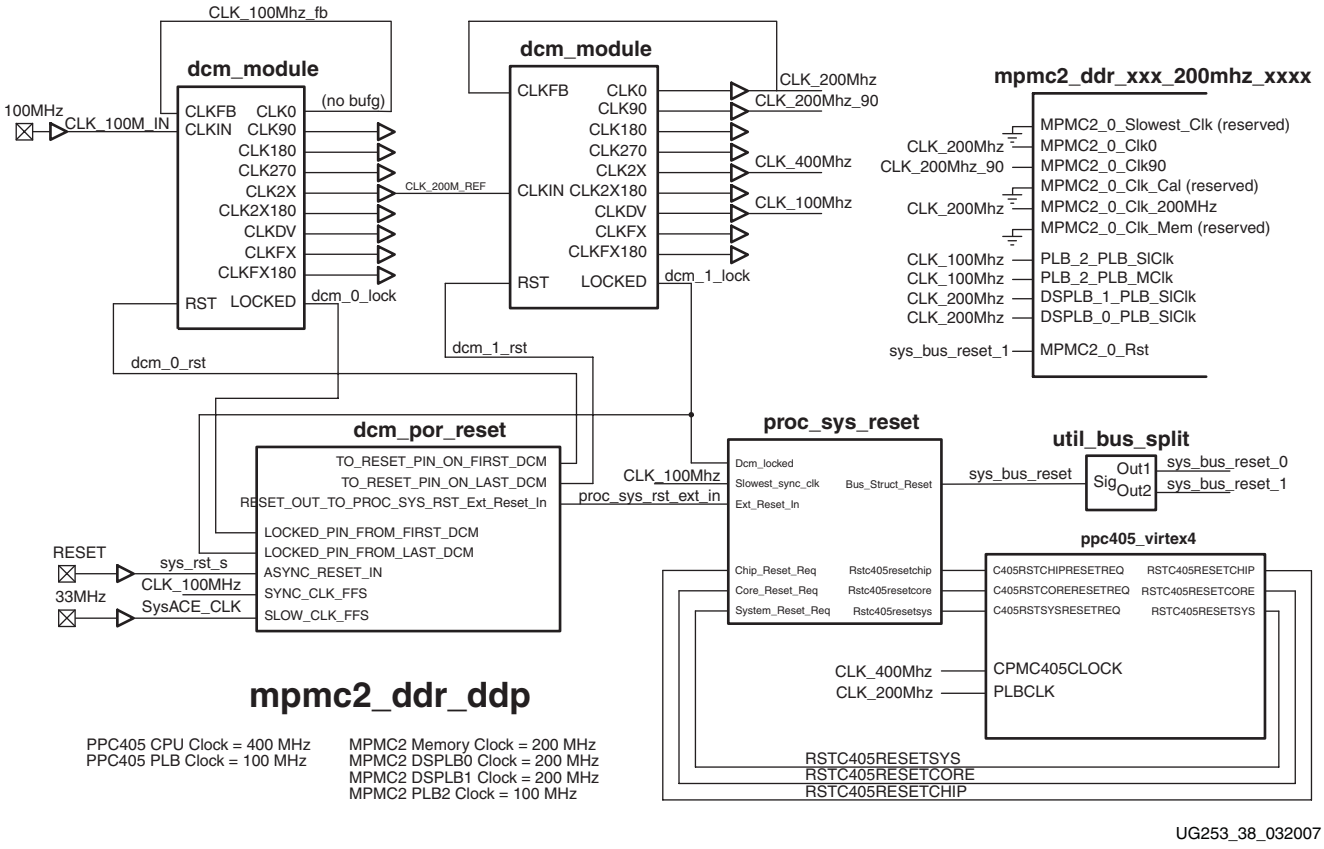


Figure 1-35: **Clock and Reset Structure**

Note: The clocks for PLB, OPB, and XCL PIMs are automatically connected up by the XPS tools by inferring the clock from the bus to which the PIM is connected. The NPI clock is taken from MPMC2_0_CLK0 internally.

Dragging and Dropping MPMC2 pcores in XPS Version 8.2 or Later

In Xilinx Platform Studio (XPS), pcores can be dragged from the IP Catalog over to the System Assembly View, then you can hook-up the required busses from that view. To drag and drop a core:

1. Create the core using the MPMC2 GUI (see the *MPMC2 IP Configurator GUI User Guide* for more details; "Additional Resources," page 13 contains a link to the document).
2. Either save or copy the generated pcore to a local project.
3. Open the project in XPS.
4. Under the IP Catalog tab, open the Project Repository; the core should be listed in the repository.
5. Drag and drop the pcore onto the System Assembly View window.
6. Connect the busses via XPS.
7. Make any other required edits either by directly editing the MHS (recommended) or via the XPS GUI. The MPMC2 reference designs provide working examples.

MPMC2 Required Parameter and Port Settings

The following sections outline the parameter and port settings required to complete the instantiation of a MPMC2 pcore in XPS.

MPMC2 Port Related Parameters

Each PIM has a set of parameters which must be set correctly for proper operation. The following are MHS code snippet examples of parameters that must be set.

ISPLB PIM

```
PARAMETER C_ISPLB_0_BASEADDR = 0x00000000
PARAMETER C_ISPLB_0_HIGHADDR = 0x03ffffff
PARAMETER C_ISPLB_0_MPMC2_TO_PI_CLK_RATIO = 1
```

DSPLB PIM

```
# Memory baseaddr
PARAMETER C_DSPLB_1_RNG0_BASEADDR = 0x00000000
# Memory highaddr
PARAMETER C_DSPLB_1_RNG0_HIGHADDR = 0x03ffffff
# baseaddr for first instance of PLB/OPB PIM
PARAMETER C_DSPLB_1_RNG1_BASEADDR = 0x80000000
# highaddr for first instance of PLB/OPB PIM
PARAMETER C_DSPLB_1_RNG1_HIGHADDR = 0x8fffffff
# baseaddr for second instance of PLB/OPB PIM
PARAMETER C_DSPLB_1_RNG2_BASEADDR = 0x90000000
# highaddr for second instance of PLB/OPB PIM
PARAMETER C_DSPLB_1_RNG2_HIGHADDR = 0x9fffffff
PARAMETER C_DSPLB_1_MPMC2_TO_PI_CLK_RATIO = 1
```

PLB PIM

```
PARAMETER C_PLB_2_BASEADDR = 0x00000000
PARAMETER C_PLB_2_HIGHADDR = 0x03ffffff
PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 1
PARAMETER C_PLB_2_BRIDGE_TO_PI_CLK_RATIO = 1
```

OPB PIM

```
PARAMETER C_OPB_0_BASEADDR = 0x10000000
PARAMETER C_OPB_0_HIGHADDR = 0x13FFFFFF
PARAMETER C_OPB_0_MPMC2_TO_PI_CLK_RATIO = 1
PARAMETER C_OPB_0_BRIDGE_TO_PI_CLK_RATIO = 1
```

CDMAC PIM

```
PARAMETER C_CDMAC_3_BASEADDR = 0x00000000
PARAMETER C_CDMAC_3_HIGHADDR = 0x07FFFFFF
PARAMETER C_CDMAC_3_DCR_BASEADDR = 0b0101000000
PARAMETER C_CDMAC_3_DCR_HIGHADDR = 0b0101111111
PARAMETER C_CDMAC_3_MPMC2_TO_PI_CLK_RATIO = 2
```

XCL PIM

```
PARAMETER C_XCL_0_BASEADDR = 0x10000000
PARAMETER C_XCL_0_HIGHADDR = 0x17FFFFFF
```

Note: The XCL PIM does not require an MPMC2_TO_PI_CLK_RATIO setting because the ratio is automatically detected by logic in the PIM.

NPI PIM

```
PARAMETER C_MPMC2_PIM_0_BASEADDR = 0x00000000
PARAMETER C_MPMC2_PIM_0_HIGHADDR = 0x07FFFFFF
```

Performance Monitors (If Enabled)

```
PARAMETER C_PM_DCR_BASEADDR = 0b1000100000
PARAMETER C_PM_DCR_HIGHADDR = 0b1000101111
```

ECC (If Enabled)

```
PARAMETER C_ECC_BASEADDR = 0b1001000000
PARAMETER C_ECC_HIGHADDR = 0b1001001111
```

Parameter Classes and Settings

There are three classes of parameter information that must be set correctly:

- RATIO
- BASEADDR and HIGHADDR
- RNGx

The following subsections describe these Parameters and their settings.

RATIO

Each RATIO parameter has a specific function to ensure that the port clock ratios are set correctly. The ratios imply a specific clock relationship that must be properly expressed in the port declarations, and the DCMs connections (see the section on "Clock and Reset," page 80).

The ratio of the port interface to MPMC2 can be set at 1:1 or 1:2.
The C_PLB_2_MPMC2_TO_PI_CLK_RATIO parameter controls this relationship which in the following case is set at 1:1

```
PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 1
```

To set the port to run at half the frequency of the memory, (normal), the parameter is set as follows:

```
PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 2
```

The following parameter applies to the PLB PIM only when it is used as a bridge from the DSPLB:

```
PARAMETER C_PLB_2_BRIDGE_TO_PI_CLK_RATIO = 1
```

The C_PLB_2_BRIDGE_TO_PI_CLK_RATIO sets the ratio of the DSPLB operating frequency to how fast the PLB operates. This parameter can be set to 1:1 or 1:2.
When set to "2", the 2:1 ratio is employed and the PLB operates at one half the frequency of the DSPLB PIMs.

Note: ***Be careful*** with these ratios: if a DSPLB is used, all OPB and PLB PIMs must run at the same frequency between the Master and Slave ports for each PIM.

BASEADDR and HIGHADDR

The BASEADDR and HIGHADDR parameters set the address locations to which the ports will have access to the memory. For example:

```
PARAMETER C_PLB_2_BASEADDR = 0x00000000
PARAMETER C_PLB_2_HIGHADDR = 0x03ffffff
```

These parameters specify the addresses a particular port will use to access memory.

It is important to understand these addresses and their operations. Failure to do so could result in a design that does not successfully complete XPS, or fails to operate as expected in hardware. The entire system memory map must be considered to ensure that operation is as intended.

Note: MPMC2 hardware allows operations which are not currently permitted by XPS such as non-power of two addressing.

You must ensure that the memory map allows each port to have the expect access in memory space. Generally, specifying the same BASEADDR and HIGHADDR for all ports will ensure that result.

RNGx

The RNGx parameters specify the DSPLB to memory address range and the address ranges of the DSPLB to PLB bridges which are built into the PLB PIM(s), and connected to the DSPLB PIM(s). The RNGx parameters require particular attention.

RNGx values for a specific DSPLB PIM cannot overlap the other BASEADDR and HIGHADDR ranges associated with that PIM.

Because of how XPS models handle addressing, the RNGx parameters apply to the DSPLB PIM (which hooks to the DSPLB on the PowerPC 405 processor) and *not* to the PLB PIM to which it applies. Further, the "x" in the RNGx does not inherently apply to the PLB PIMs instance order.

The following examples are provided:

- In an IDPP MPMC2 pcore:
  - RNG0 would refer to the memory
  - RNG1 would refer to PLB_2
  - RNG2 would refer to PLB_3
- In a PIDP MPMC2 pcore:
  - RNG0 would refer to the memory
  - RNG1 would refer to PLB_0
  - RNG2 would refer to PLB_3
- In a PPDI MPMC2 pcore:
  - RNG0 would refer to the memory
  - RNG1 would refer to PLB_0
  - RNG2 would refer to PLB_1

The RNGx information sets the window for DSPLB to allow that PIM to initiate transactions on the PLB PIM referenced by the RNGx.

The following is an example of the DSPLB2PLB or DSPLB2OPB RNGx parameters:

```
PARAMETER C_DSPLB_1_RNG1_BASEADDR = 0x80000000
PARAMETER C_DSPLB_1_RNG1_HIGHADDR = 0x8fffffff
PARAMETER C_DSPLB_1_RNG2_BASEADDR = 0x90000000
PARAMETER C_DSPLB_1_RNG2_HIGHADDR = 0x9fffffff
```

You must ensure that these addresses are set so the DSPLB can access the devices on the PLB; otherwise, the design could fail XPS or not work as expected in hardware.

Note: When using two PowerPC 405 processors, the RNG parameters for the 2 DSPLB PIMs should be set to identical values, with the possible exception of RNG0.

Note: The OPB PIM has a DSPLB 2 OPB bridge. The same principles described above apply if one or all of the PLB PIMs are replaced by OPB PIMs.

MPMC2 PIM Related Ports

Each PIM has an associated set of ports in the MHS. The MPMC2 core has ports associated with the memory interface. Besides the PIM ports, MPMC2 has the following port types:

- RESET
- EXTERNAL
- CLOCK

Note: Each required user port connection has a require tag that forces an error in XPS if the port is not contained in the MHS.

RESET Port

The RESET port is:

```
PORT MPMC2_0_Rst = sys_bus_reset_1
```

DDR and DDR2 EXTERNAL Ports:

```
PORT MPMC2_0_DDR_Clk_O = DDR_Clk
PORT MPMC2_0_DDR_Clk_n_O = DDR_Clkn
PORT MPMC2_0_DDR_CE_O = DDR_CKE
PORT MPMC2_0_DDR_BankAddr_O = DDR_BA
PORT MPMC2_0_DDR_Addr_O = DDR_Addr
PORT MPMC2_0_DDR_CS_n_O = DDR_CSn
PORT MPMC2_0_DDR_RAS_n_O = DDR_RASn
PORT MPMC2_0_DDR_CAS_n_O = DDR_CASn
PORT MPMC2_0_DDR_WE_n_O = DDR_WEn
PORT MPMC2_0_DDR_DM_O = DDR_DM
PORT MPMC2_0_DDR_DQS = DDR_DQS
PORT MPMC2_0_DDR_DQ = DDR_DQ
PORT MPMC2_0_DDR_DQS_DIV_O =DDR_DQS_DIV_O (Spartan 3 only)
PORT MPMC2_0_DDR_DQS_DIV_I =DDR_DQS_DIV_I (Spartan 3 only)
```

```
PORT MPMC2_0_DDR2_Clk_O = DDR2_Clk
PORT MPMC2_0_DDR2_Clk_n_O = DDR2_Clkn
PORT MPMC2_0_DDR2_CE_O = DDR2_CKE
PORT MPMC2_0_DDR2_CS_n_O = DDR2_CSn
PORT MPMC2_0_DDR2_ODT_O = DDR2_ODT
PORT MPMC2_0_DDR2_RAS_n_O = DDR2_RASn
PORT MPMC2_0_DDR2_CAS_n_O = DDR2_CASn
PORT MPMC2_0_DDR2_WE_n_O = DDR2_WEn
PORT MPMC2_0_DDR2_BankAddr_O = DDR2_BA
PORT MPMC2_0_DDR2_Addr_O = DDR2_Addr
PORT MPMC2_0_DDR2_DQ = DDR2_DQ
PORT MPMC2_0_DDR2_DM_O = DDR2_DM
PORT MPMC2_0_DDR2_DQS = DDR2_DQS
PORT MPMC2_0_DDR2_DQS_n = DDR2_DQSn
PORT MPMC2_0_DDR2_DQS_DIV_O =DDR_DQS_DIV_O (Spartan 3 only)
PORT MPMC2_0_DDR2_DQS_DIV_I =DDR_DQS_DIV_I (Spartan 3 only)
```

Review the port declarations for the pins in the UCF file carefully; ensure that A0 is really A0 and An is really An. The current XPS IP cores might have their busses tied as [0:n], whereas MPMC2 always uses [n:0]. The [n:0] convention corresponds 1:1 to Xilinx Board schematics.

MPMC2 Required Clocks

The port declarations for clocks require special understanding. You can look at the reference designs to see port connection examples also. The following code snippet lists the required clock ports.

```
PORT MPMC2_0_Clk0 = CLK_100MHz
PORT MPMC2_0_Clk90 = CLK_100MHz_90
```

Set the easiest clocks first:

- `MPMC2_0_Clk_200MHz = CLK_200MHz`
- `MPMC2_Slowest_Clk = net_gnd` (or left unconnected)
- `MPMC2_0_Clk0_2X = net_gnd` (or left unconnected)
- `MPMC2_0_Clk_Mem = net_gnd` (or left unconnected)
- `MPMC2_0_CLK_CAL = net_gnd` (or left unconnected)

200 MHz Clock Port

:

```
PORT MPMC2_0_Clk_200MHz = CLK_200MHz
```

This clock must be set to 200 MHz (for **Virtex-4** and **Virtex-5** only; used by IDELAY elements).

The next set of clock pins must be set for MPMC2 to be clocked properly.

Virtex-4 Clock Memory at 100 MHz

For designs that clock the memory at 100 MHz in Virtex-4, use:

```
PORT MPMC2_0_Clk0 = CLK_100MHz
PORT MPMC2_0_Clk90 = CLK_100MHz_90
```

200 MHz Clock Memory

When the memory is to be clocked at 200 MHz, use:

```
PORT MPMC2_0_Clk0 = CLK_200MHz
PORT MPMC2_0_Clk90 = CLK_200MHz_90
```

PIM-Type Required Clocks, Reset, or Interrupt Connections

Each PIM type has a set of required clocks, reset, or interrupt port connections.

- For ISPLB, DSPLB, PLB, and OPB, the clock connection is inferred from the bus.
- For XCL, ISPLB, DSPLB, PLB, and OPB, the reset connection is inferred from the bus.
- PIM resets are optional. If not specified, PIM resets come from the MPMC2 main reset port.

The following are specific settings for CDMAC and XCL PIMs:

CDMAC PIM

```
PORT CDMAC_3_Clk = CLK_100MHz
PORT CDMAC_3_Rst = sys_bus_reset_1 (optional)
PORT CDMAC_3_CDMAC_INT = CDMAC_INT (Indicates an interrupt has occurred in
the CDMAC PIM)
```

XCL PIM

The XCL PIM clock connection is inferred from XCL.

```
   PORT XCL_4_Rst = sys_bus_reset_1 (optional)
```

Example MPMC2_DDR2_ddpppp Instance

The following is an example MHS instance for MPMC2 with DDR2 memory, two DSPLB PIMs and four PLB PIMs. In the example, the DDR Memory is clocked at 200 MHz while the DSPLBs and PLBs are clocked at 100 MHz.

Example: MPMC2_DDR2_ddppp

```
BEGIN mpmc2_ddr2_ddpppp_200mhz_x32_w1d32m32r8a_5a
 PARAMETER INSTANCE = mpmc2_ddr2_ddpppp_200mhz_x32_w1d32m32r8a_5a_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_DSPLB_0_RNG0_BASEADDR = 0x00000000
 PARAMETER C_DSPLB_0_RNG0_HIGHADDR = 0x07FFFFFF
 PARAMETER C_DSPLB_0_RNG1_BASEADDR = 0x80000000
 PARAMETER C_DSPLB_0_RNG1_HIGHADDR = 0x8FFFFFFF
 PARAMETER C_DSPLB_0_RNG2_BASEADDR = 0xFFFF4000
 PARAMETER C_DSPLB_0_RNG2_HIGHADDR = 0xFFFF7FFF
 PARAMETER C_DSPLB_0_RNG3_BASEADDR = 0xFFFF8000
 PARAMETER C_DSPLB_0_RNG3_HIGHADDR = 0xFFFFBFFF
 PARAMETER C_DSPLB_0_RNG4_BASEADDR = 0xFFFFC000
 PARAMETER C_DSPLB_0_RNG4_HIGHADDR = 0xFFFFFFFF
 PARAMETER C_DSPLB_0_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_DSPLB_1_RNG0_BASEADDR = 0x00000000
 PARAMETER C_DSPLB_1_RNG0_HIGHADDR = 0x07FFFFFF
 PARAMETER C_DSPLB_1_RNG1_BASEADDR = 0x80000000
 PARAMETER C_DSPLB_1_RNG1_HIGHADDR = 0x8FFFFFFF
 PARAMETER C_DSPLB_1_RNG2_BASEADDR = 0xFFFF4000
 PARAMETER C_DSPLB_1_RNG2_HIGHADDR = 0xFFFF7FFF
 PARAMETER C_DSPLB_1_RNG3_BASEADDR = 0xFFFF8000
 PARAMETER C_DSPLB_1_RNG3_HIGHADDR = 0xFFFFBFFF
 PARAMETER C_DSPLB_1_RNG4_BASEADDR = 0xFFFFC000
 PARAMETER C_DSPLB_1_RNG4_HIGHADDR = 0xFFFFFFFF
 PARAMETER C_DSPLB_1_MPMC2_TO_PI_CLK_RATIO = 2

 PARAMETER C_PLB_2_BASEADDR = 0x00000000
 PARAMETER C_PLB_2_HIGHADDR = 0x07FFFFFF
 PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_PLB_2_BRIDGE_TO_PI_CLK_RATIO = 1
 PARAMETER C_PLB_3_BASEADDR = 0x00000000
 PARAMETER C_PLB_3_HIGHADDR = 0x07FFFFFF
 PARAMETER C_PLB_3_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_PLB_3_BRIDGE_TO_PI_CLK_RATIO = 1
 PARAMETER C_PLB_4_BASEADDR = 0x00000000
 PARAMETER C_PLB_4_HIGHADDR = 0x07FFFFFF
 PARAMETER C_PLB_4_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_PLB_4_BRIDGE_TO_PI_CLK_RATIO = 1
 PARAMETER C_PLB_5_BASEADDR = 0x00000000
 PARAMETER C_PLB_5_HIGHADDR = 0x07FFFFFF
 PARAMETER C_PLB_5_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_PLB_5_BRIDGE_TO_PI_CLK_RATIO = 1
```

Example: MPMC2_DDR2_ddppp ***(Continued)***

```
BUS_INTERFACE DSPLB_0 = plb_0
BUS_INTERFACE DSPLB_1 = plb_1
BUS_INTERFACE PLB_S_2 = plb_2
BUS_INTERFACE PLB_M_2 = plb_2
BUS_INTERFACE PLB_S_3 = plb_3
BUS_INTERFACE PLB_M_3 = plb_3
BUS_INTERFACE PLB_S_4 = plb_4
BUS_INTERFACE PLB_M_4 = plb_4
BUS_INTERFACE PLB_S_5 = plb_5
BUS_INTERFACE PLB_M_5 = plb_5

PORT DSPLB_0_PLB_SlClk = clk_100MHz
PORT DSPLB_1_PLB_SlClk = clk_100MHz
PORT PLB_2_PLB_SlClk = clk_100MHz
PORT PLB_2_PLB_MClk = clk_100MHz
PORT PLB_3_PLB_SlClk = clk_100MHz
PORT PLB_3_PLB_MClk = clk_100MHz
PORT PLB_4_PLB_SlClk = clk_100MHz
PORT PLB_4_PLB_MClk = clk_100MHz
PORT PLB_5_PLB_SlClk = clk_100MHz
PORT PLB_5_PLB_MClk = clk_100MHz
PORT MPMC2_0_Rst = sys_bus_reset0
PORT MPMC2_0_Clk0 = clk_200MHz
PORT MPMC2_0_Clk90 = clk_200MHz_90
PORT MPMC2_0_Clk_200MHz = clk_200MHz
PORT MPMC2_0_DDR2_Clk_O = DDR2_Clk
PORT MPMC2_0_DDR2_Clk_n_O = DDR2_Clkn
PORT MPMC2_0_DDR2_CE_O = DDR2_CKE
PORT MPMC2_0_DDR2_CS_n_O = DDR2_CSn
PORT MPMC2_0_DDR2_ODT_O = DDR2_ODT
PORT MPMC2_0_DDR2_RAS_n_O = DDR2_RASn
PORT MPMC2_0_DDR2_CAS_n_O = DDR2_CASn
PORT MPMC2_0_DDR2_WE_n_O = DDR2_WEn
PORT MPMC2_0_DDR2_BankAddr_O = DDR2_BA
PORT MPMC2_0_DDR2_Addr_O = DDR2_Addr
PORT MPMC2_0_DDR2_DQ = DDR2_DQ
PORT MPMC2_0_DDR2_DM = DDR2_DM
PORT MPMC2_0_DDR2_DQS = DDR2_DQS
PORT MPMC2_0_DDR2_DQS_n = DDR2_DQSn
END
```

Example MPMC2_DDR_ddp Instance

This example implements MPMC2 with two DSPLB PIMs that operate at 200 MHz and connect directly to the PowerPC 405 processor, and a PLB PIM that operates at 100 MHz. The DDR Memory is clocked at 200 MHz in this snippet.

Example: MPMC2_DDR_ddp

```
BEGIN mpmc2_ddr_ddp_200mhz_x16_hyb25d512160be_5
 PARAMETER INSTANCE = mpmc2_ddr_ddp_200mhz_x16_hyb25d512160be_5_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_DSPLB_0_RNG0_BASEADDR = 0x00000000
 PARAMETER C_DSPLB_0_RNG0_HIGHADDR = 0x07FFFFFF
 PARAMETER C_DSPLB_0_RNG1_BASEADDR = 0x80000000
 PARAMETER C_DSPLB_0_RNG1_HIGHADDR = 0xFFFFFFFF
 PARAMETER C_DSPLB_0_MPMC2_TO_PI_CLK_RATIO = 1
 PARAMETER C_DSPLB_1_RNG0_BASEADDR = 0x00000000
 PARAMETER C_DSPLB_1_RNG0_HIGHADDR = 0x07FFFFFF
 PARAMETER C_DSPLB_1_RNG1_BASEADDR = 0x80000000
 PARAMETER C_DSPLB_1_RNG1_HIGHADDR = 0xFFFFFFFF
 PARAMETER C_DSPLB_1_MPMC2_TO_PI_CLK_RATIO = 1
 PARAMETER C_PLB_2_BASEADDR = 0x00000000
 PARAMETER C_PLB_2_HIGHADDR = 0x03FFFFFF
 PARAMETER C_PLB_2_MPMC2_TO_PI_CLK_RATIO = 2
 PARAMETER C_PLB_2_BRIDGE_TO_PI_CLK_RATIO = 2
 BUS_INTERFACE DSPLB_0 = plb_0
 BUS_INTERFACE DSPLB_1 = plb_1
 BUS_INTERFACE PLB_S_2 = plb_2
 BUS_INTERFACE PLB_M_2 = plb_2
 PORT MPMC2_0_Rst = sys_bus_reset_1
 PORT MPMC2_0_Clk0 = CLK_200MHz
 PORT MPMC2_0_Clk90 = CLK_200MHz_90
 PORT MPMC2_0_Clk_200MHz = CLK_200MHz
 PORT MPMC2_0_DDR_Clk_O = DDR_Clk
 PORT MPMC2_0_DDR_Clk_n_O = DDR_Clkn
 PORT MPMC2_0_DDR_CE_O = DDR_CKE
 PORT MPMC2_0_DDR_BankAddr_O = DDR_BA
 PORT MPMC2_0_DDR_Addr_O = DDR_Addr
 PORT MPMC2_0_DDR_CS_n_O = DDR_CSn
 PORT MPMC2_0_DDR_RAS_n_O = DDR_RASn
 PORT MPMC2_0_DDR_CAS_n_O = DDR_CASn
 PORT MPMC2_0_DDR_WE_n_O = DDR_WEn
 PORT MPMC2_0_DDR_DM_O = DDR_DM
 PORT MPMC2_0_DDR_DQS = DDR_DQS
 PORT MPMC2_0_DDR_DQ = DDR_DQ
End
```

Using the Base System Builder

The Base System Builder (BSB) in Xilinx Platform Studio (XPS) automates several basic hardware and software platform configuration tasks common to most processor designs. When adding an MPMC2 pcore to a BSB design, you should first study how the MHS file is built in the provided reference systems. One MHS file can be copied to another, but the parameters must be updated correctly. After building an MPMC2 pcore and placing it in a local project pcore directory, follow these general guidelines to add the pcore to a BSB design.

Add an MPMC2 Pcore to an XPS Project

1. Double-click the `system.xmp` file in the BSB-created project folder.
2. From the IP Catalog tab, click + for Project Repository.
3. Drag the MPMC2_DDR*xxxxxx* into the System Assembly window.
4. Click the Windows **Close** button in the upper right corner to close XPS.

Edit the UCF

Edit the User Constraints File (UCF) to swap the bit order of the DDR_ADDR, Data, DQS and DM signals. The BSB reverses the bit order compared to MPMC2 PLB, OPB, or MCH memory controllers.

Edit the MHS File

1. In XPS, drop the pcore onto the System Assembly View pane.
2. Fix the top-level port declarations to use MPMC2.
   a. Copy existing PLB_Port declarations.
   b. Comment out one copy to preserve the ability to restore the PLB_DDR instance.
   c. Change the other copy to use MPMC2 port names.
      - Change the right side equal to the MPMC2 name.
      - Change all the vectors from [0:n] to [n:0].
3. Edit DCM names and add clocks.
   a. Name change for clarity.
   b. Name change *must* be global in MHS.
4. Comment out the existing PLB_DDR instance.
   a. Comment out the instance itself.
   b. Comment out the util_vector_logic for sysclk_inv.
   c. Comment out the util_vector_logic for clk90_inv.
   d. Comment out the util_vector_logic for ddr_clk90_inv.
5. Copy the MPMC2 instance as a backup.
   a. Copy the MPMC2 instance at the bottom of the MHS file.
   b. Comment out one copy for safe keeping.
6. Change the MPMC2 required parameters.

7. Add the MPMC2 required ports.
   a. DDR PORT pins (copy these from one of the provided projects).
   b. PIM PORT clock pins (copy these from one of the provided projects and modify as needed).

Edit the MSS File

1. Comment out the PLB_DDR instance in the Microprocessor Software Specification (MSS) file.

Build the Project

1. Open XPS.
2. Build the project, taking care of any warnings or errors.

PIM Implementation Note - Usage of ISPLB and DSPLB

Some systems are designed to have the ISPLB read code across the PLB from an EEPROM or memory other than DDR memory. In such systems, a DSPLB PIM can be used instead of an ISPLB PIM. This allows the ISPLB of the PowerPC 405 processor to access anywhere on any of the PLB PIMs. Because there is a limit of only two DSPLBs in an MPMC2 core, this option is not available if two PowerPC 405 processors are used.

XILINX®

Appendix A

Communication Direct Memory Access Controller (CDMAC)

Overview

The CDMAC is designed to provide high-performance DMA for streaming data. Many communication systems utilize point-to-point interconnections because the data is unidirectional, and requires little protocol. The CDMAC provides a receive data channel and a transmit data channel. This permits a full duplex communication device to have data movement via DMA. The CDMAC uses two LocalLink interfaces to communicate with up to two devices. The back end of the CDMAC is designed to connect to one MPMC2 port. The MPMC2 interface is sufficiently generic that the CDMAC could be used stand-alone for other applications. The CDMAC also uses the IBM CoreConnect® DCR bus for command and status control.

This appendix provides the following subsections:

Features

The CDMAC has the following features:

- 128-byte bursts from memory for data get and put, 32-byte bursts for gathering DMA descriptors
- Two channels of DMA controlling two LocalLink interfaces: one transmit and one receive
- Direct plug-in to the Multi-Port MPMC2
- Interruptible and stoppable DMA engines on per descriptor basis
- DMA engines broadcast application specific data across the LocalLink interfaces
- Intelligent engine arbitration built-in
- Software error detection for DMA transactions
- Simple software use model
- Low FPGA device area overhead
- Designed to be extensible to eight engines without software change

Related Documents

The *LocalLink Specification* and *IBM CoreConnect Device Control Register Bus: Architecture Specification* provides additional information. The section "Additional Resources," page 13, provides links to these documents.

High-Level Block Diagram

Figure A-1 illustrates a high-level block diagram of the CDMAC structure. The CDMAC uses an MPMC2 port interface, two LocalLink interfaces, and a DCR interface (not shown). The MPMC2 port interface connects the CDMAC into the personality module of the MPMC2 interface. The two LocalLink interfaces provide a full duplex LocalLink device access to the CDMAC. There is a transmit (Tx) LocalLink interface and a receive (Rx) LocalLink interface. The DCR interface allows the CPU to interact with the CDMAC for initiating DMA processes or status gathering.

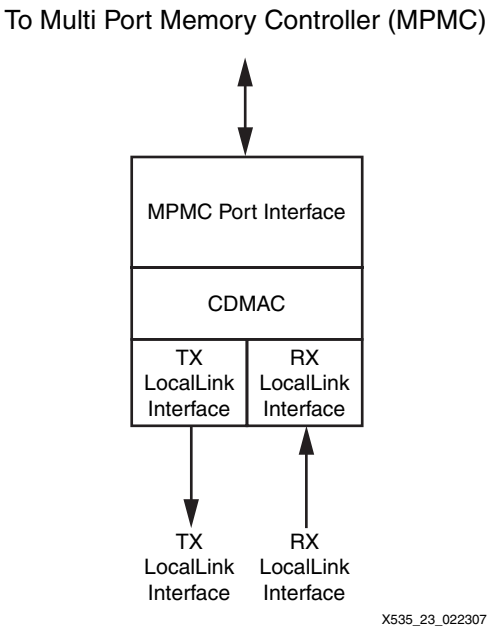


Figure A-1: **CDMAC High-Level Block Diagram**

The CDMAC simplifies the software requirements for DMA operations and provides features that simplify the software device driver and reduce the requirement of CPU interactions. While DMA itself relieves the CPU of having to move data, and thus increases the effective CPU availability, the CDMAC further streamlines this process by offering the CPU easy control and access to DMA operations.

The CDMAC has configurable options at instantiation, so you can decide if the DMA descriptors must be "scrubbed" by the CPU before being reused by the CDMAC. "Scrubbing" of DMA descriptors is the process of updating the fields of the descriptor so that they can be reused by the CDMAC.

For example:

- The LocalLink TFT controller is a continuously active repetitive device. It does not require that the CPU service the DMA engine, once it has been set up.
- The LocalLink GMAC peripheral requires that the CPU scrub the DMA descriptors before they are reused.

By providing control over these areas, the CDMAC maximizes the amount of CPU that is left to process elements other than the DMA engines. This leads to a non-obvious substantial benefit in CPU performance.

The CDMAC is designed to connect to communication devices. It is not intended to be a generic DMA controller. As such, it does not provide nor need an address interface. Instead of an address interface CDMAC uses a streaming data-centric interface. This interface is typical of full duplex communication systems.

The GMAC peripheral is an example of a typical full duplex communication system. The GMAC peripheral must be capable of simultaneous data transmission and reception. GMAC uses a unidirectional streaming data bus from GMAC peripheral to CDMAC for receive while using a unidirectional streaming data bus in the opposite direction for transmit. The communication system does not provide any form of address; it simply provides data and a context of the data that allows the data to be properly framed.

One important advantage of the CDMAC architecture is that intelligent processing can be added between the CDMAC and the LocalLink device. Consider the case in which a core is built that has various processing capabilities. These capabilities can be added via LocalLink to LocalLink interfaces and inserted in an appropriate order between the CDMAC and the final LocalLink device. This permits you to choose how much area you are willing to sacrifice to achieve a specific level of performance. If more performance is needed, more processing blocks can be instantiated. These blocks are generic because the LocalLink protocol is used.

Theory of Operation

The theory of operation covers the following topics:

- "Communication DMA"
- "DMA Process"
- "DMA Descriptor Model"

Communication DMA

Modern communication systems typically rely upon unidirectional data transport mechanisms. These links allow for streaming data to be sent across standardized interfaces. Typical systems have line cards that are aggregated together to form a large amount of streaming data. Often this data has to be contextually switched between various points to route the data between its origin and its destination. Between these route points, the data is often aggregated into very fast data streams. The CDMAC assists in the data movement.

Communication DMA refers to moving large quantities of data between the demarcation point and main system memory in a processor-based system. The communication DMA does not imply that the processor consumes the data. In fact, in some systems, the processor never touches the data, but the data is consumed by another DMA device instead. In high-data-bandwidth systems, the processor generally handles only the administrative functions, such as set up and tear down, rather than actively handling the data.

The CDMAC provides an interface between the MPMC2 and two independent channels of DMA using LocalLink interface. Figure A-1, page 96 shows the high-level diagram view of the CDMAC, and illustrates the two LocalLink interfaces and the port interface to the MPMC2. The CDMAC provides one transmit and one receive channel. Each channel uses the XILINX® *LocalLink Interface* specification (a link to this document is in "Additional Resources," page 13). The LocalLink interfaces are on one side of the CDMAC and MPMC2 port interface is on the other side.

Figure A-2 shows a simplified block diagram that illustrates the major functional elements of the CDMAC. See the "CDMAC Architecture," page 109, for more information on the internals of the CDMAC.

.

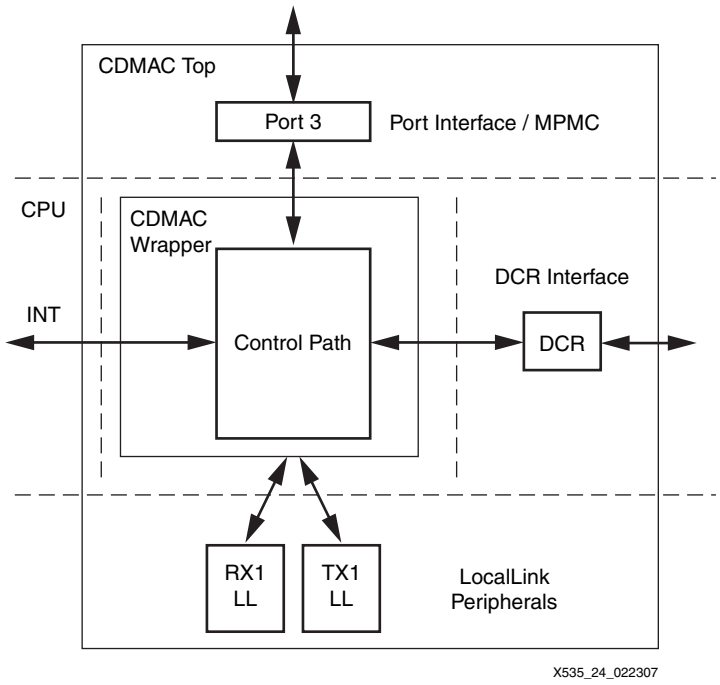


Figure A-2: **CDMAC Top Level Block Diagram**

The CDMAC offers a wide variety of features to augment communication style interfaces. Communication style interfaces differ from classical DMA because they provide structural control over the data. For example, a communication system typically needs to packetize its data in to allow for transmission and reception errors. In classical DMA there is no need to packetize the data because the device is directly consuming the data and errors are effectively impossible. The CDMAC differs from classical DMA controllers primarily because it supports packetized data.

The CDMAC also offers other important mechanisms that make communication systems easier to implement. The CDMAC provides the ability to dynamically control the context of each engine through the DMA descriptors. These descriptors provide:

- Ability to transmit and receive application unique data across the LocalLink interfaces directly to and from the descriptors
- Buffer context
- CDMAC status
- Control context by interrupts
- Engine halting

These features provide for substantially simpler software interfaces, and less processor intervention to support the DMA transactions.

DMA Process

There are three main levels to how the CDMAC handles data movement:

- the DMA process
- the Transmits (Tx)
- the Receives (Rx)

The highest level is the DMA process. The DMA process can be thought of as the execution of an entire chain of DMA descriptors to completion. DMA transfers in turn become individual MPMC2 operations such as 8-word, cache-line reads, 128-byte burst reads, 128-byte burst writes, or 8-word, cache-line writes.

Figure A-6 illustrates how CDMAC handles DMA.

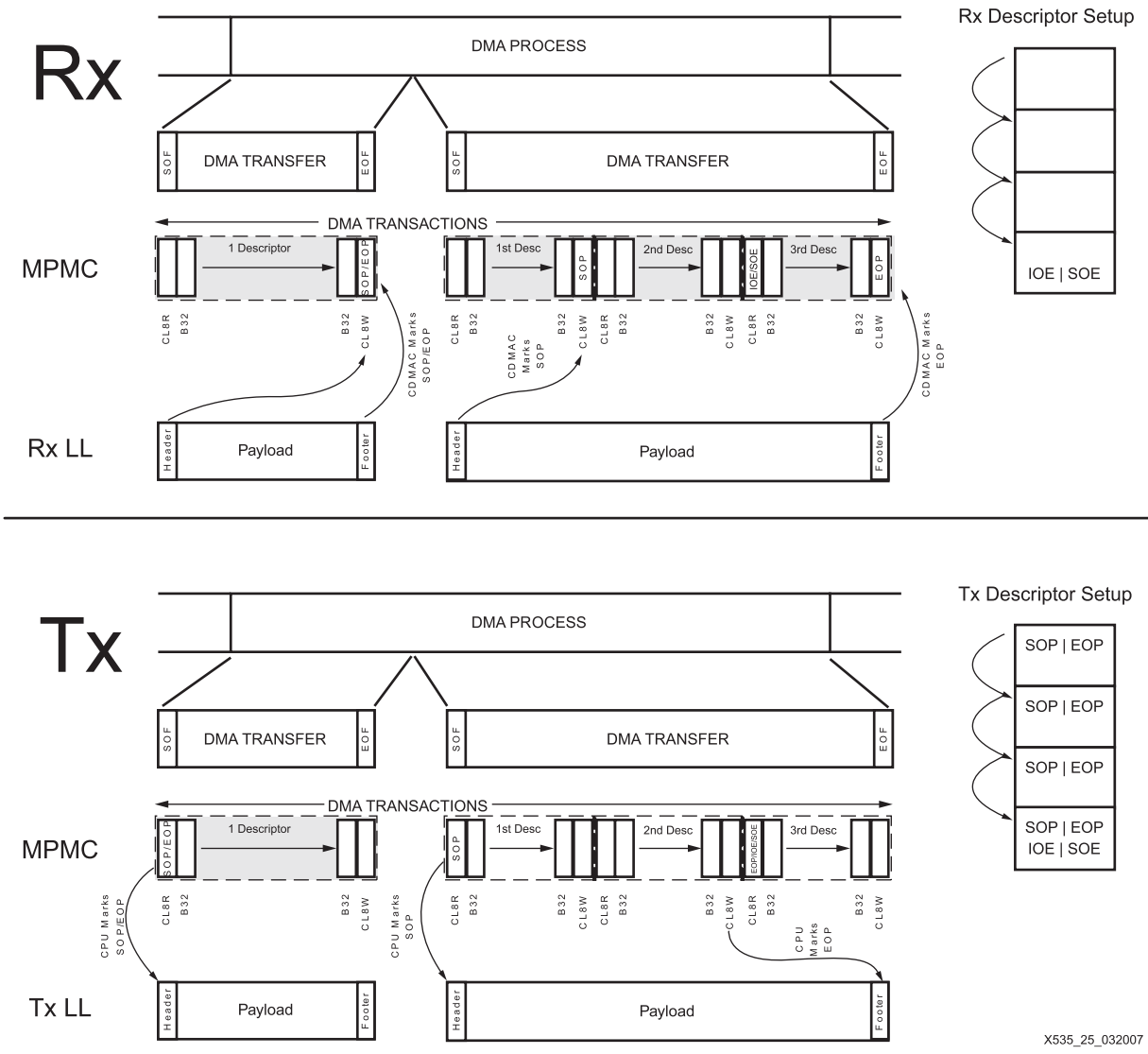


Figure A-3: **The DMA Process**

Figure A-3 illustrates the hierarchy of the DMA process and relates that to the operations in the MPMC2 and LocalLink interface. The figure shows four descriptors and LocalLink frames for Rx and Tx.

The DMA process is demarcated from the instant where DMA operations are started (for example, DCR write to the current description pointer in the engine until a DMA descriptor marked with the STOP_ON_END flag set in the CDMAC STATUS field is reached.

The DMA transfer is demarcated by the descriptor(s), which contain a START_OF_PACKET and END_OF_PACKET, and thus represent one LocalLink header, payload, and footer. For Rx, the START_OF_PACKET and END_OF_PACKET come from the LocalLink start of file and end of file (SOF/EOF) signals and are written back into the descriptor(s) as the DMA transactions complete. This is in contrast with Tx, wherein START_OF_PACKET and END_OF_PACKET are set by the CPU and control when the LocalLink interface issues the SOF and EOF signals.

The DMA transactions are individual MPMC2 operations such as 8-word cache-line write (CL8W), 8-word, cache-line read (CL8R), 128-byte burst write (B32W) and 128-byte burst read (B32R). When put together, these comprise the individual pieces of a DMA transfer. DMA transactions are atomic units: once a DMA transaction begins on the MPMC2, both the MPMC2 and CDMAC are locked together until the MPMC2 completes the memory operation.

Figure A-3, page 100 shows how Rx and Tx differ in handling the LocalLink framing flags. During Tx, the START_OF_PACKET and END_OF_PACKET flags in the descriptors are used to send the SOF and EOF signals across the LocalLink interface. In contrast, during Rx operations these flags are actually set by the LocalLink interface, and are written back into the descriptor once the descriptor has been successfully processed. In the three descriptor Rx case, the START_OF_PACKET flag is set in the first descriptor during its CL8W writeback while the END_OF_PACKET flag is set in the last (for example, third) descriptor during its CL8W writeback. Rx descriptors must always have their START_OF_PACKET and END_OF_PACKET flags cleared prior to the onset of DMA operations, or the CDMAC responds improperly. This is one of the elements that must be addressed during CDMAC scrubbing operations.

The conclusion of the DMA process is also shown in this example. To end a DMA process, the CDMAC engine must encounter a descriptor with the STOP_ON_END flag set. The last descriptors of both the Rx and Tx examples show this bit being set. The CDMAC processes DMA descriptors continually until it reaches a descriptor with the STOP_ON_END flag set. This descriptor executes to completion, and then the CDMAC engine stops in an orderly fashion. In the example shown in Figure A-3, the INT_ON_END flag is also set in the descriptor. After the CDMAC engine has executed this descriptor to completion, it sets the appropriate bit in the CDMAC interrupt STATUS register, and generates a CDMAC_INT, if enabled.

Figure A-4, page 102 shows a simple example of how a DMA process progresses for a Tx DMA engine. DMA Process #1 is the entirety of all operations performed. In this case, three separate descriptors are used for the DMA process.

- The first two descriptors demarcate the first packet of data to be transmitted across the LocalLink interface and make up the first DMA transfer.
- The third descriptor demarcates an entire packet within a single descriptor and makes up the second DMA transfer.

Figure A-4 illustrates that a DMA transfer is the movement of a packet of data across the LocalLink interface, regardless of how many descriptors it takes to declare the packet. Finally note that each box represents a separate DMA transactions. DMA transactions are memory operations to the MPMC2. In this example, three types of DMA transactions are performed: 8-word reads, 32-word reads, and 8-word writes.

Each time the MPMC2 must perform a memory operation, a DMA transaction is considered to have been performed. The number of DMA transactions is always at least three per DMA transfer. This is because there is always a reading of the descriptor, at least one transfer of data, and a writing of the descriptor. There can be many more DMA transactions as dictated by the buffer size field of the descriptor module 128 bytes.

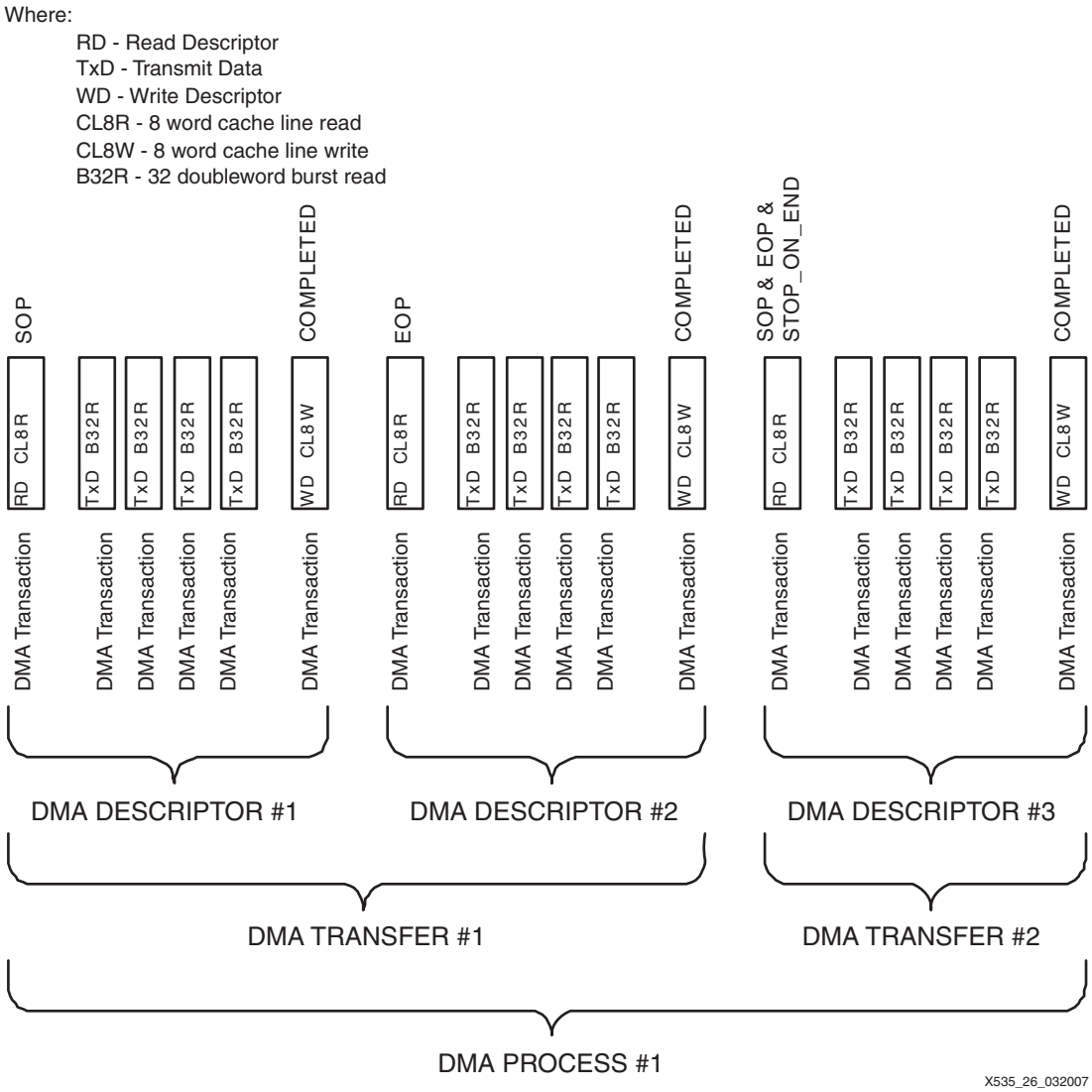


Figure A-4: **CDMAC Illustration of Tx Engine Flow**

DMA Descriptor Model

The CDMAC is controlled by DMA descriptors. The DMA descriptors are initialized by the CPU prior to starting the DMA engine. The current implementation of the CDMAC contains four independent engines that can be simultaneously processing four different DMA descriptors or chains of DMA descriptors. Figure A-5 illustrates the DMA descriptor model. The *"CDMAC Software Model," page 155* describes the software use model and register model for CDMAC.

The DMA descriptor must be 8-word aligned in its base address. This is required so that the CDMAC does not have to be inordinately complex and large. Generally, this does not place a large burden on software developers, so long as they are aware of the limitation up front.

The descriptor uses 8 words. The first three are used exclusively by the CDMAC while the fourth word contains some CDMAC information. The final words are designed to be used by the application that is using the particular CDMAC engine.

- The first word contains a pointer to the next descriptor. This pointer allows the CDMAC to continue to run until the pointer is either NULL or the engine has otherwise been instructed to stop.
- The second word in the descriptor contains a byte-aligned address that points to the location of the data buffer to be moved.
- The third word in the descriptor contains the number of bytes to move.

In the fourth word, the upper byte is used to house control and status information for the CDMAC. See Figure A-7, page 107.

The last three bytes of the fourth word, and the last three words are made available to the application, and are broadcast over the LocalLink interface at appropriate times.

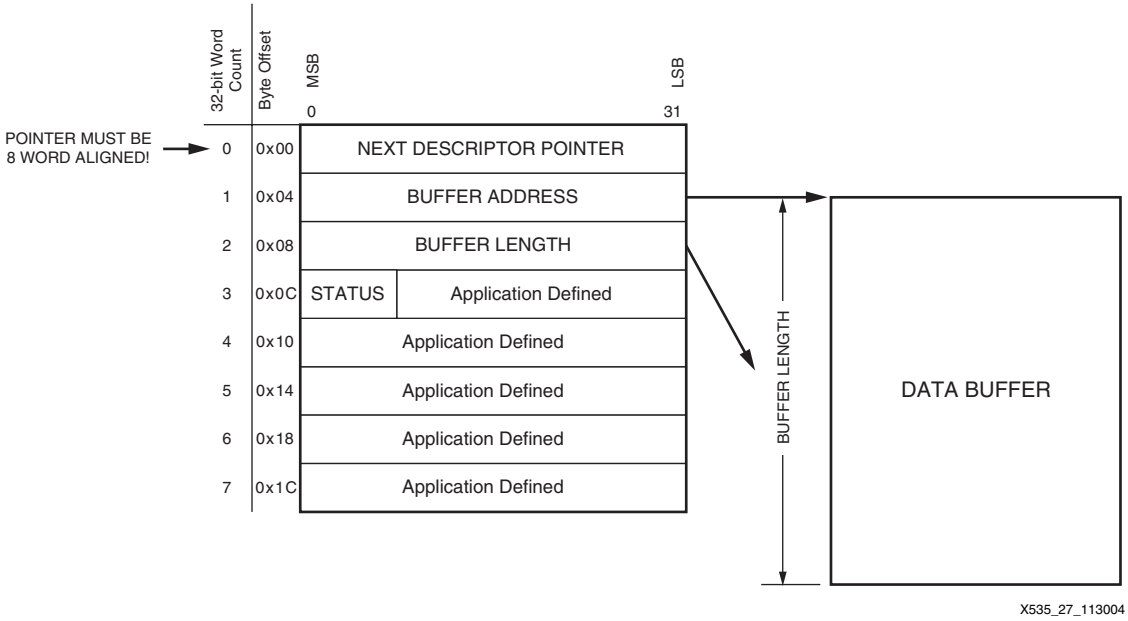


Figure A-5: **CDMAC DMA Descriptor Model**

An important detail about the APPLICATION_DEFINED fields is that the fields are only broadcast down the LocalLink interface during the first Tx descriptor that sets the SOP bit on the LocalLink interface. Subsequent descriptors in the same LocalLink payload do not have the fields sent because they do not cause a LocalLink header operation.

Similarly, for Rx, the APPLICATION_DEFINED fields are only written back to the last DMA descriptor that was in process with the LocalLink interface encountered and EOP. If the Rx was made up of several descriptors for that LocalLInk payload, only the last descriptor gets the fields updated.

The descriptor STATUS field is shown in Figure A-6. This field contains two main parts:

- The CDMAC_STATUS field
- An APPLICATION_DEFINED field

To determine the appropriate response, the STATUS field provides the CDMAC with inputs during the read of the descriptor. Similarly, when the descriptor is written back to memory upon completion, certain bits are updated. Not all bits are read during DMA transaction descriptor read nor are all bits updated during DMA transaction descriptor writes.

The CDMAC_START_OF_PACKET and CDMAC_END_OF_PACKET bits help frame the LocalLink interface; the use of these bits differ from Rx to Tx as follows:

- When the descriptor is in use for Rx, the LocalLink interface sets the CDMAC_START_OF_PACKET and CDMAC_END_OF_PACKET bits.
- When the descriptor is in use for Tx, the CPU sets the CDMAC_START_OF_PACKET and CDMAC_END_OF_PACKET bits to control the LocalLink interface.

The START_OF_PACKET instructs the LocalLink interface to initiate a header for this transaction.

Similarly, the END_OF_PACKET bit instructs the LocalLink interface to initiates a footer for this transaction. The bits can be mixed and matched. For example, in Tx, three descriptors might be defined to communicate a full payload of data across the LocalLink interface. The first descriptor would be marked START_OF_PACKET, the second neither, and the third marked as END_OF_PACKET. This allows the chaining of non-contiguous data buffers into an apparently contiguous data payload across the LocalLink interface.

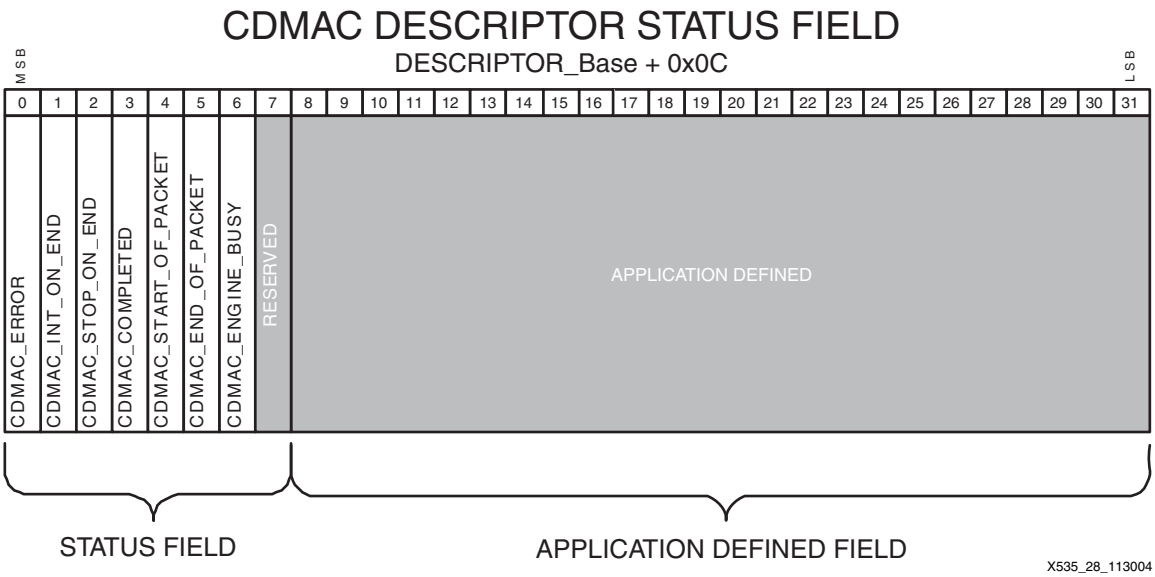


Figure A-6: **CDMAC Descriptor, STATUS field**

When set in the descriptor:

- The CDMAC_INT_ON_END bit causes the CDMAC to generate a CPU interrupt, and sets the appropriate interrupt flag in the INTERRUPT register. The interrupt is sent to the CPU only if the MIE bit is set in the INTERRUPT register, and the CDMAC has completed all the data move specified by the descriptor.
- The CDMAC_STOP_ON_END bit causes the CDMAC to stop DMA operations upon the successful completion of the current descriptor. This stop allows the CDMAC to be brought to an orderly halt and restarted by the CPU when appropriate.

The CDMAC_INT_ON_END and CDMAC_STOP_ON_END bits can be mixed and matched together to effect the best operation that software can contextually require.

The CDMAC_COMPLETED bit is written back to the descriptor upon the successful completion of the DMA transfer specified by that descriptor.

Tx Descriptor Operations

To start Tx operations, the CPU writes a pointer to the first descriptor in the chain to the CURRENT_DESCRIPTOR_POINTER register. The CDMAC begins by reading the descriptor that is pointed at by its CURRENT_DESCRIPTOR_POINTER register. During the read of the descriptor, the CDMAC records the data in the first 4 words, and passes all 8 words from the descriptor to the LocalLink interface. Only when the START_OF_PACKET is marked in the descriptor can the CDMAC create a LocalLink header on the LocalLink interface.

Figure A-7, page 107 shows an example of how Tx descriptors might be chained together and Figure A-29, page 140 shows an example of The LocalLink interface.

In these examples, the DMA descriptors are set such that a single descriptor corresponds to a single LocalLink payload transfer, including header, payload and footer.

The STATUS and APPLICATION_DEFINED fields are broadcast during the header portion of the LocalLink transaction. These fields are placed on the LocalLink interface only when the descriptor has the START_OF_PACKET set in the STATUS field.

Once the LocalLink header phase has completed, The CDMAC takes the data to which the buffer address points and moves it from memory to the LocalLink interface as data during the payload phase.

The CDMAC continues to transfer data and count down the `BUFFER_LENGTH` field to zero, and then attempts to get the next descriptor. If the next descriptor pointer field is `NULL` (for example, 0x00000000), then the DMA engine stops. If it is non-zero, and a `STOP_ON_END` has not been issued in the current descriptor, then the CDMAC transfers the contents of the `NEXT_DESCRIPTOR_POINTER` register into the `CURRENT_DESCRIPTOR_POINTER` register.

The act of transfer reinitializes the CDMAC to go fetch the descriptor to which the `CURRENT_DESCRIPTOR_POINTER` indicates. It can be thought of as the CPU writing the `CURRENT_DESCRIPTOR_POINTER` again to initiate the CDMAC. The DMA process continues until the CDMAC encounters a `NULL` pointer in the `NEXT_DESCRIPTOR_POINTER` or a `STOP_ON_END` in the `STATUS` field.

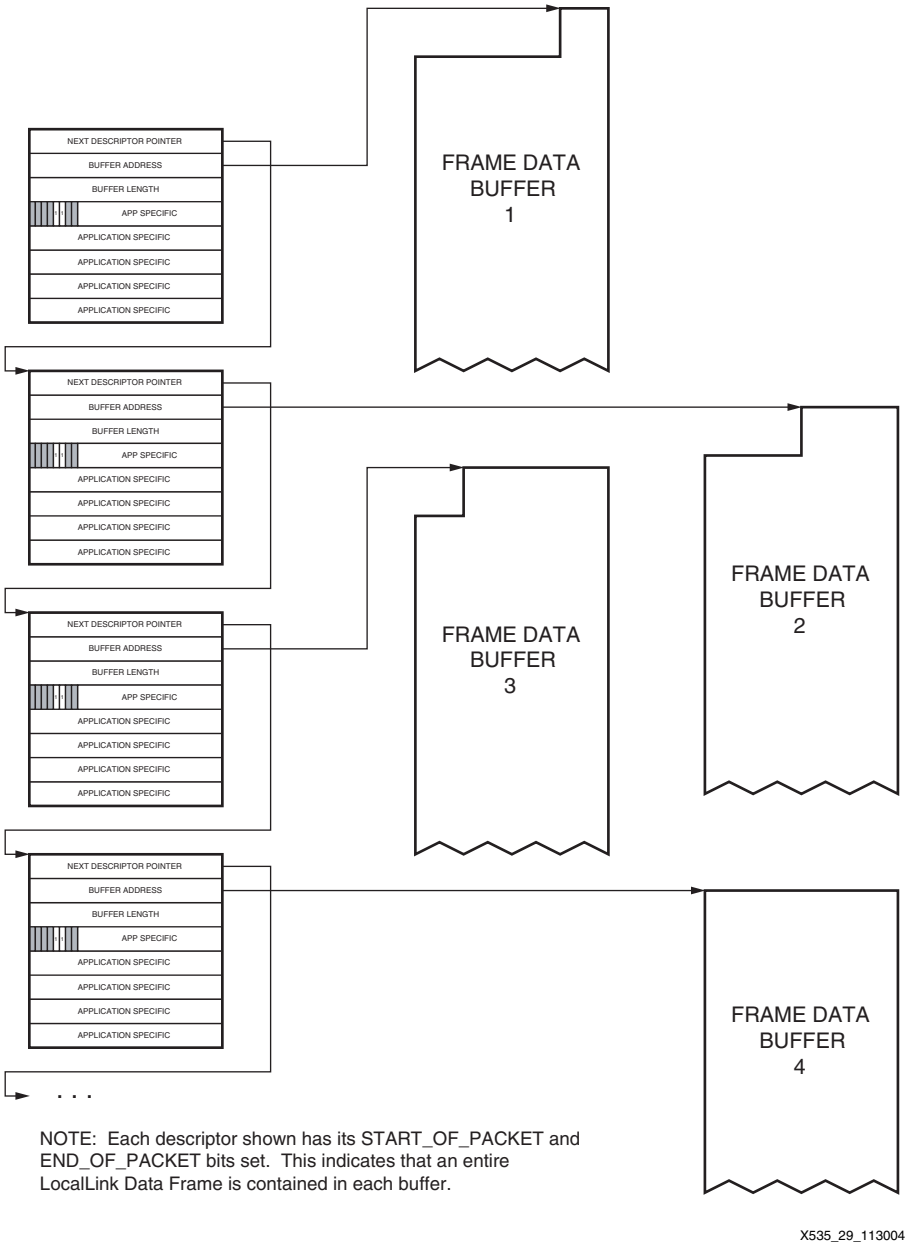


Figure A-7: **Example Chain of CDMAC Tx Descriptors**

Rx Descriptor Operations

The Rx operation is similar to the Tx, in that it begins by reading the descriptor pointed at by the CURRENT_DESCRIPTOR_POINTER. During the read of the descriptor, the CDMAC records the data in the first 4 words but does *not* send it down the LocalLink interface, 1c0000000e, during the header. This is because LocalLink is a unidirectional interface, and the data is pointing in the wrong direction. The CDMAC only receives data from the Rx LocalLink device. While the header time is maintained across the LocalLink interface, there is no valid data.

The CDMAC exits the header with the Rx LocalLink device issues a Start Of Payload (SOP) signal. The CDMAC then receives data from the LocalLink interface during the payload phase and stores the data to memory at the address to which the buffer address points. This process continues until one of two things happens:

- An End Of Payload (EOP) is received indicating the end of the payload
- The buffer length decrements to zero

If the BUFFER LENGTH decrements to zero, an error has occurred and the CDMAC halts operations.

Once an EOP signal is received, the CDMAC begins to receive the footer. The footer contains the APPLICATION_DEFINED fields, which are then written back to the memory, along with the current STATUS.

Note: Only the descriptor that has the END_OF_PACKET bit marked has valid data in the APPLICATION_DEFINED section of the descriptor.

Note: The CDMAC does not overwrite the STATUS and APPLICATION_DEFINED sections of other descriptors not marked END_OF_PACKET. This could be useful for internal device driver storage if it can be guaranteed that the descriptor will not get an EOP.

Hardware

CDMAC Architecture

The CDMAC is designed in a modular fashion. It is designed to bolt between the MPMC2 and LocalLink devices. Figure A-8 illustrates the basic functional diagram of the CDMAC. The CDMAC is composed of four effective elements. The MPMC2 port interface is used to connect to the MPMC2. Similarly, LocalLink interfaces are used to connect the producers and consumers of data to the CDMAC. The remaining block contains the main CDMAC engines and control logic. Because the CDMAC is a complex device, it is illustrated in a variety of differing manners to assist in understanding its construction and modification.

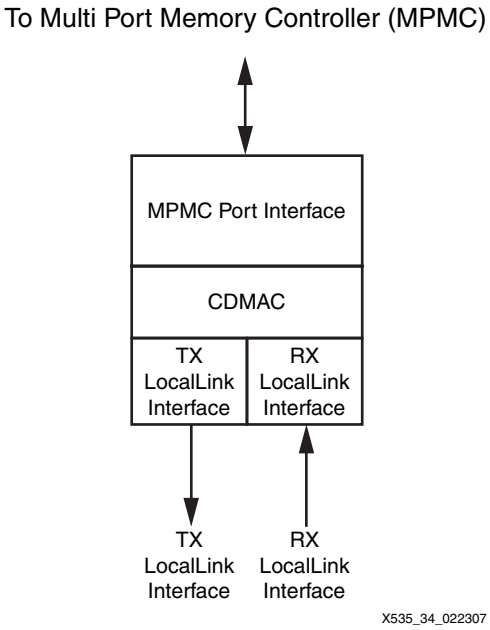


Figure A-8: **CDMAC Functional Diagram**

Figure A-9 shows the top-level module block diagram that illustrates how the source code is constructed. This diagram assists in understanding the source code and how it can be modified to fit your individual system design needs.

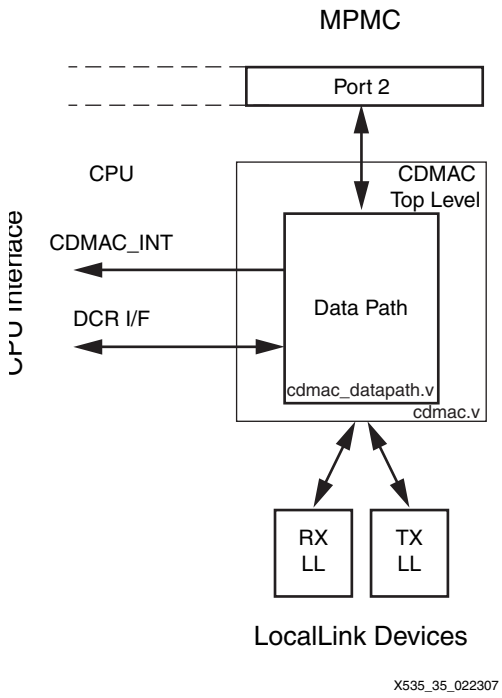


Figure A-9: **CDMAC Top Level Module Block Diagram**

The CDMAC consists of two independent DMA engines that share a common set of registers. The CDMAC is divided into separate Rx and Tx DMA engines. The Rx and Tx DMA engines share a structure that permits each engine to have fair and arbitrated access to its respective port.

Each DMA engine is connected to a unidirectional LocalLink interface. The LocalLink interface permits a streaming data device to be connected to the CDMAC. Where a device requires full-duplex operation, it uses both the Rx and Tx LocalLink interfaces. Each LocalLink interface is configured to allow for the transmission and reception of data from the CDMAC descriptors for that DMA engine, though the Rx and Tx differ in how they do this.

Top Level Functionality

Figure A-10, page 111 illustrates the basic operational aspects of the CDMAC. One of the main design principles of the CDMAC is to use the smallest FPGA area possible. The CDMAC does this by not replicating the traditional counters that exist in most other DMA controllers. Instead, the CDMAC shares a central register file with a smaller set of counters. This principle can be used to extend the CDMAC to add more engines.

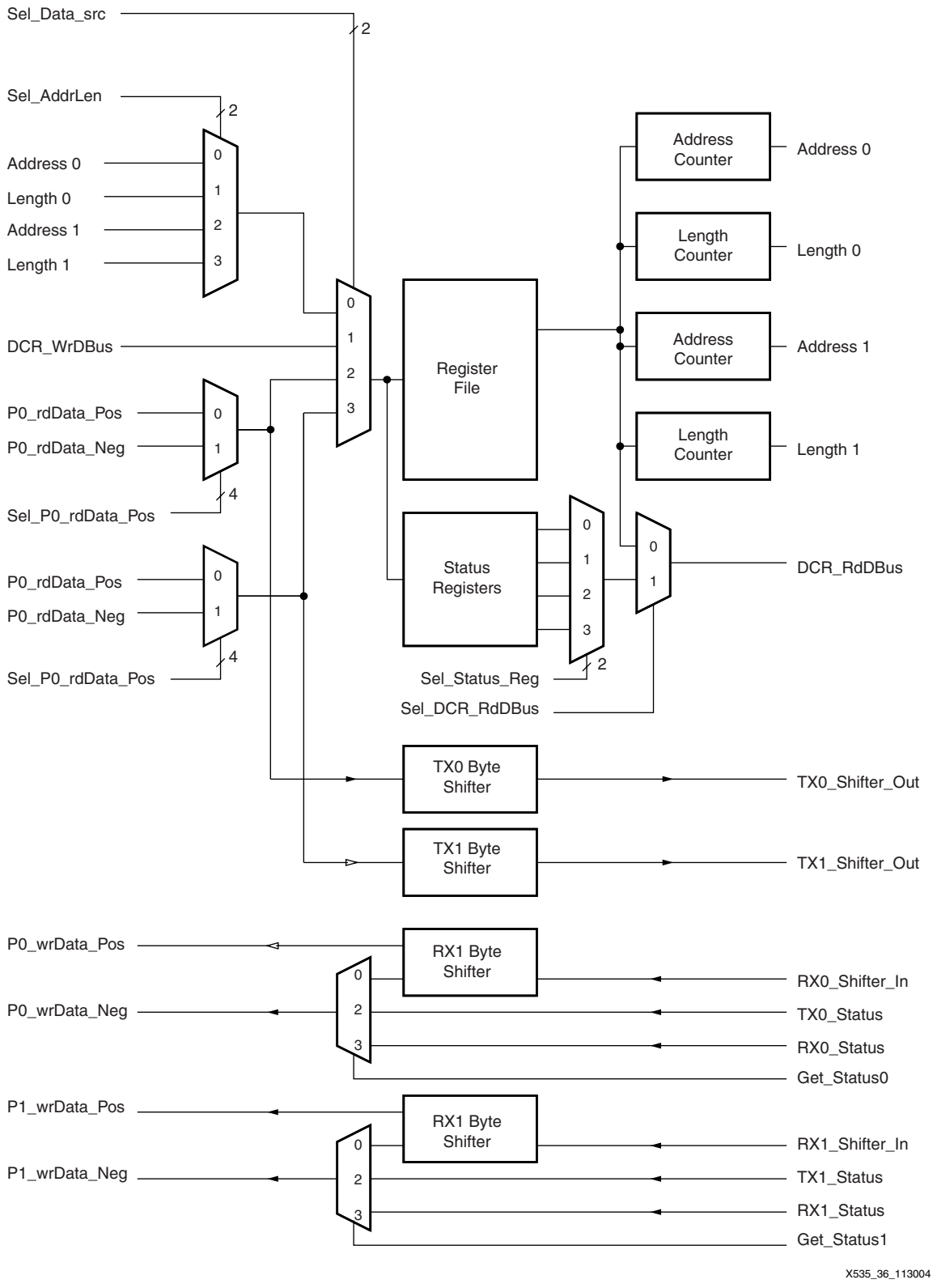


Figure A-10: **CDMAC Basic Architecture (not including control)**

State Machine Design

Figure A-11 illustrates how the various state machines interrelate. The CDMAC contains a Rx and a Tx DMA engine, which are detailed in Figure A-12, page 113, Figure A-13, page 114 and Figure A-14, page 115.

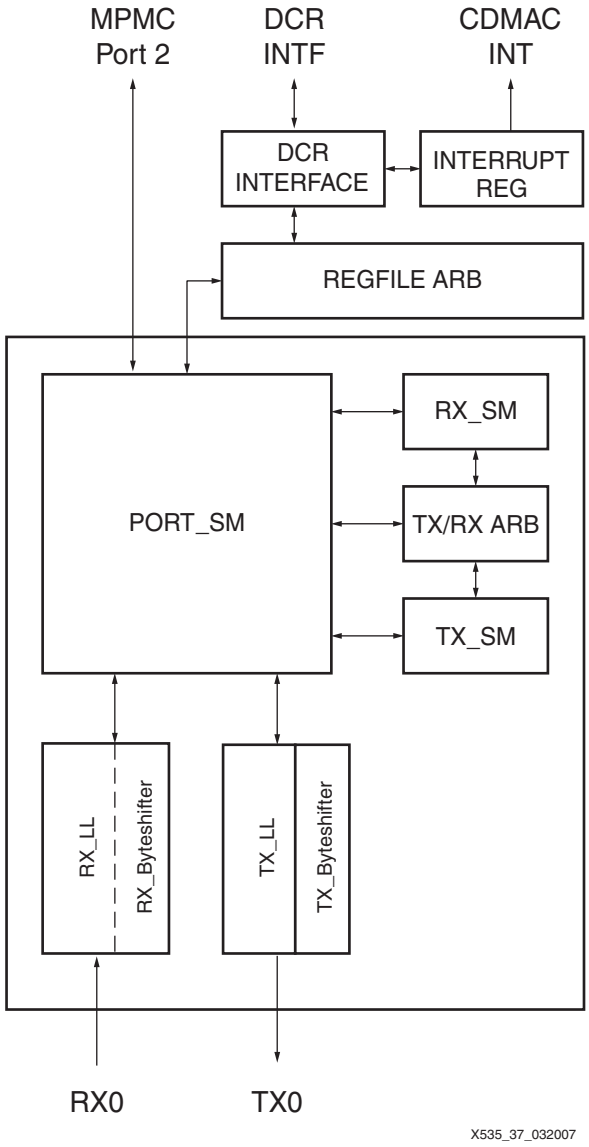


Figure A-11: **CDMAC State Machine Conceptual Block Diagram**

Figure A-12, page 113 is a lower level diagram of Figure A-11 and shows the major connections between the various state machines.

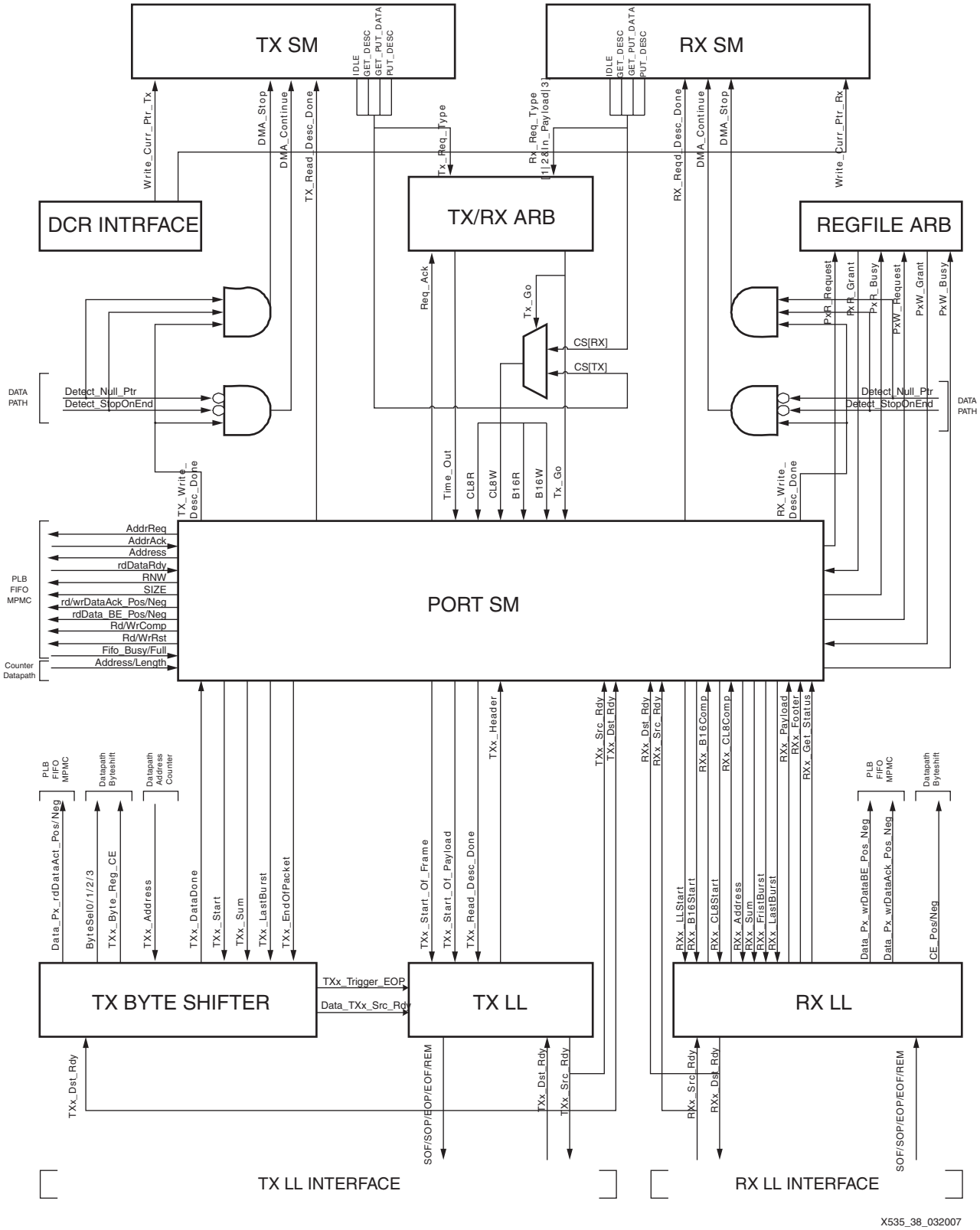


Figure A-12: **CDMAC Relationship of State Machines to each Other (per port)**

Overall Tx State Machine

The Tx state machine, shown in Figure A-13, determines if a Tx port is idle (IDLE), reading a descriptor from memory (GET_DESC), reading data from memory and sending it to the LocalLink interface (GET_PUT_DATA), or writing the status back to memory (PUT_DESC).

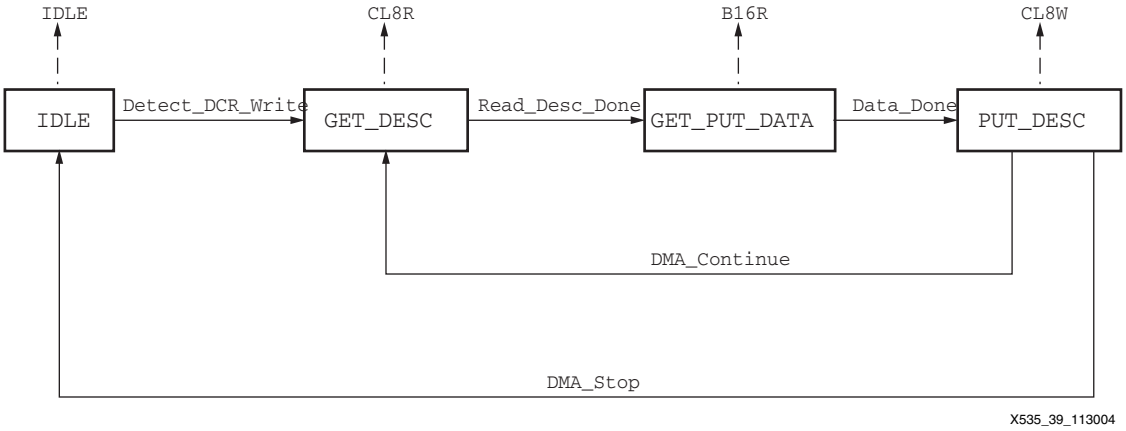


Figure A-13: **CDMAC Tx_SM State Diagram**

The state machine begins in the IDLE state. When the CPU issues a DCR_write to the TX current descriptor pointer, Detect_DCR_Write is asserted and the state machine transitions to the GET_DESC state.

While in the GET_DESC state, an 8-word cache-line read (CL8R) request is issued to the TX/RX Arbiter. Once the CL8R has completed, the Read_Desc_Done signal is asserted and the state machine transitions to the GET_PUT_DATA state.

The GET_PUT_DATA state issues continuous 32-word burst read (B32R) requests to the TX/RX Arbiter until all of the data specified by the descriptor has been collected from memory and sent across the LocalLink interface. This is indicated by the assertion of the Data_Done signal. When this signal is asserted, the state machine transitions into the PUT_DESC state.

After transitioning to the PUT_DESC state, the Tx state machine issues an 8-word cache-line write (CL8W) request to the TX/RX Arbiter.

After the CL8W has completed, either the DMA_Continue or the DMA_Stop signal is asserted.

If the STATUS register indicates that the next descriptor pointer is not a null pointer and the Stop_On_End bit is not set, the DMA_Continue signal is asserted and the state machine transitions to the GET_DESC state. Otherwise, the DMA_Stop signal is asserted and the state machine transitions to the IDLE state. The CL8R, B32R, and CL8W signals are converted to a bus called Tx_Req_Type, as shown in Figure A-12, page 113.

Overall Rx State Machine

The Rx state machine, shown in Figure A-14, determines if a Rx port is idle (IDLE), reading a descriptor from memory (GET_DESC), collecting data from the LocalLink interface and writing the data to memory (GET_PUT_DATA), or writing the status and application defined data back to memory (PUT_DESC).

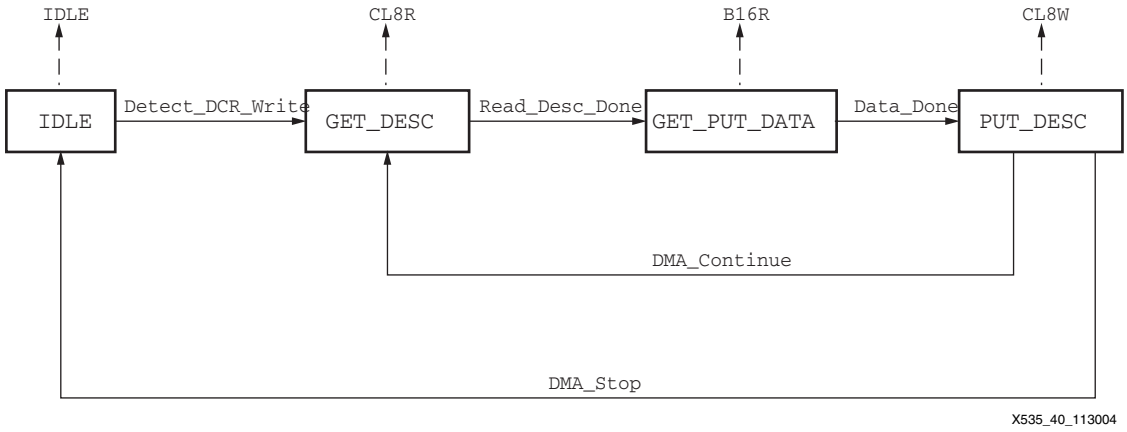


Figure A-14: **CDMAC Rx_SM State Diagram**

The state machine begins in the IDLE state. When the CPU issues a DCR write to the RX current descriptor pointer, Detect_DCR_Write is asserted and the state machine transitions to the GET_DESC state.

In the GET_DESC state, an 8-word cache-line read (CL8R) request is issued to the TX and RX Arbiter. When the CL8R has completed, the Read_Desc_Done signal is asserted and the state machine transitions to the GET_PUT_DATA state.

The GET_PUT_DATA state issues continuous 32-word burst write (B32W) requests to the TX and RX Arbiter until all of the data specified by the descriptor is collected from the LocalLink interface and written to memory. This is indicated by the assertion of the Data_Done signal. When this signal is asserted, the state machine transitions into the PUT_DESC state.

After transitioning to the PUT_DESC state, the Rx state machine issues an 8-word cache-line write (CL8W) request to the TX and RX Arbiter.

After the CL8W has completed, either the DMA_Continue or the DMA_Stop signal is asserted.

If the STATUS register indicates that the next descriptor pointer is not a null pointer and the Stop_On_End bit is not set, the DMA_Continue signal is asserted and the state machine transitions to the GET_DESC state. Otherwise, the DMA_Stop signal is asserted and the state machine transitions to the IDLE state.

The CL8R, B32W, and CL8W signals are converted to a bus called Rx_Req_Type, as shown in Figure A-12, page 113.

Arbitration State Machine for Overall Rx and Tx State Machines

Figure A-15 shows the logic for the Arbitration state machine for the Overall Rx and Tx state machines (Tx and Rx arbitration). The Overall Rx and Tx state machines assert request signals to the arbiter through the `Tx_Req` and `Rx_Req` signals. These signals are the same signals as `Tx_Req_Type` and `Rx_Req_Type` described in "Overall Tx State Machine," page 114 and "Overall Rx State Machine," page 115.

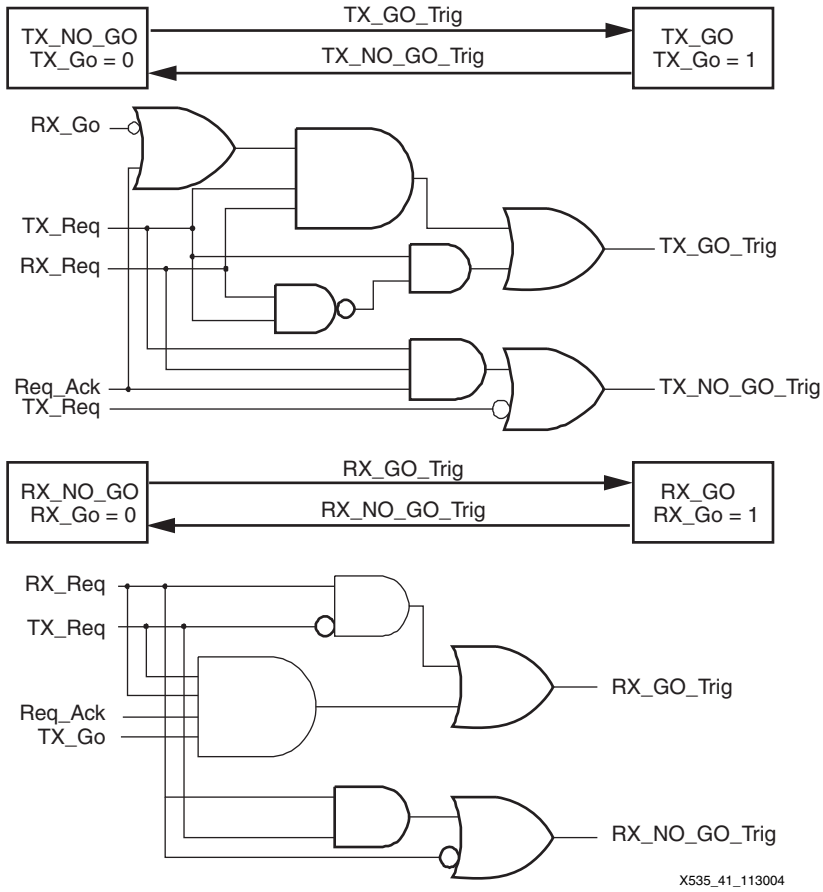


Figure A-15: **CDMAC Tx_Rx_Arb_SM State Diagram**

The arbitration algorithm is like two state machines: one for the Tx engine and one for the Rx engine.

The Tx arbitration state machine starts in the `TX_NO_GO` state. If a Tx engine request is issued from the overall Tx state machine and the Rx arbitrations state machine is in the `RX_NO_GO` state, the Tx arbitration state machine transitions to the `TX_GO` state. The state machine stays in the `TX_GO` state until the request is acknowledged, then the state machine returns to the `TX_NO_GO` state. If the Tx arbitration state machine is in the `TX_NO_GO` state and a Tx engine request is issued while the Rx arbitration state machine is in the `RX_GO` state the Tx arbitration state machine waits until the `Rx_Request` is acknowledged, then it transitions into the `TX_GO` state.

The Rx arbitration state machine behaves identically to the Tx arbitration state machine, except that if both state machines are in the `TX_NO_GO` state and the `RX_NO_GO` state, and the `Tx_Req` and the `Rx_Req` signals are asserted at the same time, the Tx arbitration state machine has priority.

Port State Machine

The port state machine is the main control for the CDMAC and is shown in Figure A-16. The port state machine contains two state machines that closely interact with each other because of the resource sharing in the register file. The read state machine executes descriptor read transactions and Tx burst read transactions. The write state machine executes descriptor write transactions and Rx burst write transactions.

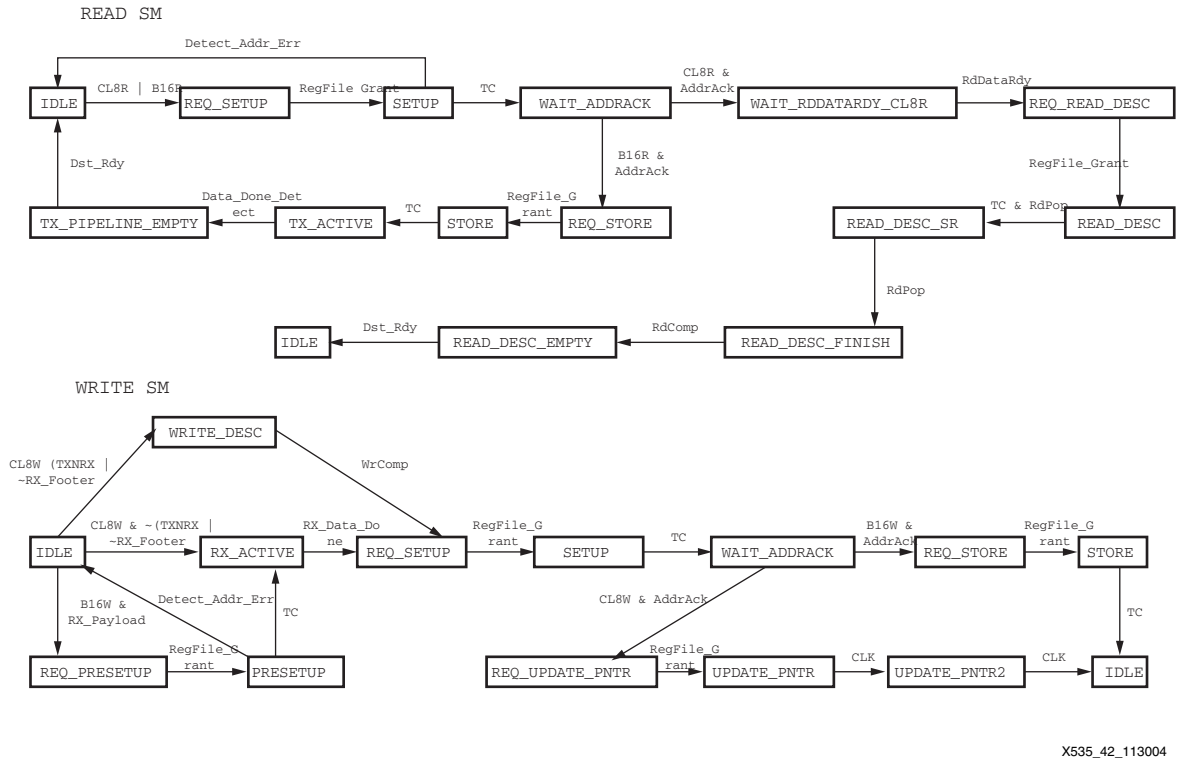


Figure A-16: **CDMAC Port_SM State Diagram**

Read State Machine

The read state machine begins in the IDLE state. When the TX and RX Arbiter issues an 8-word cache-line read (CL8R) request or a 32-word, burst read (B32R) request, the read state machine enters the REQ_SETUP state.

While the read state machine is in the REQ_SETUP state, the state machine requests access to the register file to read the buffer address and buffer length registers. Once access has been granted, the state machine transitions into the SETUP state.

The buffer address and buffer length counters are loaded from the register file while the read state machine is in the SETUP state.

If the buffer address is invalid, an error is generated and the read state machine returns to the IDLE state.

If there is no error when the counters are loaded, the read state machine transitions to the WAIT_ADDRACK state. The WAIT_ADDRACK state issues either a CL8R request or a B32R read request. When the request is acknowledged, the read state machine transitions to one of two states. The first state, WAIT_RDDATARDY_CL8R, is for CL8Rs and asserts control signals to read a descriptor. The second state, REQ_STORE, is for B32Rs and asserts control signals to read data from memory and transmit that data over the LocalLink interface.

If the read state machine is reading a descriptor, the state machine transitions out of the WAIT_ADDRACK state and into the WAIT_RDDATARDY_CL8R state after the port interface acknowledges the CL8R request.

In the WAIT_RDDATARDY_CL8R state, the read state machine waits until the port interface asserts the RdDataRdy signal to indicate that data is available on the port interface and that the CDMAC can pop data out of the MPMC2 read FIFOs on every clock cycle following the assertion of RdDataRdy.

The state machine then transitions into the REQ_READ_DESC state. In the REQ_READ_DESC state, the state machine requests access to the register file. When access is granted, the state machine transitions into the READ_DESC state.

The READ_DESC state pops the next descriptor pointer, the buffer address, and the buffer length out of the MPMC2 read FIFOs. This data is placed into the register file and the read state machine transitions to the READ_DESC_SR. The data is also sent across the LocalLink interface as header data. The READ_DESC_SR state pops the STATUS register value out of the MPMC2 read FIFOs. When this data is stored in the STATUS register, the read state machine transitions to the READ_DESC_FINISH state.

In the READ_DESC_FINISH state, the last four words of data pop out of the MPMC2 read FIFOs. This data is ignored. When the port interface issues the RdComp signal, the read state machine transitions to the READ_DESC_EMPTY state.

The READ_DESC_EMPTY state waits for the LocalLink interface to be ready to receive data from the CDMAC. When the Dst_Rdy signal is received from the LocalLink interface, the read state machine transitions into the IDLE state.

If the read state machine is in the WAIT_ADDRACK state and is reading data to be transmitted over the LocalLink interface, the state machine transitions out of the WAIT_ADDRACK state and into the REQ_STORE state after the port interface acknowledges the B32R request.

In the REQ_STORE state, the state machine requests access to the register file. When access is granted, the state machine transitions into the STORE state.

In the STORE state:

- The buffer address and buffer length registers are updated with the buffer address and the buffer length to be used in the next transaction.
- The buffer address is incremented by the number of bytes that read from memory on this transaction.
- The buffer length is decremented by the number of bytes that are read from memory on this transaction.

When these registers have been updated, the read state machine transitions to the TX_ACTIVE state.

In the TX_ACTIVE state, data is read from memory and sent to the Tx byteshifter.

After the data pops out of the MPMC2 read FIFOs or the MPMC2 read FIFOs are reset, the read state machine transitions to the TX_PIPELINE_EMPTY state.

The read state machine transitions from the TX_PIPELINE_EMPTY state to the IDLE state after the last word of data is acknowledged on the LocalLink interface.

Write State Machine

The write state machine is very similar to the read state machine. The write state machine begins in the IDLE state. Depending on the type of request being issued from the TX/RX arbiter, the write state machine transitions into one of the following three states:

- If the TX/RX arbiter is issuing a CL8W request and either the request is for the TX engine or the RX LocalLink interface is not in the footer state, the state machine transitions into the WRITE_DESC state.
- If the TX/RX arbiter is issuing a CL8W request, and the request is for the RX engine, and the RX engine is in the footer state, the state machine transitions into the RX_ACTIVE state.
- If the TX/RX arbiter is issuing a B32W request and the RX engine is in the payload state, the state machine transitions to the REQ_PRESETUP state.

If the write state machine transitions from the IDLE state to the WRITE_DESC state, the state machine waits for eight words of descriptor data to be pushed into the MPMC2 write FIFOs, then the state machine transitions into the REQ_SETUP state.

If the write state machine transitions from the IDLE state to the REQ_PRESETUP state, the state machine requests access to the register file. Once access has been granted, the state machine transitions into the PRESETUP state.

In the PRESETUP state, the buffer address and the buffer length counters are loaded with the contents of the register file. When these counters are loaded, the write state machine transitions to the RX_ACTIVE state.

If the write state machine transitions from the IDLE state or the PRESETUP state to the RX_ACTIVE state, the state machine waits for payload or footer data from the Rx LocalLink interface to be pushed into the MPMC2 write FIFOs, then the state machine transitions into the REQ_SETUP state. If footer data is being pushed into the MPMC2 write FIFOs, the data or the byte enables are modified as specified in "Rx LocalLink and Byteshifter," page 127.

The REQ_SETUP state requests access to the register files. When access is granted, the write state machine transitions into the SETUP state.

While in the SETUP state, the STATUS register is updated, then the write state machine transitions into the WAIT_ADDRACK state.

A write request is issued on the port interface when the write state machine is in the WAIT_ADDRACK state. When the request is acknowledged, the state machine transitions to one of two states. If the request is a:

- B32W request, the state machine transitions to the REQ_STORE state.
- CL8 request, the state machine transitions to the REQ_UPDATE_PNTR state.

If the write state machine transitioned from the WAIT_ADDRACK state into the REQ_STORE state, the state machine requests access to the register file. When access is granted, the state machine transitions into the STORE state.

In the STORE state, the buffer address and buffer length registers are updated with the buffer address and buffer length to be used in the next transaction. After these registers are updated, the write state machine transitions into the IDLE state.

If the write state machine transitioned from the WAIT_ADDRACK state into the REQ_UPDATE_PNTR state, the state machine requests access to the register file.

When access is granted, the state machine transitions into the `UPDATE_PNTR` state.

- In the `UPDATE_PNTR` state, the next descriptor pointer register is read from the register file, then the write state machine transitions into the `UPDATE_PNTR2` state.
- In the `UPDATE_PNTR2` state, the next descriptor pointer is written to the `CURRENT_DESCRIPTOR_POINTER` register, then the write state machine transitions into the `IDLE` state.

Tx LocalLink and Byteshifter

The Tx LocalLink and byteshifter logic take data from the appropriate place in memory and move the data across the LocalLink interface. This concept is shown in Figure A-17.

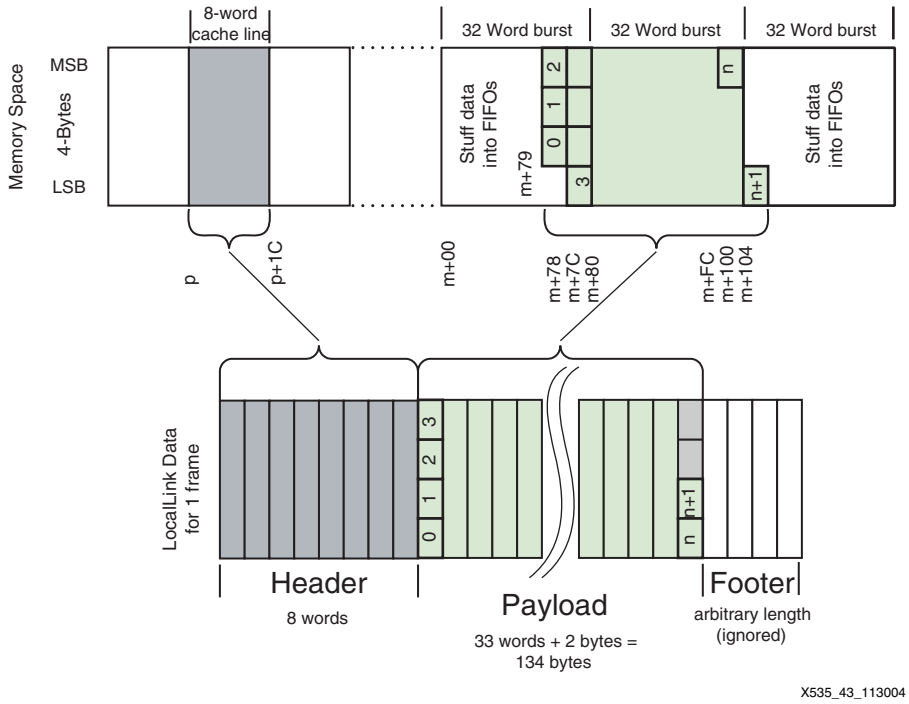


Figure A-17: **CDMAC Tx Byteshift Example**

In this example:

- The CDMAC reads the descriptor at address `p` to `p+1C` and sends it to the LocalLink as the header. The payload is 136 bytes and starts at address `m+79`.
- The Tx byteshifter sends data acknowledges to the memory controller while keeping the `Src_Rdy` signal to the LocalLink deasserted, because address `m+79` is not 32-word aligned.
- Data from address `m` to `m+78` are discarded. Data is offset by 78 bytes, so the first byte of data occurs on the second byte location on the posedge of the DDR SDRAM.
- The Tx byteshifter takes the `posedge` (x 0 1 2) and `negedge` (3 4 5 6), which are both present at the time, recombines them to form a new, correctly shifted, word (0 1 2 3), and sends it over the LocalLink as the payload.

At the end of the first 32-word burst read (B32R), three bytes are left over and kept in the byteshifter.

After the second burst occurs, those three bytes are combined with the first byte of the second burst and sent over LocalLink, and again between the second burst and third burst.

On the last word of the payload the Rem signal is set to indicate which bytes of the word are valid.

Rem is 0x0 in Figure A-17 to indicate all four bytes are valid. After byte n+1 is sent, the FIFOs in MPMC2, which hold all 32 words of the burst, are reset to avoid extra data acknowledge.

For Tx transfer, the footer is not used.

The STATUS bits are written back to the descriptor's STATUS field.

Tx Byteshifter Logic

The Tx byteshifter Block Diagram is shown in Figure A-18, page 122. The Tx byteshifter logic has two stages.

In stage one:

- two 32-bit data, one from the posedge of the DDR SDRAM (rdData_Pos) and one from the negedge (rdData_Neg), are fed into fTWIST from the FIFOs in the MPMC2.
- fTWIST forms a new word Port_RdData by multiplexing each byte from either the rdData_Pos or rdData_Neg, depending on the offset represented by StartOffset.
- fTWIST is capable of producing a new word every clock cycle because both rdData_Pos and rdData_Neg last for two clock cycles. For example, if StartOffset is **1** when bytes 4-7 become present on rdData_Neg, bytes 0-3 would have already been present on rdData_Pos for one cycle. At this point Sel_Px_rdData_Pos becomes 0b0111. The pattern 0b0111 indicates that the first byte is taken from the first byte from rdData_Neg and the rest from rdData_Pos to form (**4 1 2 3**).
- One clock later, rdData_Pos refreshes to the next value and Sel_Px_rdData_Pos becomes 0b1000. The pattern 0b1000 indicates that the first byte is taken from the first byte from rdData_Pos and the rest from rdData_Neg to form (**8 5 6 7**).

In stage two, the Byte_Selx signals are produced to reorder the bytes in Port_RdData to form the final word.

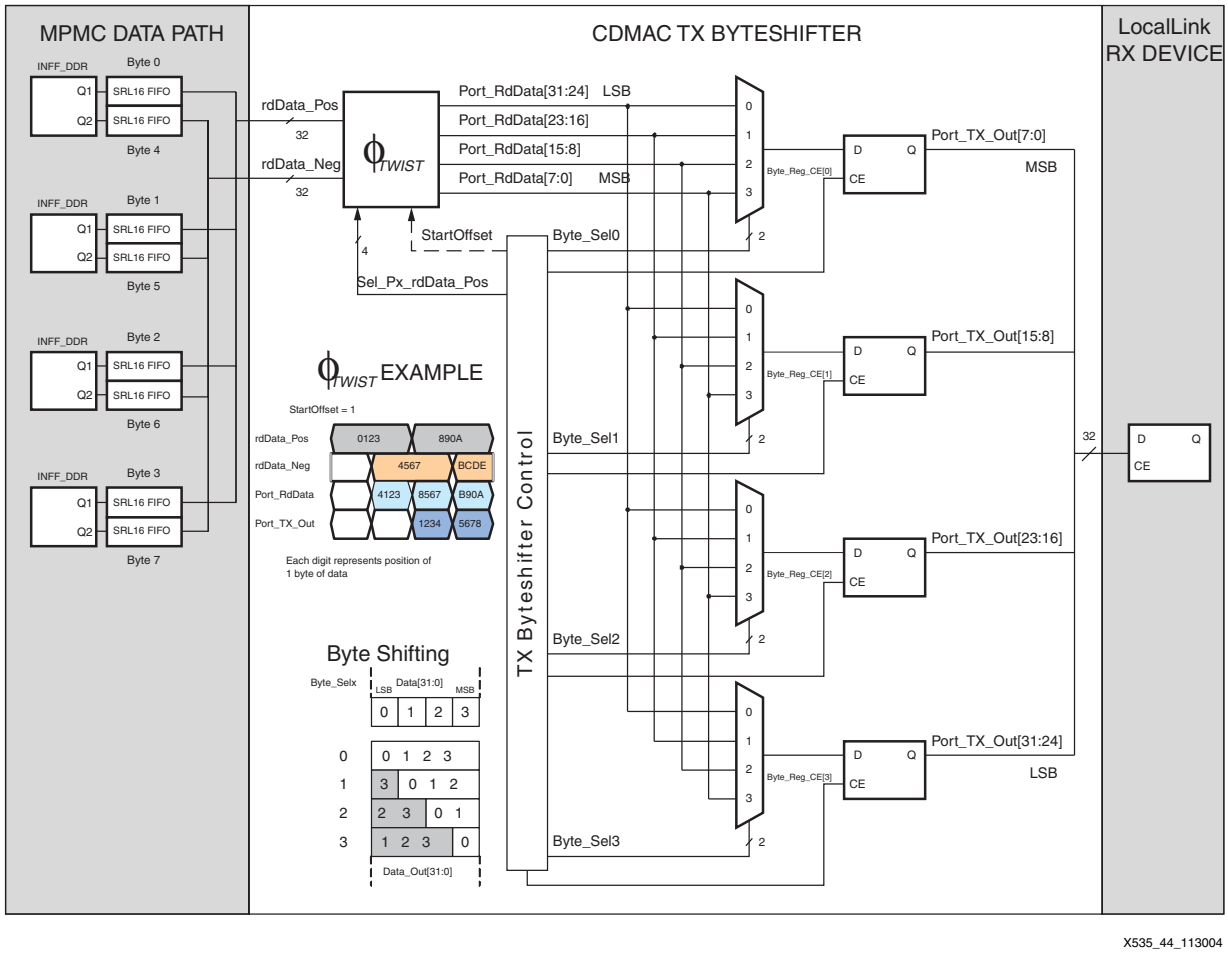


Figure A-18: **CDMAC Tx Byteshifter Block Diagram**

In this example:

- The vector `Byte_Sel[0-3` displays values of **`0 3 2 1`** respectively. This configuration of `Byte_Sel` vectors swap the first byte with the last three bytes to form (**`1 2 3 4`**) and (**`5 6 7 8`**).
- The `Byte_Reg_CE` clock enables the registers at the appropriate time.

For the first burst, `Byte_Reg_CE` is **`0xF`** until the last word.

For the last word, `Byte_Reg_CE` holds the left over bytes from the current burst in the registers by disabling clock(s) to the register(s).

For example, if `StartOffset` is **`1`** and `rdData_Neg`=**`0x4567`** is the last word of the burst, `Byte_Reg_CE[3:0]` is **`0x0001`**. In this case the last three bytes (**`5 6 7`**) is held in the registers until the next burst starts.

On the second burst, the first byte is loaded into the register enabled by `Byte_Reg_CE[0]`, then `Byte_Reg_CE` returns to **`0xF`** again until the last word of that burst.

Tx Byteshifter State Diagram

The Tx byteshifter state machine generates StartOffset, Byte_Reg_CE, Byte_Selx, Src_Rdy, and RdDataPop signals to control the Tx byteshifter data path, part of the LocalLink signals, and rdDataAcks to the memory.

- StartOffset controls the number of bytes to be shifted. It is set to the last three bits of the memory initially, but is changed at the end of every burst to account for the leftover bytes.
- Byte_Reg_CE is a 4-bit clock enable to the registers in the data path. It is multiplexed by the bytes being held in the byteshifter during a burst transition, as shown in Figure A-18, page 122.
- Byte_Selx control the multiplexers that are responsible for reordering the bytes coming out of fTWIST, as shown in Figure A-18.
- Src_Rdy indicates to LocalLink that the data is valid. Src_Rdy is asserted during the burst but deasserted while in the discard stage, the between descriptors stage, and the between bursts stage.
- RdDataPop generates rdDataAck, which is used to acknowledge data read from memory. rdDataAck_Pos and rdDataAck_Neg are asserted alternately. RdDataPop is asserted at the same time as Src_Rdy. RdDataPop is also asserted in the discard stage to pop out invalid data.
- Tx byteshifter state machine starts in IDLE state. When a Start signal is given by the port state machine, it goes into the discard stage and pops off data until it is at the current address.

Using the example in Figure A-17, page 120:

The byteshifter state machine discards the first 30 words of the first burst.

The state machine then moves to the START state if there is at least one complete word left in the burst, or to the STARTFINISH state if there is not a complete word.

The byteshifter state machine moves to the START state because there is a complete word ([**0 1 2 3**]). From the START state, it goes to:

- The PROCESS state, if there is not a complete word of data
- The FINISH state, if there is at least one more word of data

In the example, the state machine goes to FINISH state directly because there is not a second complete word of data.

In the STARTFINISH or FINISH state, the byteshifter state machine saves the leftover bytes by setting Byte_Reg_CE to disable clock(s) to the register(s) holding those bytes.

From either STARTFINISH or FINISH, the byteshifter state machine can go to BTWN_BURST state or BTWN_DESC state or IDLE state if, respectively:

- There is another burst for the same descriptor.
- Current descriptor is finished and the engine is moving on to the next descriptor in the chain.
- There are no more bursts and no more descriptors.

In all states the counters are reset.

For the example, the state machine goes to the BTWN_BURST state because it is still in the first descriptor.

The state machine behavior is as follows based on the current state:

- In `BTWN_BURST`, it goes to `EXTRA` to update `BurstLengthCount`, then to `START` state again.
- In `BTWN_DESC`, it returns to `DISCARD` state.

Figure A-19 provides the detailed information on the states and their inputs and outputs.

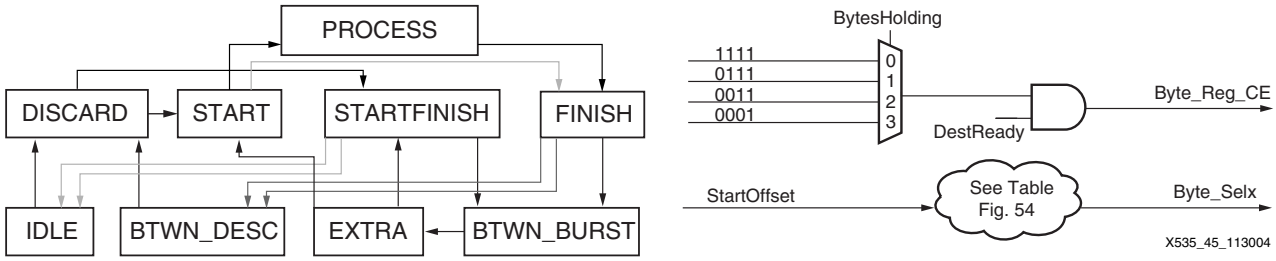


Figure A-19: **CDMAC Tx_Byte_Shifter_SM State Diagram**

Table A-1 provides a description of each Tx ByteShifter state.

Table A-1: **CDMAC Tx_Byte_Shifter_SM State Diagram Description**

STATE	PREVIOUS STATE	DESCRIPTION	INPUT STIMULUS	RESULTANT OUTPUT	INTERNAL OPERATIONS
IDLE (C)	INITIAL	Initializes values	RST	RdDataPop = 0 Src_Rdy = 0 StartOffset = 0 Used to generate Byte_Selx BytesHolding = 0 *Track bytes left over from previous burst. Used to generate Byte_Reg_CE*	BurstLengthCount = 128 - Address[1:0] *Total bytes need to be transferred in this burst*
	START FINISH		EndOfPacket *from Port SM. No more descriptors in the chain*		
	FINISH				
DISCARD	IDLE	Pop off invalid data from current burst until beginning of address	Start *Port SM gives permission for burst 16 read from MPMC2*	@CLK RdDataPop=1 Pop off invalid data Src_Rdy = 0 *NOT Ready signal to LL DEVICE* StartOffset= Address[2:0] Control byteshifter Multiplexers	DiscardDone = (PopCount == Address[6:2]-1) Signals end of discarding @CLK BurstLengthCount -= 4 *Update BurstLengthCount* @DiscardDone BurstLengthCount -= BytesNeeded BytesNeeded = 4 – *Bytesholding* *Adjust for leftover bytes from previous burst*
	BTWN_ DESC			StartOffset = Address[2:0]- BytesHolding *Adjust for leftover bytes*	
START	DISCARD	Leftover data+current data more than 1 word. Process 1st word of data	Discard Done *encounter 1st valid and complete word of current descriptor*	@DestReady RdDataPop=1 *Pop off 1 word of data* Src_Rdy = 1 *Ready signal to LocalLink DEVICE* BytesHolding=0 *Reset BytesHolding*	@CLK BurstLengthCount -= 4 *Update BurstLengthCount*
	EXTRA		CLK		
START FINISH	DISCARD	Leftover data+current data less than 1 word. Save this data	Length0Start *Length0Start = (BurstLengthCount <= BytesNeeded)* *Leftover data+current data less than 1 word*	@DestReady RdDataPop=1 *Pop off 1 word of data* *Src_Rdy = 1 (if BurstLengthCount=0)* *Ready signal to LocalLink DEVICE* BytesHolding = case(NextState): IDLE: 0; BTWN_BURST: -StartOffset; BTWN_DESC: BurstLengthCount *Adjust BytesHolding according to next state*	Rem = Case(BurstLengthCount): 0: 0b0000 1: 0b0111 2: 0b0011 3: 0b0001
	EXTRA				
PROCESS	START	Process data until last word	~Length0Middle *Length0Middle = (BurstLengthCount<=4)* *More than 1 word of data exist*	@DestReady RdDataPop=1 Pop off data Src_Rdy = 1 *Ready signal to LocalLink DEVICE*	BurstLengthCount -=4 @CLK *Update BurstLengthCount*

Table A-1: **CDMAC Tx_Byte_Shifter_SM State Diagram Description *(Continued)***

STATE	PREVIOUS STATE	DESCRIPTION	INPUT STIMULUS	RESULTANT OUTPUT	INTERNAL OPERATIONS
FINISH	START	Process last complete word of the burst if exists. Save remaining bytes to be combined with first few byte(s) of next burst	Length0Middle *Have reached the last word of the burst*	@DestReady RdDataPop=1 *Pop off last word of data* Src_Rdy = 1 (if BurstLengthCount=0) *Ready signal to LocalLink DEVICE* BytesHolding = case(NextState): IDLE: 0; BTWN_BURST: -StartOffset; BTWN_DESC: BurstLengthCount *Adjust BytesHolding according to next state*	Rem=case(BurstLengthCount) : 0: 0b0000 1: 0b0111 2: 0b0011 3: 0b0001
	PROCESS				
BTWN_ BURST (A)	START FINISH	Still more data to be transferred in current descriptor. Prepare for another burst. Resetting counters.	~LastBurst Length0Middle = *(BurstLengthCount<=4)* *More than 1 word of data exist*	RdDataPop = 0 Src_Rdy = 0	BurstLengthCount = 128 *Reset BurstLengthCount*
	FINISH				
EXTRA	BTWN_ BURST	Update BurstLengthCount to combine with left over data from last burst	Start *Port SM gives permission for burst 16 read from MPMC2*	RdDataPop = 0 Src_Rdy = 0	BurstLengthCount -= BytesNeeded BytesNeeded = 4 – *Bytesholding* *Adjust for leftover bytes from previous burst*
BTWN_ DESC (B)	START FINISH	More descriptors to process. resetting values. Similar to IDLE	~EndOfPacket *Still more descriptors in the chain*	RdDataPop = 0 Src_Rdy = 0	BurstLengthCount = 128 address[1:0] *Total bytes need to be transferred in this burst*
	FINISH				

LocalLink Tx State Machine

The LocalLink Tx state machine shown in Figure A-20 represents the steps to transfer data from the memory to the LocalLink as directed by the descriptor.

- From the `IDLE` state, the port state machine instructs the LocalLink Tx interface to issue a Start of Frame.
- When the `Dst_Rdy` signal is asserted by the LocalLink device, the LL Tx state machine goes into the `HEADER` state to transfer a CacheLine-8 read of the descriptor to the LocalLink as the header.
- When port state machine issues a `Read_Desc_Done` signal, the LL Tx State goes in to `PAYLOADSTART` and issues an `SOP` signal. It immediately goes to the `PAYLOAD` state if there is more than one word of payload, or to `PAYLOADEND` if there is less than one word of payload.
- From `PAYLOAD` state, when it reaches the last word, the Tx byteshifter issues a `Trigger_EOP` signal that brings it to the `PAYLOADEND` state. The Tx byteshifter issues an `EOP`, then goes to the `FRAMEEND` state because there is no footer in Tx transfer.
- The Tx byteshifter issues an end of frame in the `FRAMEEND` state then goes back to the `IDLE` state.

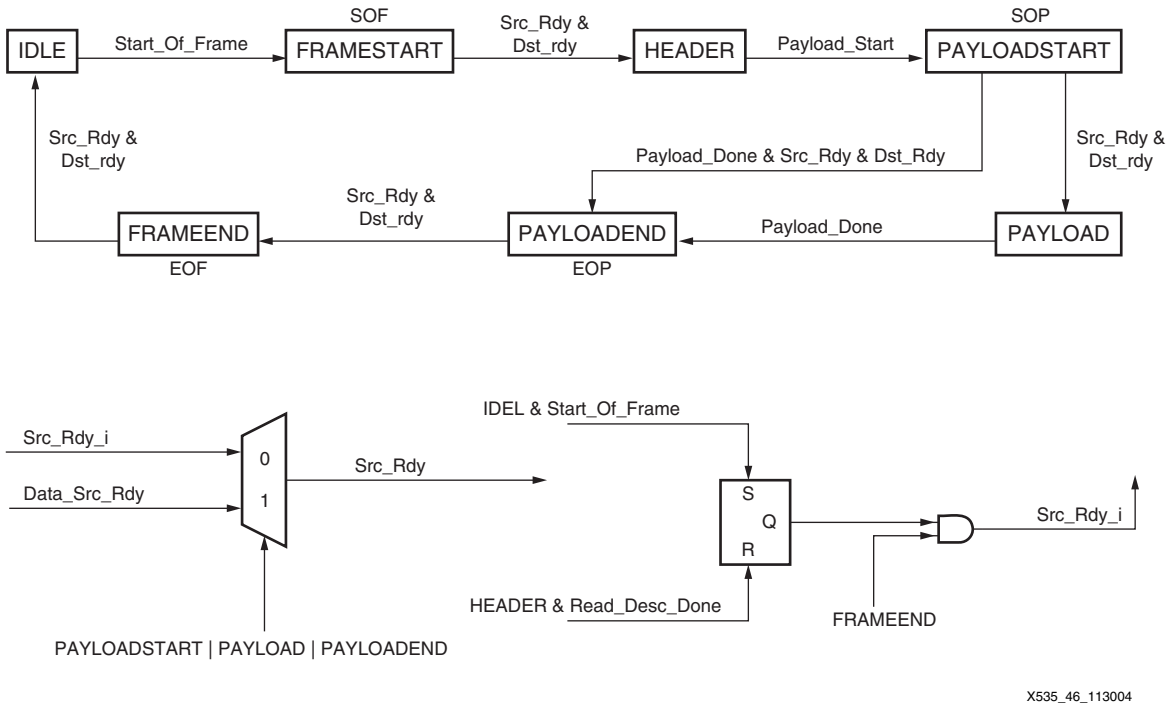


Figure A-20: **CDMAC Tx_LL_SM State Diagram**

Rx LocalLink and Byteshifter

The LocalLink and byteshifter Rx logic receive data from the LocalLink interface and move the data to the appropriate place in memory. This concept is illustrated in Figure A-21, page 127. The CDMAC always ignores the Rx LocalLink header. The payload is processed by the Rx byteshifter and pushed into the MPMC2 write FIFOs.

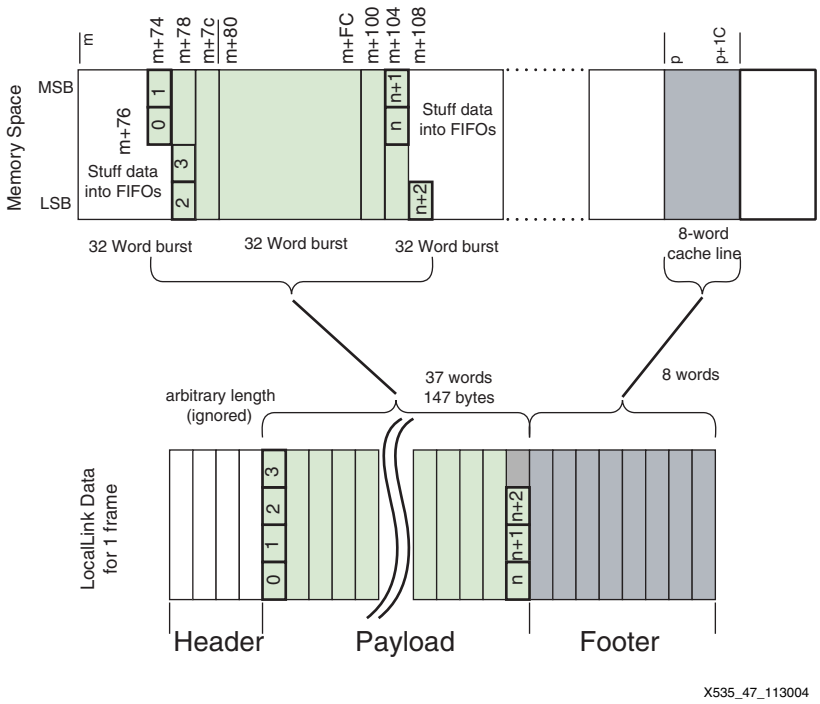


Figure A-21: **CDMAC Rx Byte Shift Example**

In this example, the payload is 147 bytes.

The data is pushed into the FIFOs in bursts of 32-words (`B32W`).

Data is stuffed into the FIFOs from address m through address **`m+0x75`** because the payload is written to address **`m+0x76`**, which is a not a 32-word aligned address. When these bytes are written to memory, the byte enables are turned off.

Ten bytes of payload data are pushed into the FIFOs after this data is stuffed into the FIFOs.

In the second `B32W`, all of the data is valid.

In the last `B32W`, only nine bytes need to be written to memory. This means that the remaining 119 bytes must be stuffed into the FIFOs at the end of the burst.

After the payload has been processed, the footer is processed and written to memory at address **`p`**.

The CDMAC changes the byte enables of first three words to prevent the next descriptor pointer, buffer address, and buffer length from being overwritten.

Rx Byteshifter Logic

The Rx byteshifter Data Path shown in Figure A-22, page 129 is controlled by the following signals: `First_offset`, `Pos_CE`, and `Neg_CE`.

First_offset

The `First_offset` signal is three-bits wide and it represents the number of bytes by which the LocalLink data needs to be shifted and is used as select bits to the Rx byteshifter's data path multiplexers.

For example, if `First_offset` is set to **`0x6`**:

- The first two bytes of the LocalLink data are presented to the last two bytes of the `WrDataBus_Pos` register and to the last two bytes of the `WrDataBus_Neg` register.
- The last two bytes of the LocalLink Data is presented to the first two bytes of the `WrDataBus_Pos` register and to the first two bytes of the `WrDataBus_Neg` register.

Pos_CE

If `Pos_CE` is asserted while `First_offset` is set to **`0x6`**, the clock enables on the last two bytes of the `WrDataBus_Neg` register and the first two bytes of the `WrDataBus_Pos` register are asserted.

Neg_CE

If `Neg_CE` is asserted while `First_offset` is set to `0x6`, the clock enables on the last two bytes of the `WrDataBus_Pos` register and the first two bytes of the `WrDataBus_Neg` register are asserted.

Using the example in Figure A-21, page 127 where the address offset is `0x76` and the length is set to `0d147` (which implies that `First_offset` is set to `0x6`):

- Stuff 28 words of data into the MPMC2 write FIFOs by asserting `WrDataAck_Pos` and `WrDataAck_Neg` on alternating clock cycles. `WrDataBE_Pos` and `WrDataBE_Neg` are set to `0b1111`.
- In clock cycle `N+1`, the `WrDataAck_Pos` signal to the MPMC2 is asserted with `WrDataBE_Pos` set to `0b1111`. `Pos_CE` is asserted. Two bytes of data are clock-enabled into `WrDataBus_Neg` and two bytes of data are clock-enabled into `WrDataBus_Pos`.
- In clock cycle `N+2`, the `WrDataAck_Neg` signal is asserted with `WrDataBE_Neg` set to `0b1100`. The last two bytes of `WrDataBus_Neg` are valid from clock cycle `N+1`. At the same time, `Neg_CE` is asserted to clock-enable two bytes into `WrDataBus_Pos` and two bytes into `WrDataBus_Neg`.
- In clock cycle `N+3`, the `WrDataAck_Pos` signal is asserted with `WrDataBE_Pos` set to `0b0000`. The first two bytes of `WrDaBus_Pos` is valid from clock cycle `N+1` and the last two bytes of data are valid from clock cycle `N+2`. `Pos_CE` should also be asserted in this clock cycle to continue collecting data from the LocalLink interface. This continues until there is no more data, or until `First_offset` needs to change.
- In clock cycle M, the `WrDataAck_Neg` signal is asserted with `WrDataBE_Neg` set to 0b0000. `Neg_CE` does not need to be asserted, as all data has been collected from the LocalLink interface.
- In clock cycle `M+1`, the `WrDataAck_Pos` signal is asserted with `WrDataBE_Pos` set to `0b0111`. `Pos_CE` does not need to be asserted.

- Stuff 29 words of data into the MPMC2's write FIFOs by asserting `WrDataAck_Neg` and `WrDataAck_Pos` on alternating clock cycles. `WrDataBE_Pos` and `WrDataBE_Neg` are set to `0b1111`.

To use this data path efficiently, `First_offset` must be held constant while `Pos_CE` and `Neg_CE` are asserted on opposite clock cycles.

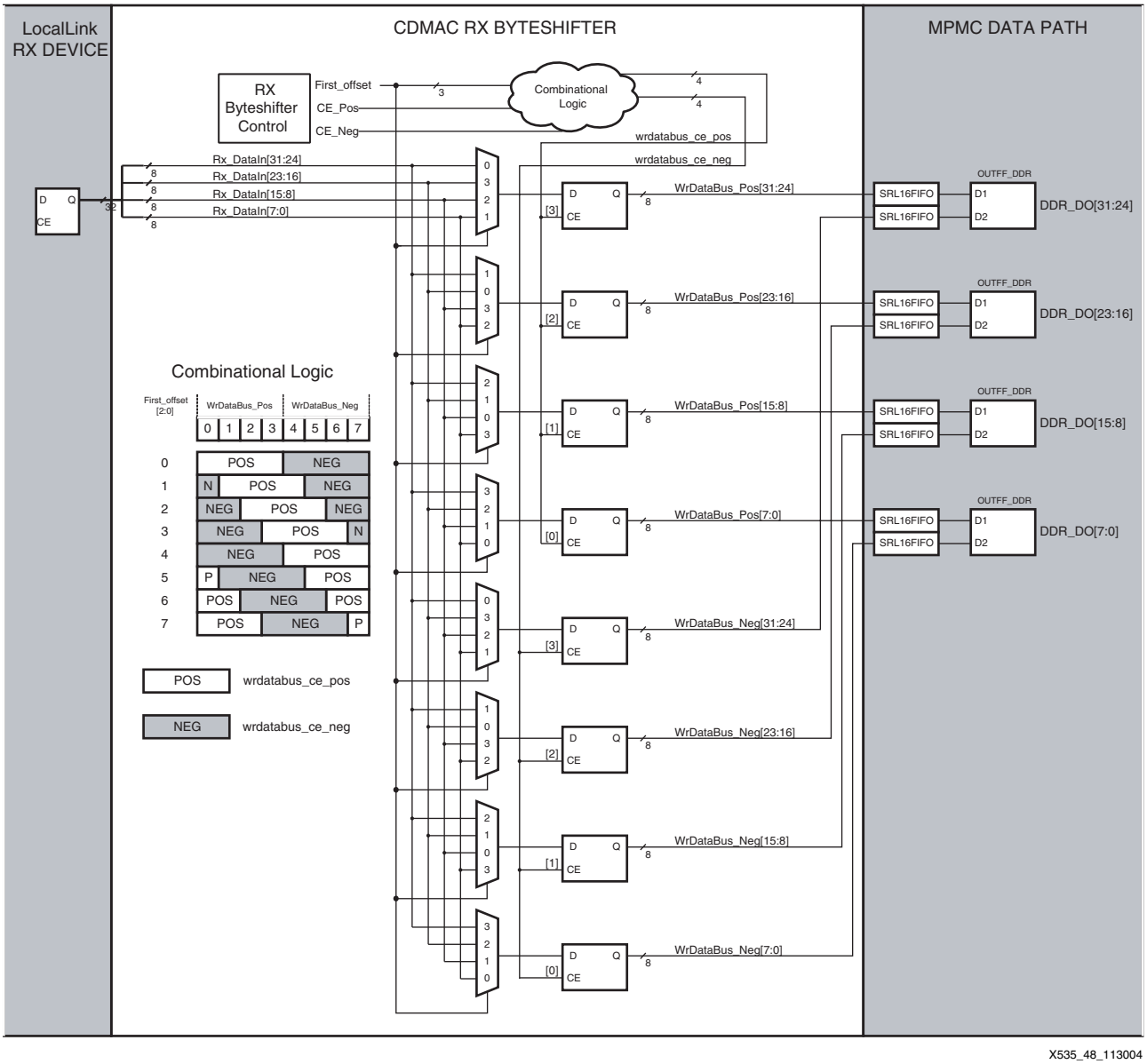


Figure A-22: **CDMAC Rx Byte Shifter Block Diagram**

Rx Byteshifter State Diagram

In the case of the CDMAC, the First_offset signal is a 3-bit number representing the number of bytes that the LocalLink Data needs to be shifted by before it is pushed into the MPMC2 write FIFOs. This value is simply the three Least Significant Bytes (LSBs) in the descriptor's buffer address. The First_offset signal must be valid when the port state machine issues a B32W request or an CL8W request.

- Pos_CE is used to push even words from the Rx LocalLink interface into the Rx byteshifter. The First_offset signal is used as select bits to the Rx byteshifter data path multiplexers. Combinational logic is used to clock-enable the correct registers so that the data from the Rx LocalLink interface is placed into the correct memory location.
- Neg_CE is used to push odd words from the Rx LocalLink interface into the Rx byteshifter. The word numbering starts at zero, the first word of the payload, and is reset to zero again at the first word of the footer.
- Pos_CE and Neg_CE are asserted for every payload and footer word that is received from the LocalLink interface.

The Rx byteshifter control state machine represents the way Pos_CE and Neg_CE are generated and is shown in Figure A-23.

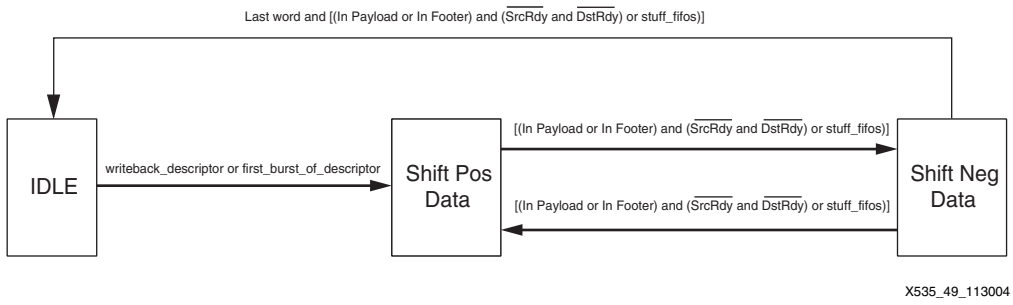


Figure A-23: **CDMAC LocalLink and byteshifter Diagram**

Table A-2 describes the state operations of the Rx ByteShifter.

Table A-2: **CDMAC LocalLink and byteshifter Diagram Description**

STATE	PREVIOUS STATE	DESCRIPTION	INPUT STIMULUS	RESULTANT OUTPUT
IDLE	Initial	Waits for 32-word burst write request or for 8-word cache-line write request.		Pos_CE = 0 Neg_CE = 0
	Shift Neg Data			
Shift Pos Data	IDLE	Instructs even words from the LocalLink interface to be pushed into the RX Byte Shifters.	byte_shift_ce_pos byte_shift_ce_pos = Even Word Data Valid on the LocalLink interface and (In header or In footer)	Pos_CE = byte_shift_ce_pos Neg_CE = 0
	Shift Neg Data			
Shift Neg Data	Shift Pos Data	Instructs odd words from the LocalLink interface to be pushed into the RX Byte Shifters.	byte_shift_ce_neg byte_shift_ce_neg = Even Word Data Valid on the LocalLink interface and (In header or In footer)	Pos_CE = 0 Neg_CE = byte_shift_ce_neg

Initially, the Rx byteshifter control state machine starts in the IDLE state.

When the CDMAC is ready to push payload data into the MPMC2, the port state machine instructs the Rx byteshifter control logic to transition to the In Header state and to start pushing data from the LocalLink interface through the Rx byteshifter.

Each time Src_Rdy and Dst_Rdy are asserted together, data is pushed into the Rx byteshifter and the Rx byteshifter control state machine toggles between the Shift Pos Data state and the Shift Neg Data state.

All 32-word burst transfers to the MPMC2 must be 32-word address aligned, so data can be stuffed into the MPMC2 write FIFOs with the byte enables deasserted. This only needs to occur at the beginning of the payload or at the end of the payload. See Figure A-21, page 127.

If data needs to be stuffed into the FIFOs, data is processed by the Rx byteshifter, which also causes the Rx byteshifter state machine to toggle between the Shift Pos Data state and the Shift Neg Data state.

When the descriptor buffer length has reached zero bytes and the last word has been pushed into the MPMC2 write FIFOs, the byteshifter control state machine transitions back to the IDLE state.

The two exceptions to this event description occur when the port state instructs the Rx byteshifter control logic to push footer data into the FIFOs instead of the payload data as described:

- Because the address is always aligned, data never needs to be stuffed into the FIFOs.
- Exactly 8 words of data are pushed into the FIFOs.

The write control logic going to the MPMC2 parallels the Rx byteshifter logic. There are four signals that go to the MPMC2: WrDataAck_Pos, WrDataAck_Neg, WrDataBE_Pos, and WrDataBE_Neg.

The WrDataAck logic is shown in Figure A-24, page 132. As payload or footer data comes across the Rx LocalLink interface or as data is stuffed into the MPMC2 write FIFOs:

- WrDataAcks are asserted to match the data coming out of the Rx byteshifter data path.
- WrDataBEs are also asserted to match the data coming out of the Rx byteshifter data path.

To write to memory location m+0x74:

- WrDataAck_Neg must be asserted with WrDataBE_Neg holding the value 0b1100.
- WrDataAck_Pos must be asserted with WrDataBE_Pos holding the value 0b0000 to write to the m+0x78 memory location.
- WrDataAck_Pos must be asserted at the appropriate time with WrDataBE_Pos holding the value 0b0111 to write to the m+0x108 memory location.

Refer to Figure A-21, page 127 in the "Rx Byteshifter Logic"section.

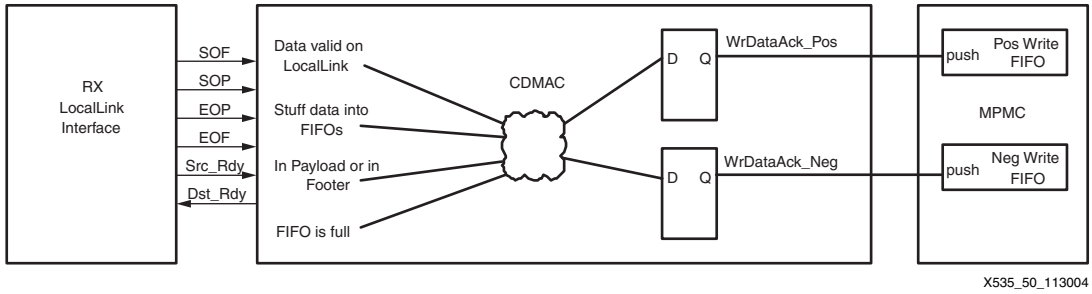


Figure A-24: **CDMAC Rx WrDataAck Logic to MPMC2**

LocalLink Rx State Machine

The LocalLink Rx state machine represents the steps to process the LocalLink data and transfer it into memory.

- From the `IDLE` state, the port state machine instructs the LocalLink Rx interface to start processing the header data by asserting the `LL_Start` signal.
- The CDMAC ignores the header data and the `Dst_Rdy` signal is asserted until `SOP` is asserted. At this point the LocalLink interface is presenting the first word of payload data and the LocalLink Rx state machine stays in the `In Payload IDLE` state until the port state machine asserts `B32_Start`, indicating that 32-words of data can be pushed into the MPMC2 write FIFOs.
- The LocalLink Rx state machine transitions to the `In Payload` state when the `B32_Start` signal is asserted. The state machine stays in this state until 32 words have been pushed into the MPMC2 FIFOs.

Using the example in Figure A-25, page 133:

- 118 bytes are stuffed into the MPMC2 write FIFOs, then `Dst_Rdy` is asserted until three words have been processed by the Rx byteshifter.
- The state machine transitions back to the `In Payload IDLE` state until the port state machine asserts `B32_Start` again.
- While in the `In_Payload` state, the Rx byteshifter processes 32 more words then the state machine transitions back into `In Payload IDLE` state.
- The port state machine again issues the `B32_Start` signal.
- When the LocalLink Rx state machine is in the `In_Payload` state, the Rx byteshifter processes two more words, then 119 bytes are stuffed into the MPMC2 write FIFOs.
- The state machine transitions back to the `In Payload IDLE` state.
- The port state machine transitions to the `In_Payload_IDLE` state. Because all of the payload data has been received (the `EOP` has been acknowledged by `Src_Rdy` and `Dst_Rdy`) and there are no bytes left to send to the MPMC2 write FIFOs, the port state machine asserts `CL8_Start` when it is ready to write back the descriptor.
- The LocalLink Rx state machine transitions to the `In Footer` state, processes the footer, and sends data to the MPMC2 write FIFOs.
- The CDMAC overrides the byte-enables on the first three words to prevent the next descriptor pointer, the buffer address, and the buffer length from being overwritten.

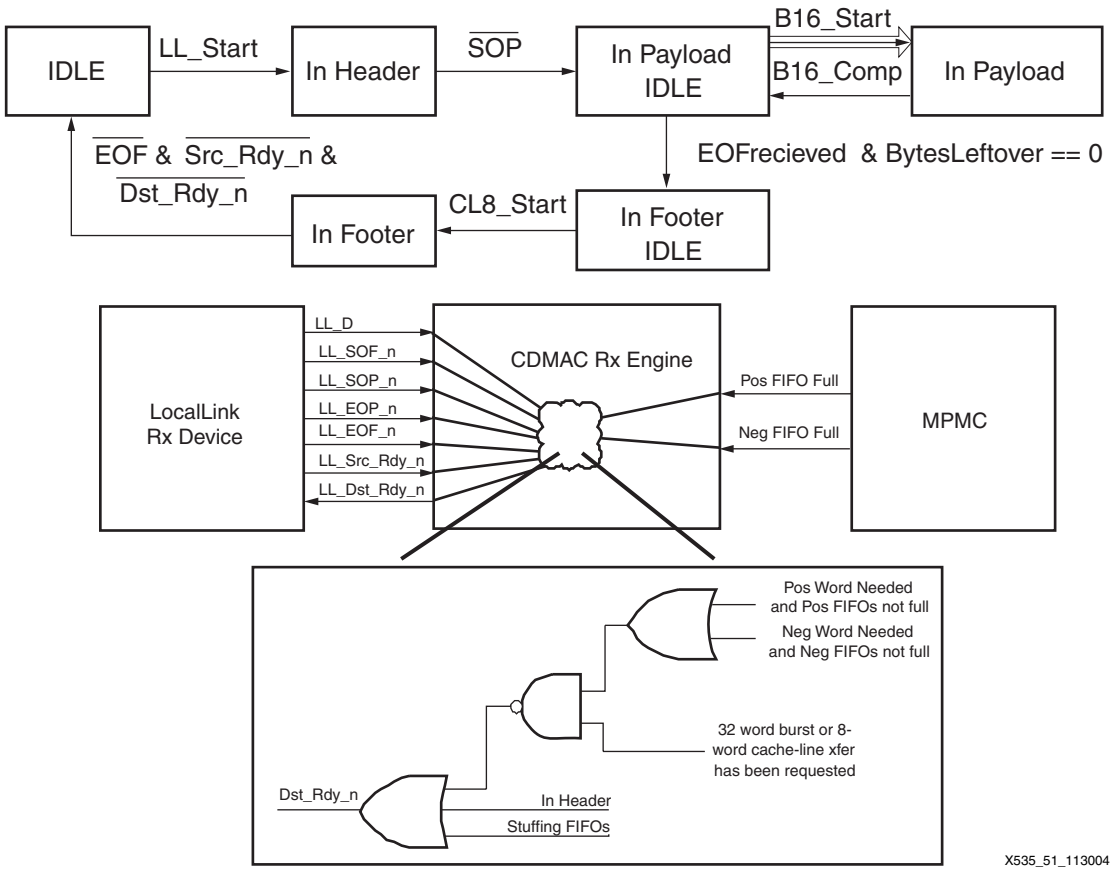


Figure A-25: **CDMAC Rx LocalLink State Machine & Dst_Rdy Logic**

Regfile Arbiter State Machine

The register file can be accessed by the DCR interface and each of the engines. The DCR interface issues DCR requests and the port state machine issues requests for the TX and RX engines. The Regfile Arbiter grants one of the five at a given time.

The state diagram shown in Figure A-26, page 135 represents a conceptual overview of the Regfile Arbiter logic.

- The DCR interface always has priority access to the register file because of the short duration of DCR access.
- If only one request is issued, it is granted as soon as Regfile is not busy.
- If DCR and the port state machine both issue requests, DCR is granted access when Regfile is not busy.
- If only port state machine issues requests, one of the engines is granted access based on which engine was granted last time.
- When DCR is not issuing a request, Regfile Arbiter arbitrates between the two engines as shown in the two interlocked circles in Figure A-26, page 135. Prev holds the value of the last grant, defaulted to `P0W` at reset. The two engines have circular priority in the order [`P0R`, `P0W`].
  For example, if `P0W` was granted last time (default), then for the next grant, `P0R` has priority `P0W`. If `P0R` was granted last time, then the new priority queue becomes [`P0W`, `P0R`].
- When DCR issues a request, DCR has priority over both engines and is granted access. However, the previous grant is still in effect, which means when DCR finishes and does not issue a new request, the arbiter returns to arbitration of the engines based on the last engine granted.
  For example, if `P1R` was the last grant and both engines issue requests plus the DCR, DCR gets the next grant, followed by `P0W`, because the priority queue at this time is [`P0W`, `P0R`].

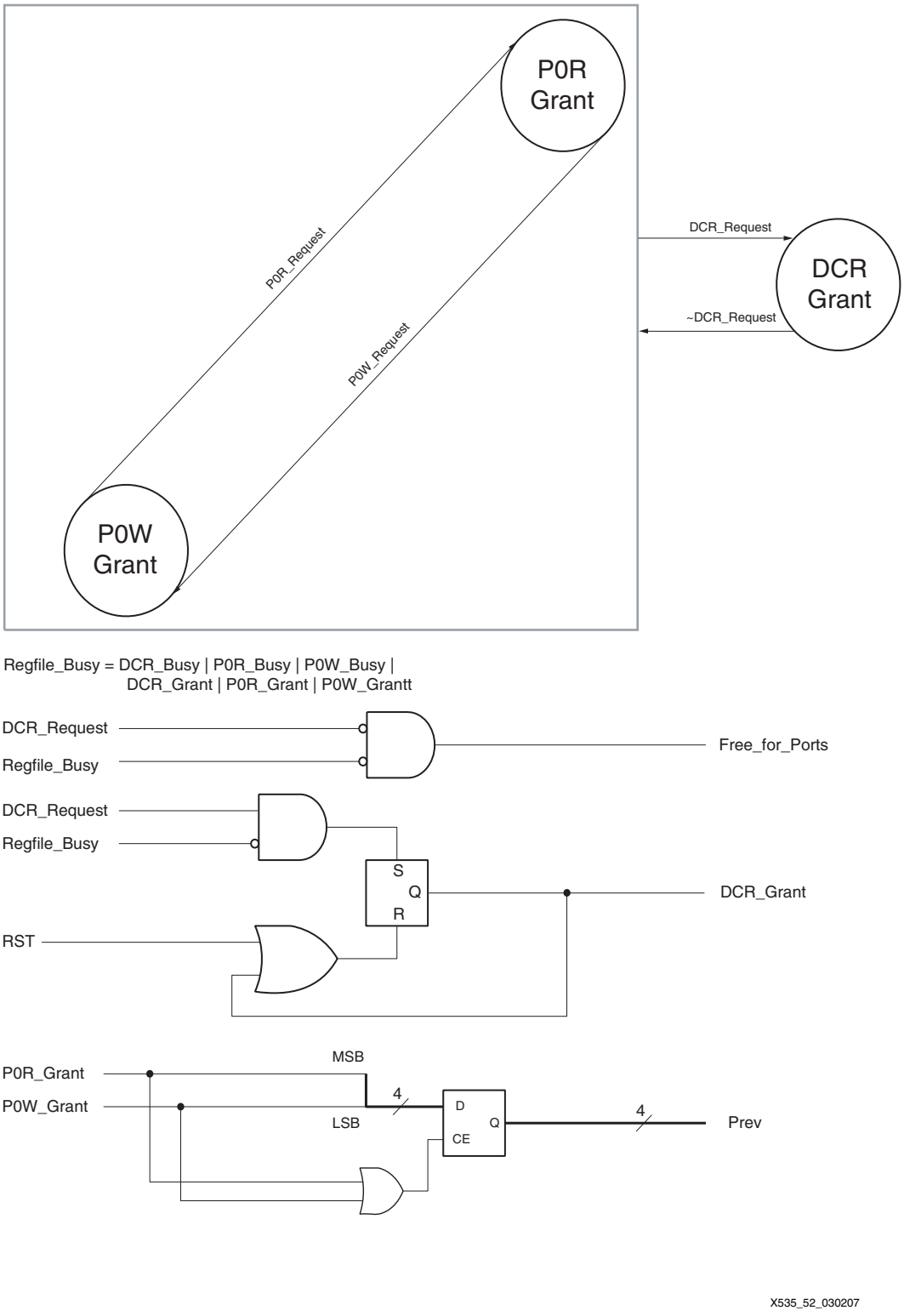


Figure A-26: **Regfile_Arb_SM State Diagram**

Status Register Logic

The CDMAC STATUS register uses bits zero through six to configure the CDMAC engine and send status information. Bit seven is reserved, and the other 24 bits are used for application-defined data. The logic to generate the STATUS register bits is illustrated in Figure A-27.

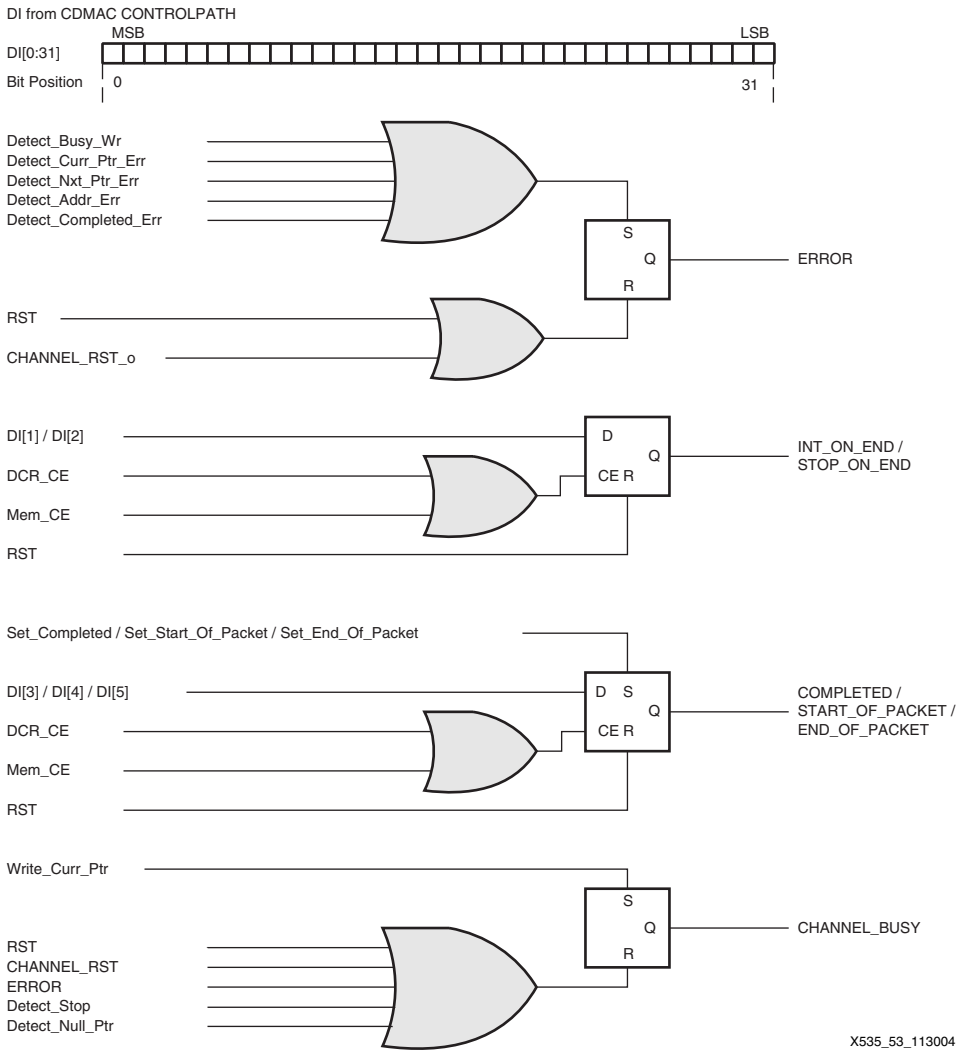


Figure A-27: **CDMAC Status Register**

ERROR Bit

The ERROR bit is set when one or more of the error conditions occur. It can only be reset by a system or channel reset. Refer to the list of known issues. The error conditions are:

- The CPU issues a DCR_write to the current descriptor pointer when the BUSY bit is set.
- The current descriptor pointer points to an invalid location in memory.
- The next descriptor pointer points to an invalid location in memory.
- The engine attempts to read or write from an invalid location in memory.
- Completed error checking is enabled and the COMPLETED bit is set in the descriptor.

Interrupt On End Bit

The Interrupt On End (INT_ON_END) bit is clock-enabled into the STATUS register during a DCR_write to the STATUS register or while the CDMAC is reading the STATUS register of the descriptor from memory.

Stop On End Bit

The STOP_ON_END bit is clock-enabled into the STATUS register during a DCR_write to the STATUS register or when the CDMAC is reading the STATUS register of the descriptor from memory.

Completed Bit

The COMPLETED bit is set whenever all of the data for the descriptor has been processed. This occurs before the descriptor is written back to memory. The COMPLETED bit is clock- enabled into the STATUS register during a DCR write to the STATUS register or while the CDMAC is reading the STATUS register of the descriptor from memory.

Start Of Packet, Start of Payload, End of Packet, End of Payload Bits

On an RX engine:

- The START_OF_PACKET bit is set when the LocalLink interface receives the Start_Of_Payload (SOP) signal. The START_OF_PACKET bit is clock-enabled into the STATUS register during a DCR_write to the STATUS register or when the CDMAC is reading the STATUS register of the descriptor from memory for both TX and RX engines.
- The END_OF_PACKET bit is set when the LocalLink interface receives the End_Of_Payload (EOP) signal. The END_OF_PACKET bit is clock-enabled into the STATUS register during a DCR_write to the STATUS register or when the CDMAC is reading the STATUS register of the descriptor from memory for both TX and RX engines.

Channel Busy Bit

The CHANNEL_BUSY bit is set when the CPU issues a DCR write to the current descriptor pointer.

Channel Reset Bit

The CHANNEL_RST bit is reset under the following conditions:

- System or channel reset. Refer to the list of known issues in root directory of ZIP file.
- The ERROR bit is set.
- The descriptor has been processed and the STOP_ON_END bit is set.
- The descriptor has been processed and the next descriptor pointer contains the NULL pointer.

Interrupt Register Logic

The INTERRUPT register controls the interrupts to the CPU. Figure A-28, page 139 shows the Interrupt register logic. The Most Significant Bit (MSB), bit 0, is the master interrupt enable. If this bit is asserted, interrupts become visible to the CPU. The CDMAC set the four Least Significant Bits (LSBs), bits 28-31, to indicate that the CPU must handle an interrupt. The RX_Int_Detect and TX_Int_Detect represent these bits.

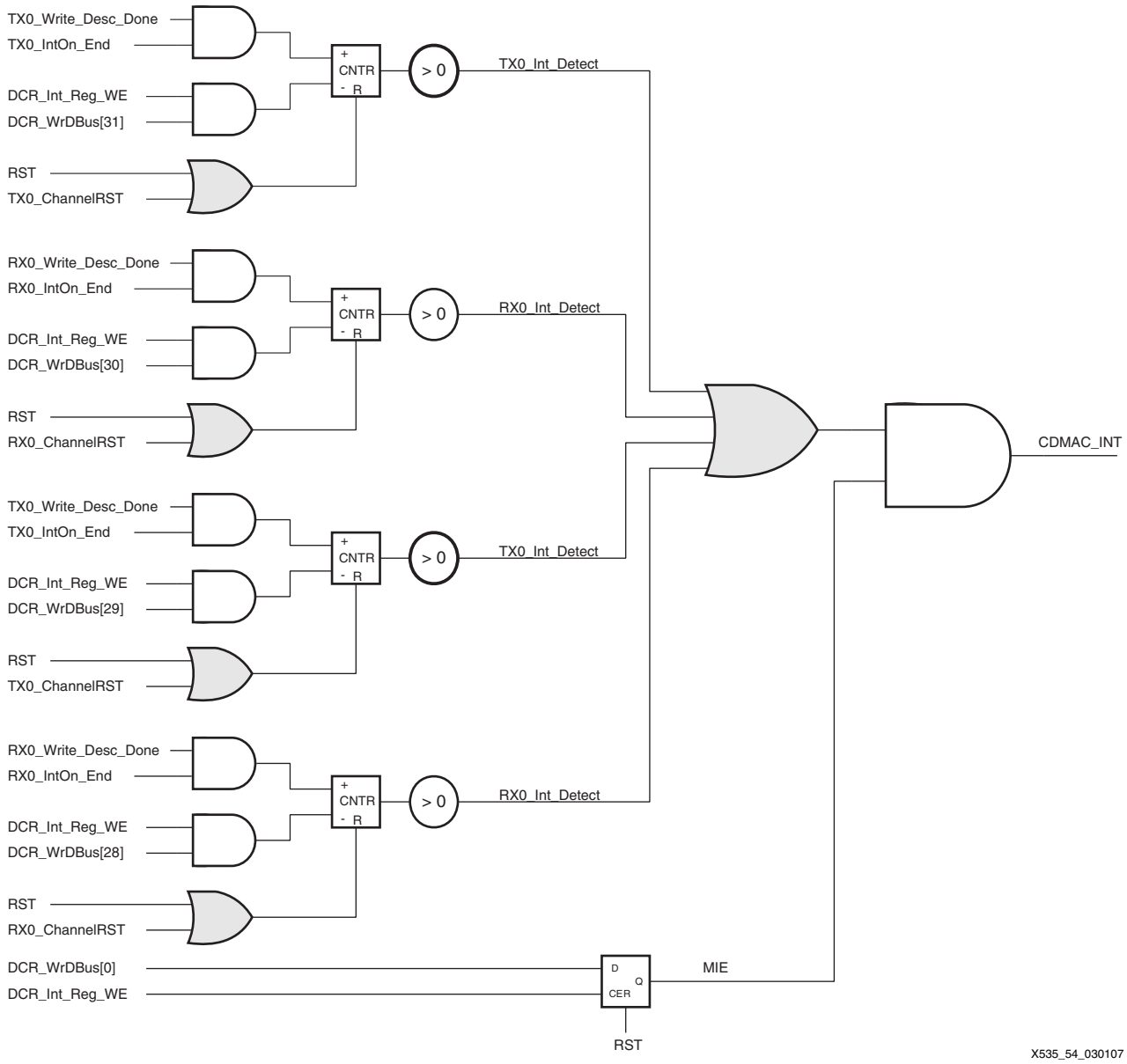


Figure A-28: **CDMAC Interrupt Register Logic**

The master interrupt enable is set or reset through `DCR_Writes` to the Interrupt register.

The interrupt detect signal is controlled by an up and down counter that counts up as interrupts are received from the CDMAC and counts down as the CPU processes each interrupt. The interrupt detect signal remains asserted as long as the counter is greater than zero. This method is one way to verify that the CPU is keeping up with the CDMAC.

The CDMAC issues an interrupt if an engine has written back a descriptor and the descriptor's `INT_ON_END` bit was asserted. If this happens on the TX engine, bit 31 of the Interrupt register is set. The CPU acknowledges the interrupt on the TX engine by issuing a `DCR_write` to the Interrupt register, writing a logical `1` to bit 31.

LocalLink Interface Usage

The CDMAC has two DMA engines. Each DMA engine is associated with a LocalLink interface. One DMA engine is used for transmitting, and one is used for receiving. A single transmit DMA engine is paired up with a single receive DMA engine to form a full duplex communication channel in the CDMAC. Each full duplex communication channel occupies the MPMC2 port. See Figure A-1, page 96 and Figure A-9, page 110 for simplified CDMAC structure diagrams.

- The LocalLink interface provides for the ability to transmit encapsulated data. The data itself is embedded in a package that starts with a header and ends with a footer.
- The `START_OF_FRAME` signal initiates the header of the package. Between the time this signal starts, and the time the `START_OF_PAYLOAD` signal occurs, the header of the package is being transmitted.
- Between the `START_OF_PAYLOAD` and `END_OF_PAYLOAD` signal, the data of the package is being transmitted.
- Information transmitted between the `END_OF_PAYLOAD` and `END_OF_FRAME` signal delete constitutes a header. In this way, the LocalLink interface permits the encapsulation of data content into a standardized package. The CDMAC uses this package to communication extra control information.
- The transmit DMA engine uses the header to broadcast the first DMA descriptor of the DMA process to the device on the other end of the LocalLink interface.
- The DMA descriptor contains flag information that communicates how the CDMAC processes the descriptor, specifically the `START_OF_PACKET` and `END_OF_PACKET` bits within the CDMAC `STATUS` field.

Figure A-29 provides an example of the LocalLink interface.

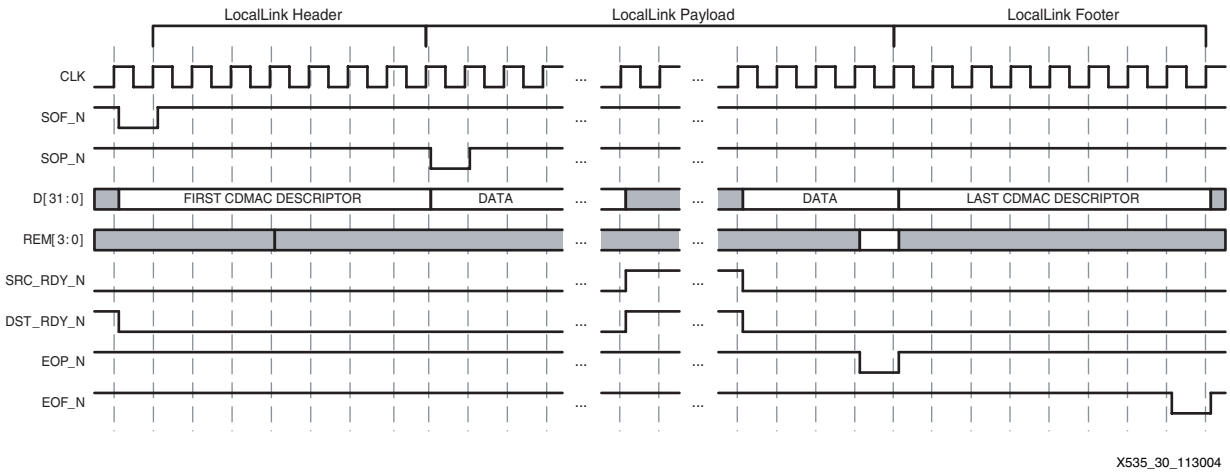


Figure A-29: **CDMAC LocalLink Interface General Purpose Example**

LocalLink Tx Interface

Figure A-30 shows an example of the LocalLink Tx interface in the CDMAC Tx DMA engine. This interface provides read data from the CDMAC. One important aspect of the communication style of DMA is that it depends on the use of streaming data interfaces, and consequently, it has no address context.

Data transfers across the interface when both sides transmit RDY signals.

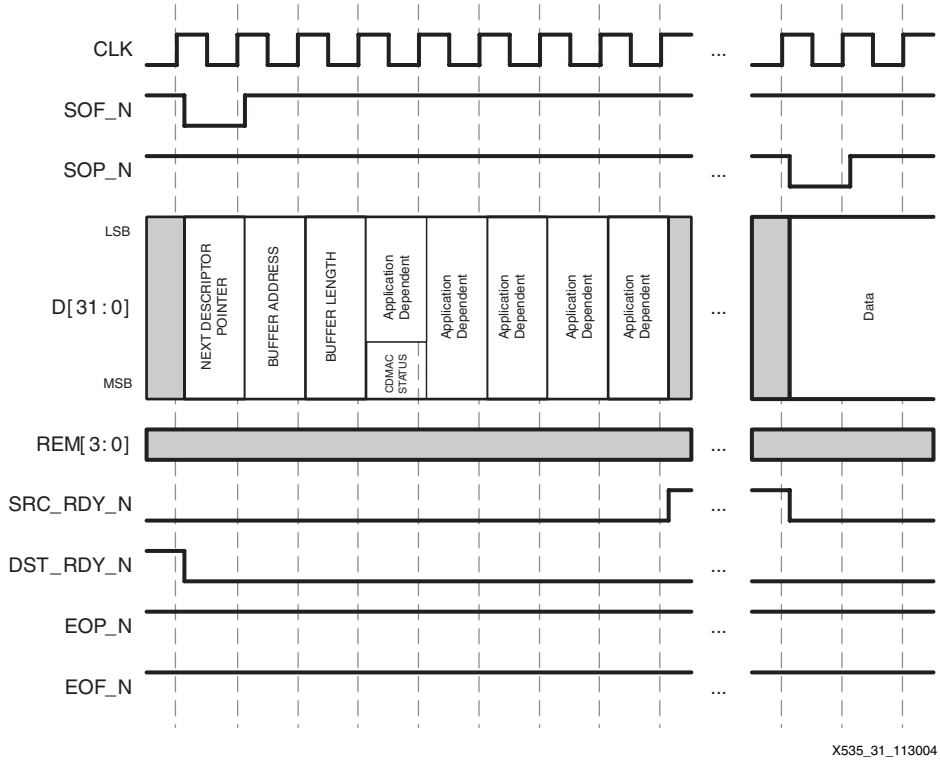


Figure A-30: **CDMAC LocalLink Tx Interface**

- When the CDMAC encounters a TX descriptor with the START_OF_PACKET bit set, it initiates a header transaction across the LocalLink interface and moves the data according to that DMA descriptor. Because the CDMAC allows for a chain of DMA descriptors on a per engine basis, the CDMAC can have some or all of the data contained within that first descriptor.
- If it is all contained in the first descriptor the END_OF_PACKET bit is also set with the descriptor CDMAC STATUS field.
- If there is more data to be transferred, perhaps using a different data buffer, the CDMAC runs its buffer length to zero then retrieves another DMA descriptor.
- Data continues to be transferred across the LocalLink interface during this time, because the CDMAC moves the data from memory to the interface.
- When CDMAC encounters a DMA descriptor whose END_OF_PACKET bit is set, the CDMAC closes down the LocalLink interface by outputting the footer field.
- During CDMAC Tx operations, the footer field is not used. It is intended to be used during receiving only.

LocalLink Rx Interface

Figure A-31 shows the CDMAC LocalLink Rx interface.

- Where the Tx CDMAC engine transmits real information during the header and bogus information during the footer, the Rx transmits bogus information during the header and real information during the footer.
- The CDMAC Rx engine ignores information from the device during the header, but takes the information broadcast from the footer and writes that to memory as part of the last DMA descriptor for that Rx channel.

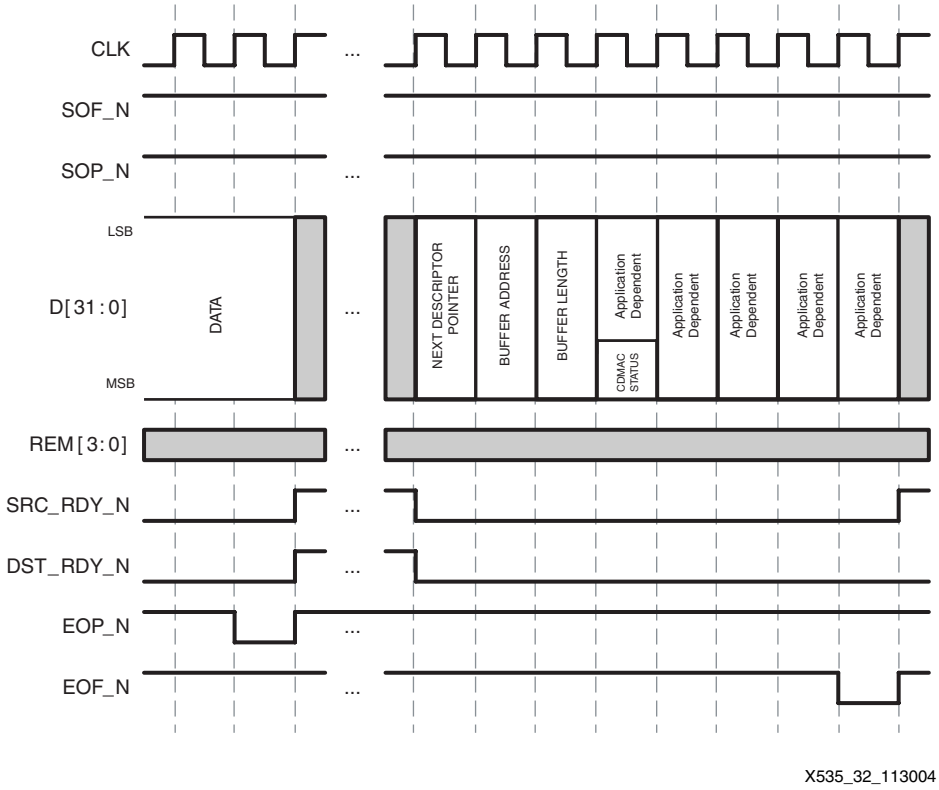


Figure A-31: **CDMAC LocalLink Rx Interface**

The Tx and Rx engines use the DMA descriptors in different ways.

For Tx, if there is a chain of DMA descriptors, only the first DMA descriptor in that chain is broadcasted as header across the LocalLink interface until a DMA descriptor is encountered which contains a `END_OF_PACKET` flag, wherein the process repeats.

In Rx, when there is a chain of descriptors, the current DMA descriptor has its application-dependant data written from the information contained when a footer is broadcasted.

The Tx DMA engine controls when the Tx LocalLink interface sees headers and footers by the `START_OF_PACKET` and `END_OF_PACKET` flags.

The Rx LocalLink interface controls when the Rx DMA engine marks these flags in the current DMA descriptors it is processing. When the Rx engine identifies an `END_OF_PACKET`, it also updates the contents of the DMA descriptor with the footer information into the application-defined areas.

Shared Resources

To conserve FPGA resources, the CDMAC uses an implementation technique to share resources. The registers for the CDMAC from DCR base address `0x00` to `0x0F` are not actual registers; they are entries into a LUT RAM that is organized 16-bits deep by 32-bits wide.

The LUT RAM forms a register file as illustrated in Figure A-32. The register file would consume an enormous number of flip-flops unless implemented as LUT RAM. The whole register file only consumes 16 LUTs. One issue with this kind of structure is that the LUT RAM cannot access every 'register' simultaneously.

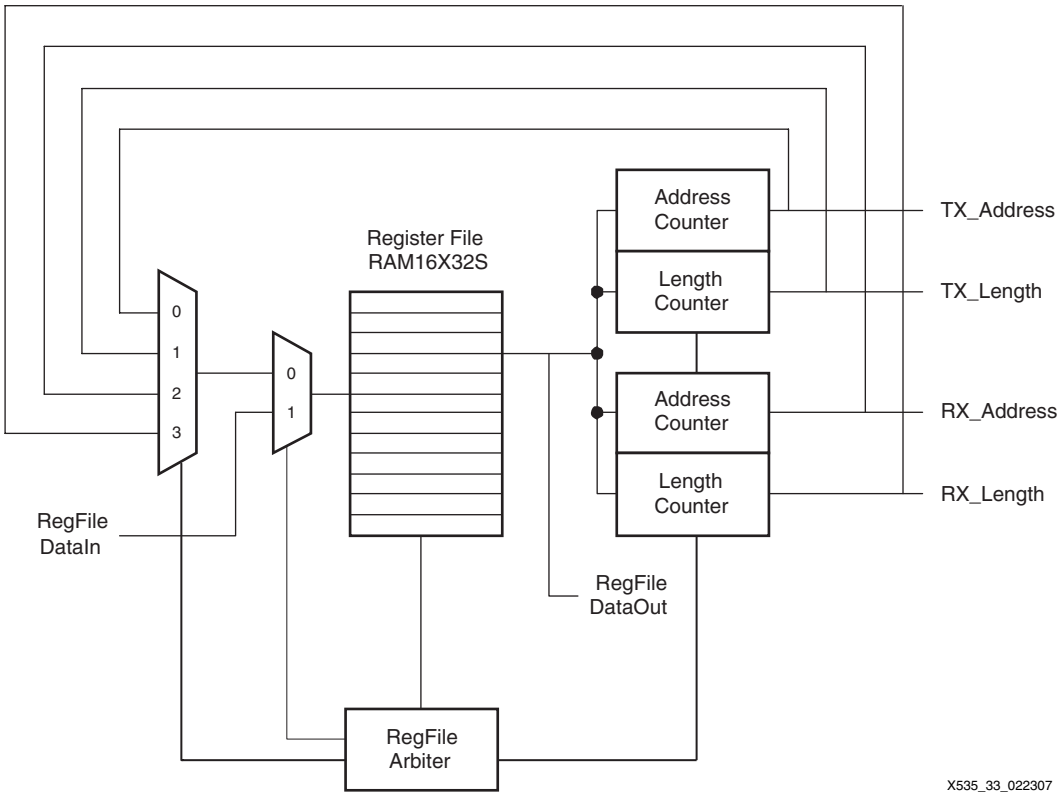


Figure A-32: **CDMAC Resource Sharing**

The CDMAC logic resolves the issue of simultaneous access by temporally sharing the outputs of the LUT RAM as needed. This arrangement is particularly favorable for the CDMAC because actual usage of the register file is predictable.

To accomplish this temporary resource sharing, 0a register file arbiter (see Figure A-26, page 135) allows the CDMAC to determine which DMA engine gains access to the reg file, and manages the contents of the two sets of address and length counters. Tx engine uses one set of counters and the Rx engine uses the other counter set.

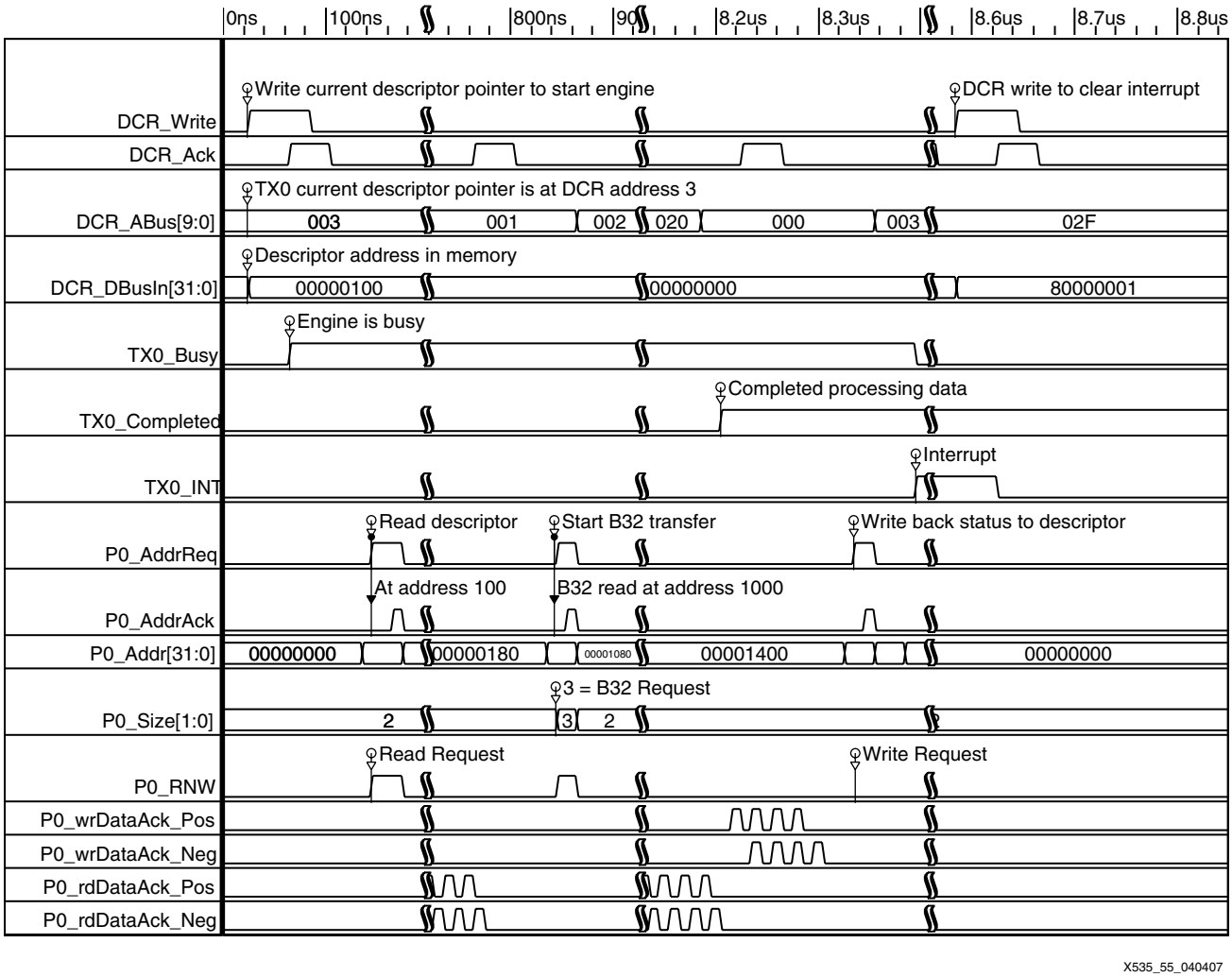
Note: The CDMAC status registers, along with the CDMAC Interrupt register, are implemented as regular flip-flop based registers. There is no way to use LUT RAM for these because their bitwise contents are dynamically changeable, and must be made simultaneously readable across the DCR bus at any time.

Timing Diagrams

The timing diagrams in this section illustrate essential CDMAC functionality. Together these timing diagrams demonstrate DCR writes, Port read and Writes for bursts and cache-lines, and Tx and Rx byteshifter operation. The first timing diagram shows a DMA Process. The following two timing diagrams break the process down into individual DMA transfers. The first two diagrams show Tx and Rx Byteshifting. The final diagram shows the way that descriptors are written back in the case of a two-descriptor chain.

CDMAC TX0 DMA Process Timing Diagram

Figure A-33 is an example of a TX0 DMA process.



X535_55_040407

Figure A-33: **CDMAC TX DMA Process Timing Diagram**

The CPU issues a `DCR_Write` to address 0x003, which is the TX0 current descriptor pointer register. This starts the CDMAC's TX0 engine and sets the `TX0_Busy` bit. The CDMAC then reads the descriptor from memory using an 8-word cache-line read (`CL8R`) request on the port interface.

Once the descriptor is read, the CDMAC reads the data to be transmitted on the LocalLink interface by issuing 32-word burst read (`B32R`) requests on the port interface.

After all of the data has been read, the CDMAC sets the `TX0_Completed` bit and writes the status back to the descriptor using an 8-word, cache-line write (`CL8W`) request.

Once the status has been written back to memory, if the status contains an asserted `Interrupt_On_End` bit, the CDMAC generates an interrupt to the CPU.

The CPU then clears the interrupt by issuing a `DCR_Write` to address 0x02F.

The `P0_wrDataAck` signals are asserted before the `P0_AddrReq` is asserted. This pushes the data into the MPMC2 write FIFOs and allows the MPMC2 to have more efficient arbitration.

TX0 Transfer Timing Diagram

Figure A-34 is an example of a TX0 transfer.

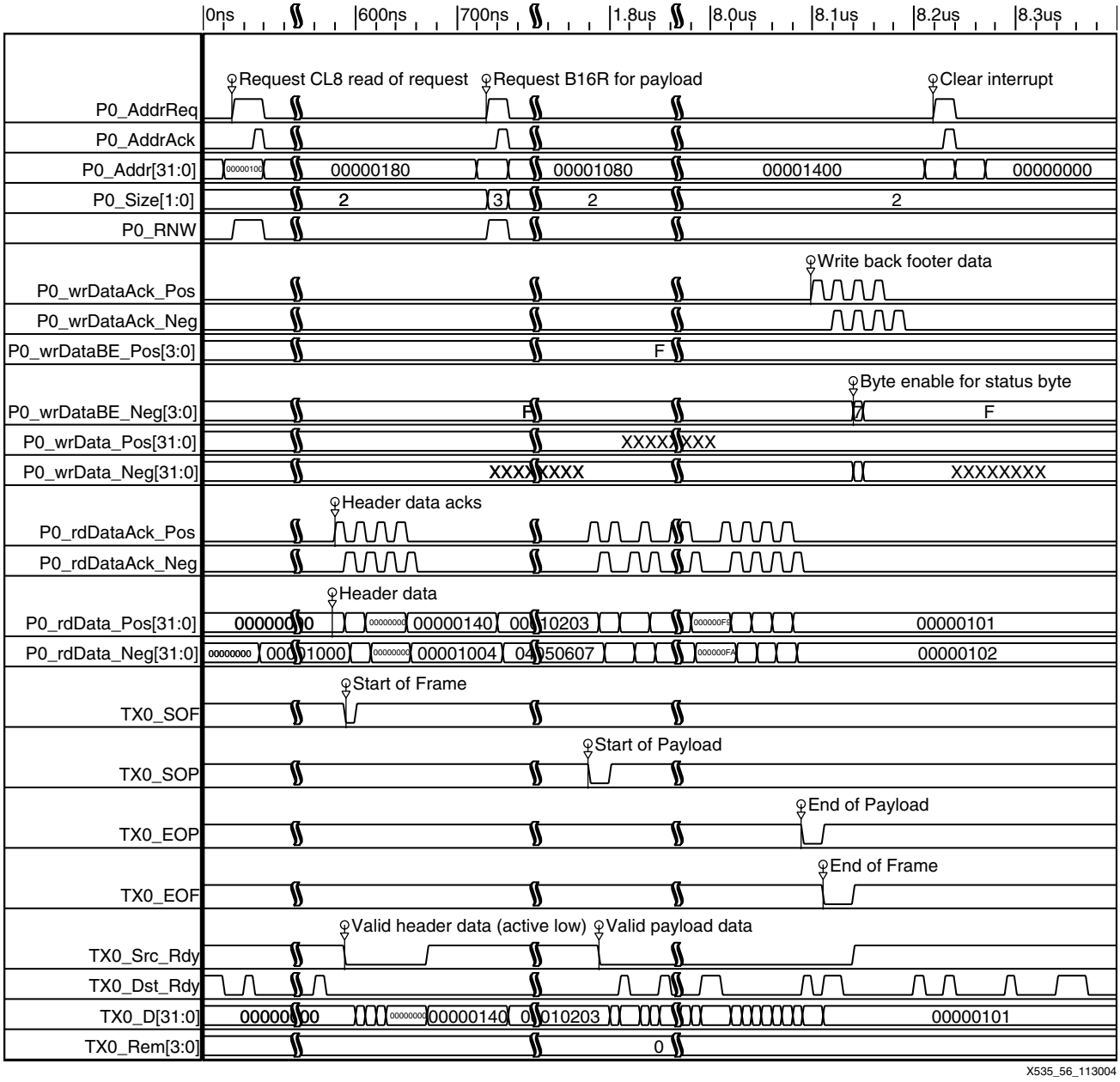


Figure A-34: **TX0 Transfer Timing Diagram**

Per the example in Figure A-34, page 146:

- The P0_wrDataAck signals are asserted before the P0_AddrReq is asserted. This pushes the data into the MPMC2 write FIFOs and allows the MPMC2 to have more efficient arbitration.
- CDMAC issues an 8-word, cache-line read (CL8R) request to the port interface.
- The descriptor data is passed through to the LocalLink interface as header data because this is the first descriptor of a process or the first descriptor following a descriptor with the EOP bit set.
- After the descriptor has been processed, the CDMAC begins issuing 32-word burst read (B32R) requests.
- The data is passed to the LocalLink interface as payload data.
- The CDMAC continues to issue B32Rs until the buffer length register reaches zero.
- The EOP bit is set in the STATUS register, so the CDMAC asserts the TX0_EOP and the TX0_EOF signal.
- Then the CDMAC issues an 8-word, cache-line write (CL8W) request to the port interface to write back the STATUS register.

RX0 Transfer Timing Diagram

Figure A-35 is an example of a RX0 Transfer.

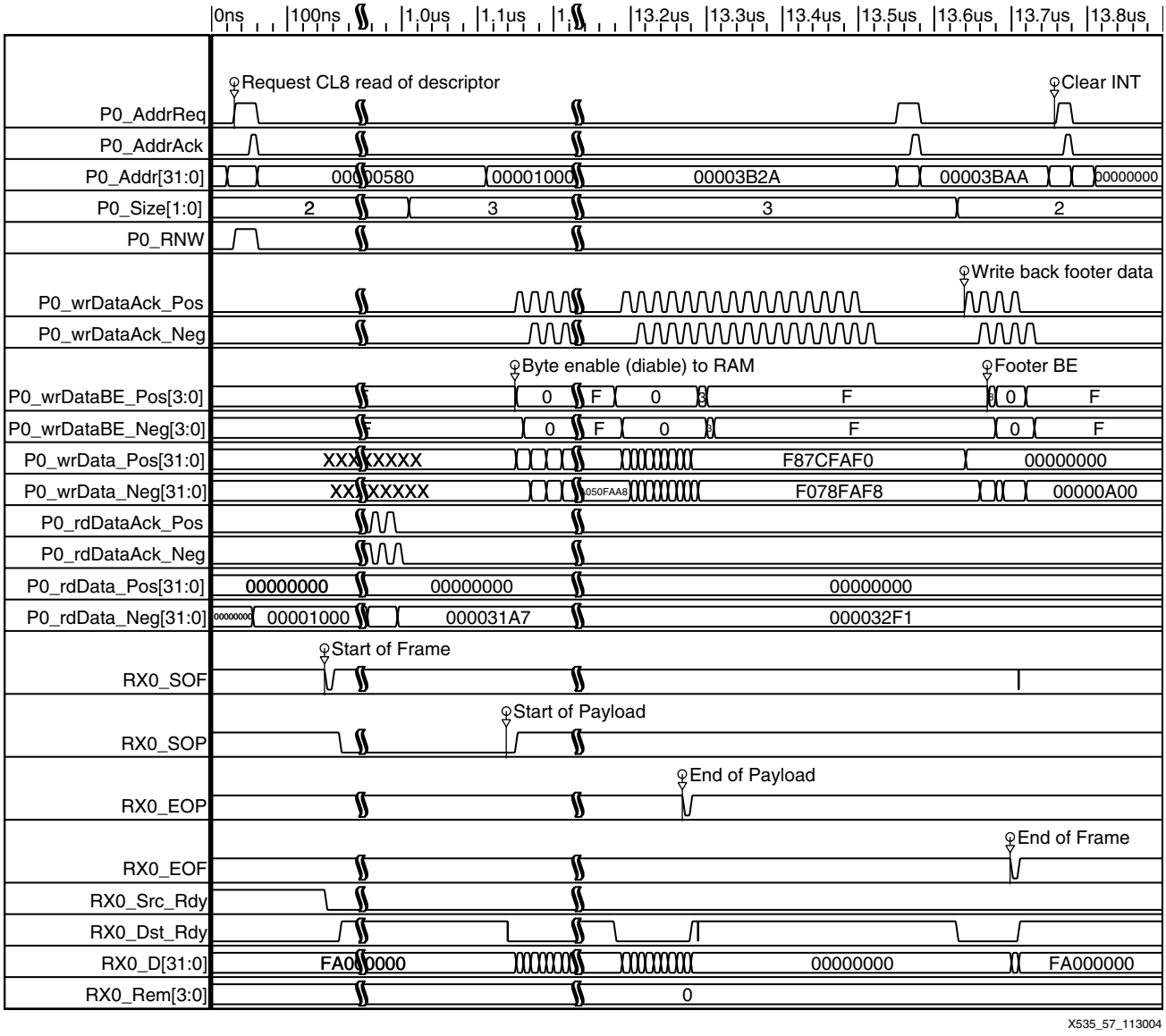


Figure A-35: **RX Transfer Timing Diagram**

Per the example in Figure A-35:

The CDMAC issues an 8-word cache-line read (CL8R) request to the port interface.

After the descriptor has been read and the RX LocalLink interface is in the payload state, the CDMAC instructs the RX LocalLink interface to begin collecting payload data from the RX LocalLink interface and writing it to memory.

- If the RX0_SOP signal is asserted, the Start Of Packet bit is set in the `STATUS` register.
- If the RX0_EOP signal is asserted, the End Of Packet bit is set in the `STATUS` register.

To process the payload data, the CDMAC issues 32-word burst write (B32W) requests until all payload data has been written to memory, or until the buffer length register reaches zero.

In Figure A-35, page 148, all of the payload data has been received, as indicated by RX0_EOP.

Because there is no more payload data to process, the CDMAC instructs the RX LocalLink interface to collect footer data and to write the status and the application data back to memory using an 8-word cache-line write (CL8W) request to the port interface.

The P0_wrDataAck signals are asserted before the P0_AddrReq is asserted. This pushes the data into the MPMC2 write FIFOs and allows the MPMC2 to have arbitration that is more efficient.

TX0 Byteshifter Timing Diagram

Figure A-36, page 150 is an example of the TX0 byteshifter. A descriptor has already been read by the CDMAC. The buffer address register was set to 0x1079. This sets the 3-bit Start_Offset signal to 0x1. This diagrams shows the first and second 32-word burst read (B32R) transactions.

- The first 120 bytes are ignored on P0_rdData_Pos and P0_rdData_Neg.
- The cycle that the 122nd byte is valid on P0_rdData_Pos, the last three bytes of P0_rdData_Pos are placed in the last three bytes of Port_RdData, as indicated by Set_Px_rdData_Pos.
- All four bytes of Port_RdData are clock-enabled into Port_TX_Out by asserting all four bytes of Byte_Reg_CE.
- The Byte_Sel signals move the Port_RdData bytes into the correct location.
- On the cycle where the 125th byte is valid, the first byte of P0_rdDataNeg is placed in the first byte of Port_RdData.

Again, all four bytes are clock-enabled into Port_TX_Out by asserting Byte_Reg_CE. Port_TX_Out now contains the last three bytes of P0_rdData_Pos and the first byte of P0_rdData_Neg in the correct order: 0x01020304.

Port_TX_Out is passed on to the LocalLink interface by asserting the TX_Src_Rdy signal. The leftover three bytes from P0_rdDataNeg are stored by clock-enabling them into Port_TX_Out and deasserting the last three bytes of Byte_Reg_CE. These bits are deasserted until the P0_rdDataAck_Pos is asserted for second B32R.

The three leftover bytes from the first B32R and the first byte from the second B32R are passed on to the LocalLink interface by asserting the TX_Src_Rdy signal. The Byte_Reg_CE begins clock enabling all four bytes of Port_RdData as it becomes available. This data is passed on to the LocalLink interface.

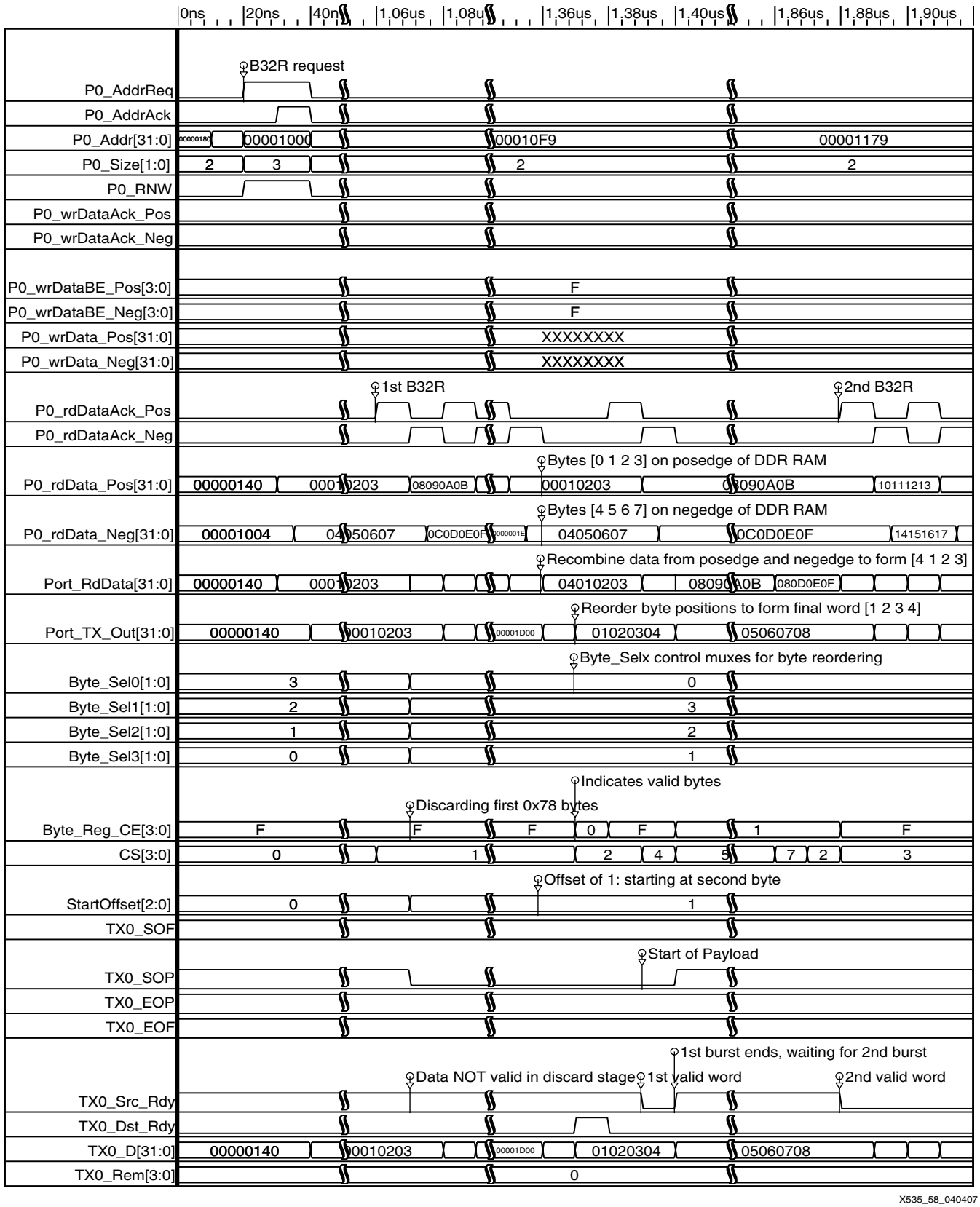


Figure A-36: **TX0 Byteshifter Timing Diagram**

RX0 Byteshifter Timing Diagram

Figure A-37 is an example of the RX0 byteshifter.

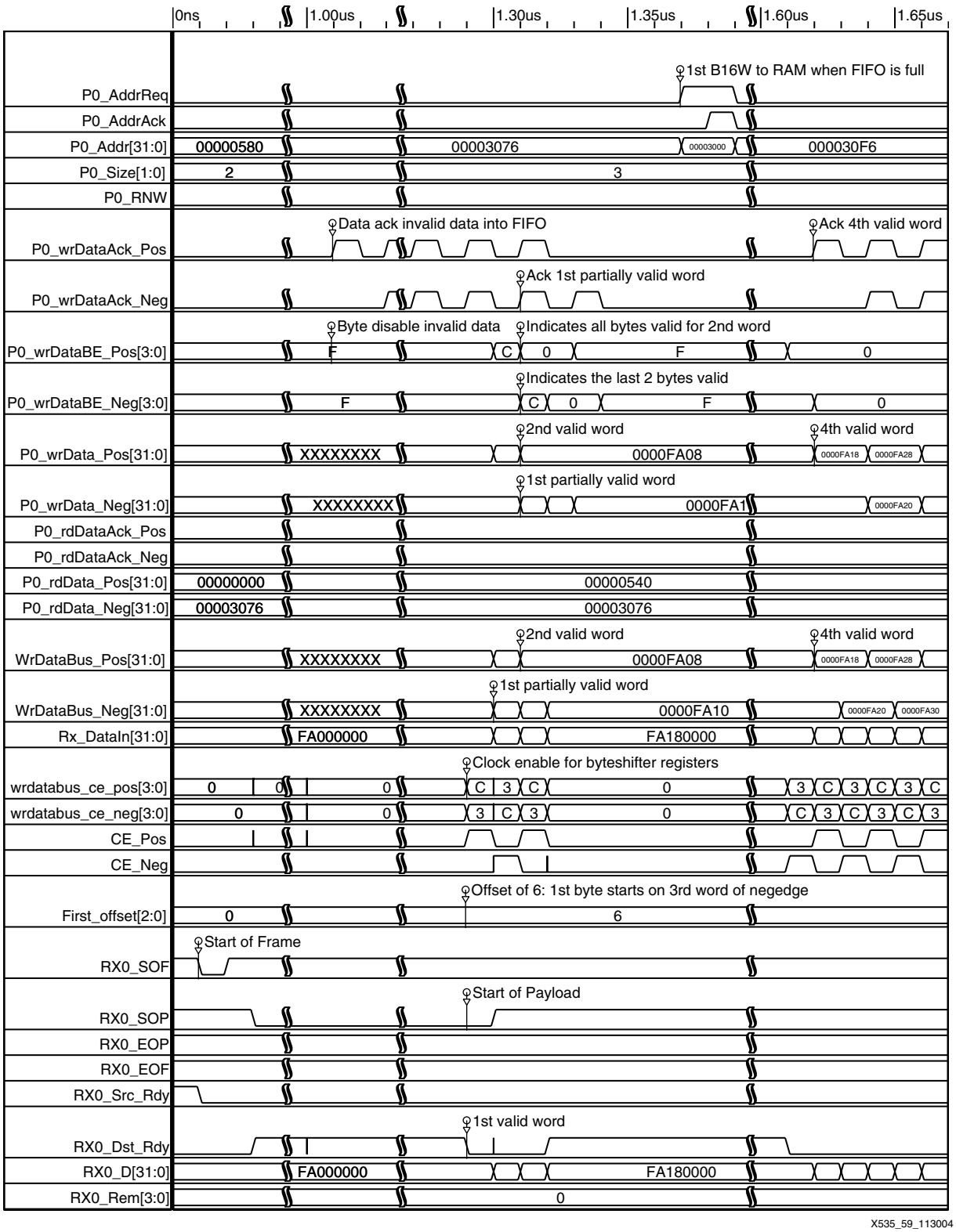


Figure A-37: **RX0 Byteshifter Timing Diagram**

In the Figure A-37, page 151 example:

- A descriptor has already been read by the CDMAC. The buffer address is set to `0x3076`. This sets the three-bit `First_offset` signal to `0x6`. In this diagram, the first and part of the second 32-word burst write (`B32W`) transactions are shown.
- The CDMAC stuffs 112 bytes of data into the MPMC2 FIFOs by asserting `P0_WrDataAck_Pos` and `P0_WrDataAck_Neg` with the byte enables (`P0_wrDataBE_Pos` and `P0_wrDataBE_Neg`) deasserted.

  Note: Four bytes of LocalLink payload data is collected and shifted by six bytes by asserting `CE_Pos`. Because the offset is by six bytes, `P0_wrDataAck_Pos` is asserted with the byte enable signals deasserted.
- `P0_wrDataAck_Neg` is asserted with the last two-byte enable signals asserted. From this point on, all byte enables are asserted until the LocalLink interface indicates that there is no more payload data, as specified by `RX_EOP`, or until the number of bytes specified by buffer length register have been written to memory.
- The LocalLink interface stalls between `B32W` requests by deasserting `RX_Dst_Rdy`.
- When the CDMAC instructs the byteshifter to execute a `B32W`, the data is pushed into the MPMC2 write FIFOs before the `P0_AddrReq` is asserted.

RX0 Descriptor Write Back for a Two-Descriptor Chain Timing Diagram

Figure A-38 is an example of how RX0 Descriptor Write Back works for a two-descriptor chain.

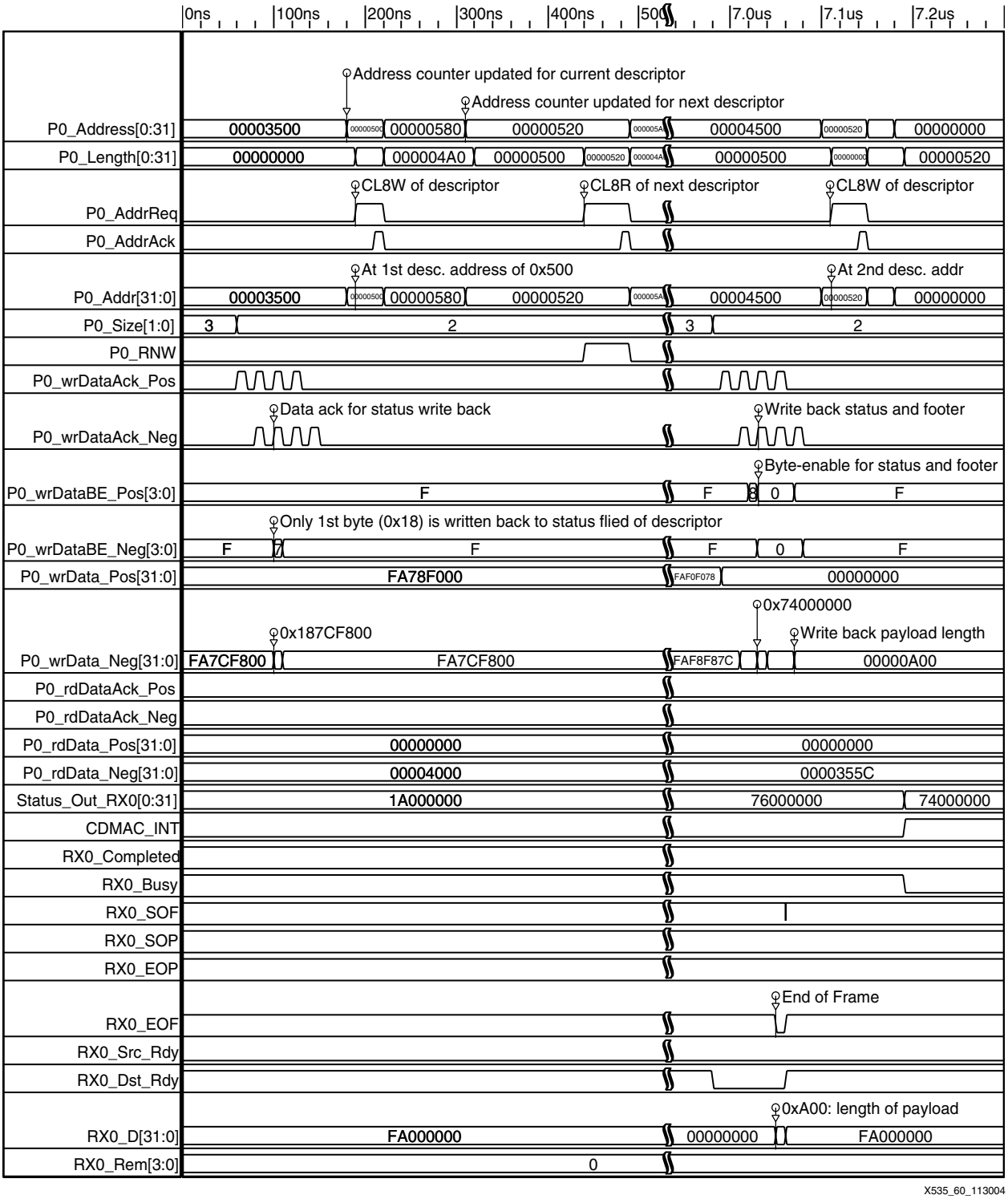


Figure A-38: **RX Descriptor Write Back for a 2-Descriptor Chain**

In the Figure A-38, page 153 example:

The first descriptor was set up with the buffer address register set to `0x3000` and the buffer length register set to `0x500`.

- The `P0_Address` bus contains the address in memory that the CDMAC accesses next.
- The `P0_Length` bus contains the number of bytes left to read or write.

At the top of the diagram, `P0_Length` has decremented until it reached zero bytes. Because there is still more LocalLink payload data to write to memory and the `Stop_On_End` bit is not set in the `STATUS` register, the status is written back to memory issuing an 8-word, cache-line write (`CL8W`). The only byte-enables that are asserted are for the status.

The `Start_Of_Payload` bit is asserted in the `STATUS` register because the LocalLink interface asserted `RX_SOP` while processing this descriptor. After this descriptor is written back to memory, the `P0_Address` is updated for the next descriptor.

The CDMAC then reads the descriptor from this location in memory and processes the descriptor in the normal fashion.

The LocalLink payload length is `0xA00` bytes, of which 500 bytes were processed by the first descriptor. The second descriptor has the buffer address register set to `0x4000` and the length set to `0xA00`. This means that 0x500 bytes are processed by the second descriptor before the LocalLink interface issues the `RX_EOP` signal. This sets the `End_Of_Packet` bit in the `STATUS` register.

The CDMAC then stops the transfer, collects the footer data from the LocalLink interface, and uses this to write the descriptor back to memory.

The byte enables for the `STATUS` register and the `APPLICATION-DEFINED` data is asserted.

As the `Interrupt_On_End` bit is set, an interrupt is generated and sent to the CPU.

CDMAC Software Model

The following subsections detail the CDMAC software model, including the overall programming model and the register and bitmap definitions.

See the "DMA Descriptor Model," page 103 for a detailed description of the way DMA decriptors are used.

CDMAC Programming Model

The CDMAC is designed in a modular fashion to bolt between the MPMC2 and the LocalLink devices. Figure A-39 illustrates the high-level architecture of the CDMAC.

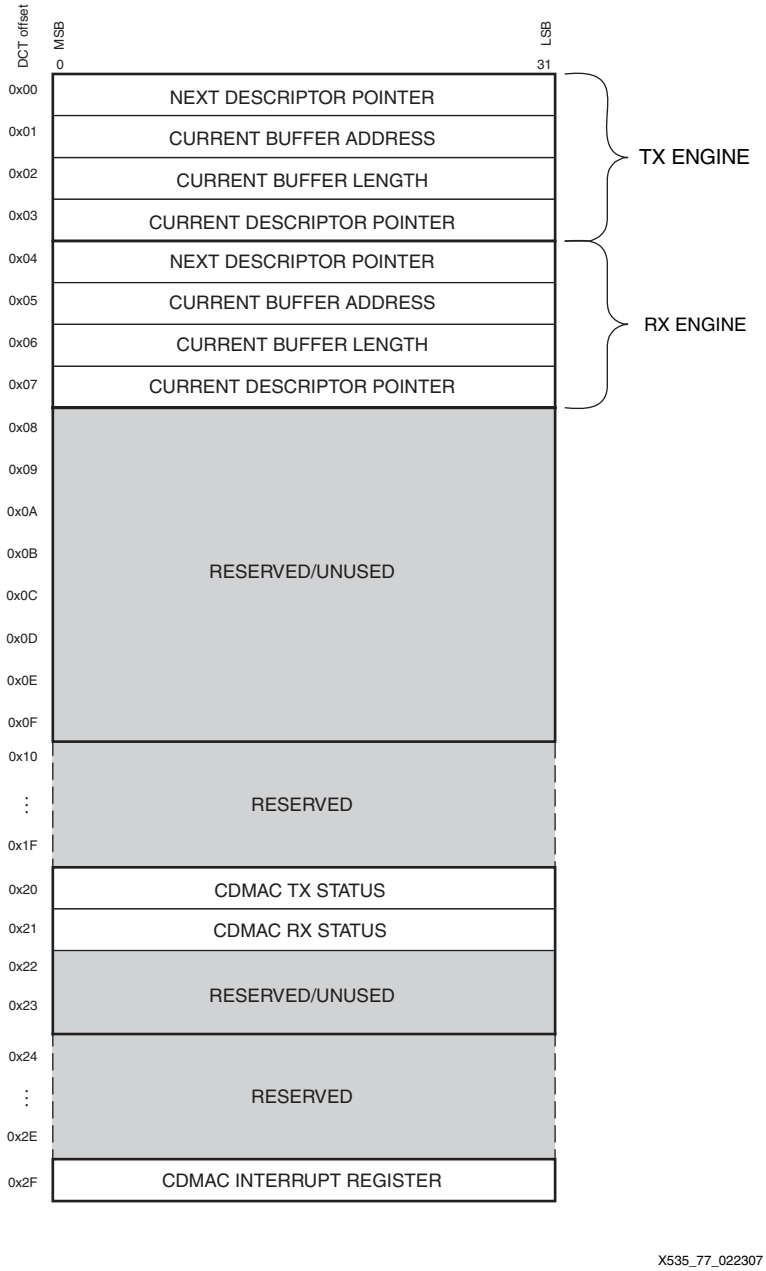


Figure A-39: **CDMAC Programmer Model**

CDMAC Register Definitions

The following pages detail the bitmaps of each register. There are four total DMA engines, and each has the same register set with the exception of the Interrupt Register that all four DMA engines share.

CDMAC Next Descriptor Pointer Register

Table A-3: **Next Descriptor Pointer Register (DCR_Base + 0x00, 0x04, 0x08, 0x0C)**

MSB																													LSB
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ADDRESS																															

Table A-4: **CDMAC Next Descriptor Pointer Register Bitmap**

Bit	Description
0:31	Address: 8 word aligned pointer to the next descriptor in the chain. If Null (0x0), DMA engine stops processing descriptors

The Next Descriptor Pointer register is loaded from the value contained in the `NEXT_DESCRIPTOR_POINTER` field in the currently pointed to descriptor. This value is kept in the respective CDMAC register until the CDMAC has completed all DMA transactions within the DMA transfer (reference Figure A-3, page 100).

After all DMA transactions are complete, the current descriptor is complete, and the `CDMAC_COMPLETED` bit is set in the `STATUS` register.

The current descriptor is written to update the `STATUS` field within the descriptor.

The CDMAC evaluates the address contained in the Next Descriptor Pointer register and performs the following:

- If the register contains a `NULL` (0x00000000), the CDMAC engine stops processing descriptors.
- If the address in the register is not 8-word aligned, or reaches beyond the range of available memory, the CDMAC halts processing and sets the `CDMAC_ERROR` bit in the `STATUS` register.
- If the register contains a valid address, the contents are moved to the Current Descriptor Pointer register. This movement causes the CDMAC to begin another DMA transaction.

CDMAC Current Address Register

Table A-5: **Current Address Register (DCR_Base + 0x01, 0x05, 0x09, 0x0D)**

MSB																													LSB
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ADDRESS																															

The Current Address register:

- Maintains the contents of the address in memory where the DMA operation is conducted next. This value is originally loaded into the CDMAC when the descriptor is read by the CDMAC.
- Once set by the current descriptor, the CDMAC then periodically transfers this value to an internal address counter that then updates the value for each DMA transaction completed.
- Upon termination of the transaction, the CDMAC overwrites the value of the Current Address register with the last value of the address counter. This process continues repeatedly until the CDMAC has completed the current descriptor.

This mechanism allows CDMAC to maintain multiple temporal channels of DMA at a substantially reduced hardware cost. Software can find this mechanism useful for identifying where the CDMAC is in a DMA operation; however, Xilinx does not recommend that software use the Current Address register for this purpose because it changes dynamically.

CDMAC Current Length Register

Table A-6: **Current Length Register (DCR_Base + 0x02, 0x06, 0x0A, 0x0E)**

MSB																													LSB
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED								24-BIT LENGTH																							

The Current Length register:

- Maintains the contents of the remaining length of the data to be transferred by the CDMAC. The value is originally loaded into the CDMAC when the descriptor is read by the CDMAC.
- Once set by the current descriptor, the CDMAC then occasionally transfers this value to an internal length counter, which then updates the value for each DMA transaction completed.
- Upon termination of the transaction, the DMAC overwrites the value of the Current Length register with the last value of the internal length counter. This process continues repeatedly until the CDMAC has completed the current descriptor.

This mechanism allows CDMAC to maintain multiple temporal channels of DMA at a substantially reduced hardware cost. Software can find this mechanism useful for identifying where the CDMAC is in a DMA operation; however, Xilinx does not recommend that software use this mechanism for that purpose because it changes dynamically.

CDMAC Current Descriptor Pointer Register

Table A-7: **Current Descriptor Pointer Register (DCR_Base + 0x03, 0x07, 0x0B, 0x0F)**

MSB LSB

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ADDRESS																															

The Current Descriptor Pointer register maintains the pointer to the descriptor that is currently being processed. The value was set either by the CPU when it first initiated a DMA operation, or is copied from the Next Descriptor Pointer register upon completion of the prior descriptor. This value is maintained by the CDMAC as a pointer so that the CDMAC can update the descriptor `STATUS` and `APPLICATION_DEPENDENT` fields once the descriptor has been fully processed.

CDMAC Status Register

Table A-8: **CDMAC Status Registers (DCR_Base + 0x20, 0x21, 0x22, 0x23)**

MSB LSB

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CDMAC_ERROR	CDMAC_INT_ON_END	CDMAC_STOP_ON_END	CDMAC_COMPLETED	CDMAC_START_OF_PACKET	CDMAC_END_OF_PACKET	CDMAC_ENGINE_BUSY	RESERVED																										

Table A-9: **CDMAC Status Register Bitmap**

Bit	Description
0	**CDMAC_ERROR**: When set (for example = 1) indicates the CDMAC encountered an error. The CDMAC sets this bit in the STATUS field of the descriptor. This bit, when set, indicates one of the following possible errors, which result from software issues: • Completed Error - CDMAC encountered a descriptor where the completed bit was already set prior to use • Buffer Address Error - CDMAC buffer address was not in range of memory • Next Descriptor Pointer Error - CDMAC found that the Next Descriptor Pointer was not 8-byte aligned • Current Descriptor Pointer Error - CDMAC found that the Current Descriptor Pointer was not 8-byte aligned • Busy Write Error - The CPU attempted to write to the Current Descriptor Pointer register when the CDMAC was busy In all cases, an error results in halting the CDMAC channel and setting the interrupt for that channel. If the CDMAC Interrupt register has the Master Interrupt Enable bit set, the CPU receives an interrupt when an error is encountered.
1	**CDMAC_INT_ON_END**: When set (for example = 1) forces the CDMAC to interrupt the CPU when the descriptor is completed. The CPU sets this bit in the STATUS field of the descriptor. The bit is then read into the CDMAC_STATUS_REGISTER as each descriptor is processed. If the bit is set in any given descriptor, the CDMAC generates an interrupt to the CPU. **Note:** CDMAC_STOP_ON_END and CDMAC_INT_ON_END are independent of each other. As such the CDMAC can be made to do any of four possible operations: • Halt upon completion of current descriptor without interrupt • halt upon completion of current descriptor with interrupt • Interrupt upon completion of current descriptor while beginning to process the next descriptor (if existing) • Begin to process the next descriptor (if existing)
2	**CDMAC_STOP_ON_END**: When set (for example = 1) forces the CDMAC to halt operations when the descriptor is completed. The CPU sets this bit in the STATUS field of the descriptor. The bit is then read into the CDMAC_STATUS_REGISTER as each descriptor is processed. If the bit is set in any given descriptor, the CDMAC halts processing any further descriptors for that channel. It will *NOT* generate an interrupt to the CPU unless explicitly told to do so by the CDMAC_INT_ON_END bit being set in the same descriptor. **Note:** CDMAC_STOP_ON_END and CDMAC_INT_ON_END are independent of each other. As such, the CDMAC do any of the following possible operations: • Halt upon completion of current descriptor without interrupt • Halt upon completion of current descriptor with interrupt • Interrupt upon completion of the current descriptor while beginning to process the next descriptor (if there is one) • Begin to process the next descriptor (if existing)
3	**CDMAC_COMPLETED**: When set (for example = 1) indicates that the CDMAC has transferred all data defined by the current descriptor. The CDMAC sets this bit upon completing the transaction associated with the descriptor. In the case of a transmit operation from memory (for example, READs), the CDMAC transfers data until the length field specified in the descriptor is zero, and then set this bit. In the case of a receive operation to memory (for example, WRITEs), the CDMAC sets this bit upon transferring data in the descriptor until the length is zero, OR, when it receives an END_OF_PACKET indicated from the CDMAC interface. **Note:** Because the receive buffers are typically larger than the size of received data, the length field in the descriptor will *NOT* specify how much data was actually received. Please see the CDMAC interface specification for more details on how length is established. **Note:** For software to properly utilize the CDMAC_COMPLETED feature, it must clear the bit in the descriptor prior to initiating CDMAC transactions based upon this descriptor. The CDMAC only sets this bit. It resets only upon hardware reset.

Table A-9: **CDMAC Status Register Bitmap *(Continued)***

Bit	Description
4	**CDMAC_START_OF_PACKET**: When set (for example = 1) indicates that the current descriptor is the start of the packet. The CDMAC sets this bit upon completing the transaction associated with this descriptor. In the case of a transmit operation from memory (for example, READs). Therefore CDMAC_START_OF_PACKET is set by the CPU in the descriptor to indicate to the CDMAC that this is the first descriptor of the packet to be transmitted. CDMAC RX Engine performs receive operations to memory (for example, WRITEs). Therefore CDMAC_START_OF_PACKET is set by the CDMAC in the descriptor to indicate to the CPU that this is the first descriptor of the packet being received.
5	**CDMAC_END_OF_PACKET**: When set (for example = 1) indicates that the current descriptor is the final one of the packet. CDMAC TX Engine performs transmit operations from memory (for example, READs). Therefore CDMAC_END_OF_PACKET is set by the CPU in the descriptor to indicate to the CDMAC that this is the last descriptor of the packet being transmitted. CDMAC RX Engine performs receive operations to memory (for example, WRITEs). Therefore CDMAC_END_OF_PACKET is set by the CDMAC in the descriptor to indicate to the CPU that this is the last descriptor of the packet being received.
6	**CDMAC_CHANNEL_BUSY**: When set (for example = 1) indicates that the CDMAC channel is busy processing DMA operations. The CDMAC_CHANNEL_BUSY bits tell software when the CDMAC is busy with the appropriate channel. In general, software should not disturb the CDMAC when the busy bit is set. The software can read the STATUS registers when the CDMAC is busy, but not write to any of the channel registers while that channel is busy. The CDMAC Interrupt register can be read at any time while any or all channels are busy. **Note:** CDMAC_CHANNEL_BUSY does communicate where the CDMAC is in processing the current descriptor to software. That information can be obtained by reading the Current Length register for the appropriate channel.
7:31	**RESERVED**:

CDMAC Interrupt Register

Table A-1: **CDMAC Interrupt Register (DCR_Base + 0x2F)**

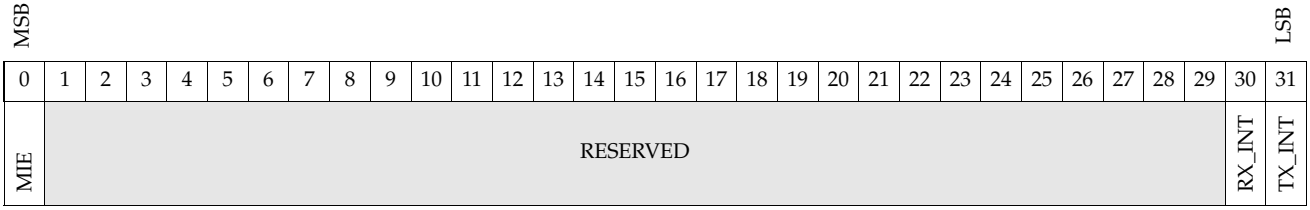


Table A-2: **CDMAC Interrupt Register Bitmap**

Bit	Description
0	**Master Interrupt Enable:** When set (for example = 1) indicates that the CDMAC is enabled to generate interrupts to the CPU. The Master Interrupt Enable (MIE) is used to enable and disable the CDMACs interrupt pin. • If the MIE is set to a 1, then the CDMAC generates a HIGH level (1) on the INT pin whenever any of the interrupt sources have a valid interrupt. • If the MIE is cleared to a 0, the CDMAC always leaves the INT pin set at a LOW level (0), regardless of any pending internal interrupts. **Note:** The MIE permits software to choose between polled mode of operation and interrupt driven.
1:29	**Reserved**:
30	**RX_INT**: When set (for example = 1) indicates that the CDMAC channel is requesting interrupt service from the CPU. This bit reflects the RX channel request for service. This channel independently requests access to the CPU. The 4-channel interrupt bits are OR'd together and then AND'd with the Master Interrupt Enable (MIE) to produce the INT pin. If the MIE is disabled, each of these bits can still be read from this register in polled mode. This bit is a set and reset Flip-flop. Acknowledging an interrupt on the RX channel is accomplished by writing a one to this bit. Note that this mechanism assures that interpose cannot be lost, and permits multiple channels to be simultaneously acknowledged.
31	**TX_INT**: When set (for example = 1) indicates that the CDMAC channel is requesting interrupt service from the CPU. This bit reflects the TX channel request for service. This channel independently requests access to the CPU. The 4-channel interrupt bits are OR'd together and then AND'd with the Master Interrupt Enable (MIE) to produce the INT pin. If the MIE is disabled, each of these bits can still be read from this register in polled mode. This bit is a set and reset Flip-flop. Acknowledging an interrupt on the TX channel is accomplished by writing a one to this bit. Note that this mechanism assures that interpose cannot be lost, and permits multiple channels to be simultaneously acknowledged.

Using the CDMAC in a System

The CDMAC is normally instantiated along with the MPMC2. The reference systems provided in this appendix show how it is connected and used. By examining the contents of the provided hardware source files, simulation, and test software, you can better understand the functionality of the CDMAC.

There are many methods of use for the CDMAC. Each method depends on to what the CDMAC is to be connected, and the data rate requirements.

The provided reference systems show a typical example of a video application wherein the CDMAC is connected to a set of video devices that are streaming in data.

The DMA engines contained in the CDMAC are independent of one another. This allows the software that is manipulating the DMA descriptors to not have to communicate with other channels. This is a very important facility for device driver development. The features currently provided in the CDMAC are designed to help further offload the CPUs required load to manage DMA traffic. The preferred methods of operation (as the CDMAC is currently implemented) are best observed when analyzing the stand-alone software applications that are documented in the *Multi-Port Memory Controller 2 (MPMC2) Application Note*. The "Additional Resources," page 13 contains a link to this document.

Appendix B

Error Correcting Code (ECC)

Overview

This appendix describes the ECC implementation in MPMC2. The ECC is optionally included via parameter control. The ECC features are:

- Supports 8, 16, 32, and 64-bit wide DDR and DDR2 memories
- Provides Single Error Correction (SEC) and Double Error Detect (DED)
- Can generate interrupts when the number and type of errors reach a programmed threshold value

Implementation

ECC functionality is implemented by inserting the ECC decode and encode logic between the Physical Interface (PHY) and Data Path of the MPMC2. Figure B-A-1 shows the current MPMC2 PHY Data Path connection.

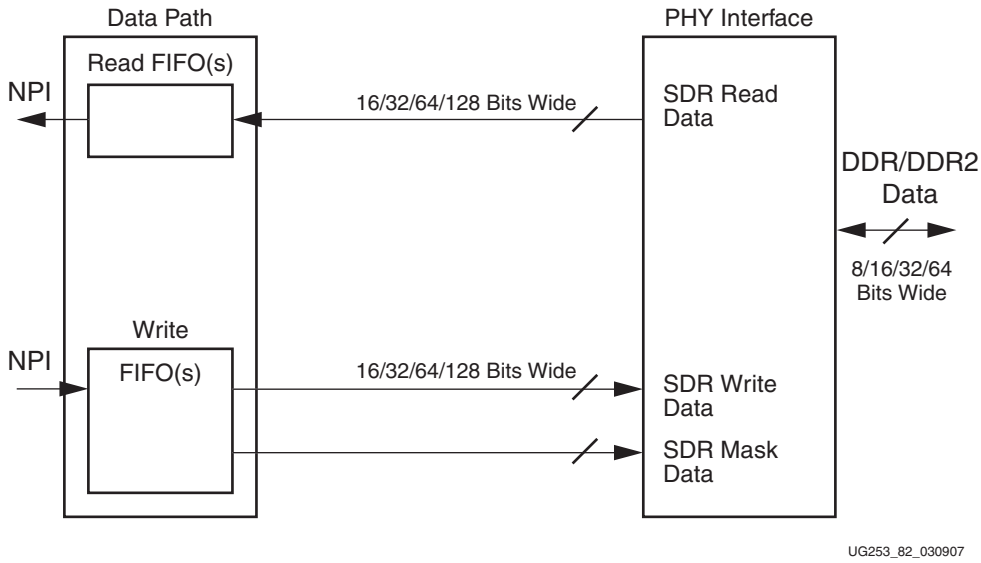


Figure A-1: **MPMC PHY-Data Path Connection**

The PHY interface converts the Double Data Rate (DDR) data from the memory into a Single Data Rate (SDR) bus that is twice as wide as the memory width.

Because MPMC2 supports 8, 16, 32, and 64-bit memory, this results in a 16, 32, 64, or 128-bit wide SDR bus going to the data path module. The data path module implements the per-port FIFOs used by MPMC2.

Figure B-A-2 illustrates the changes made when ECC is enabled.

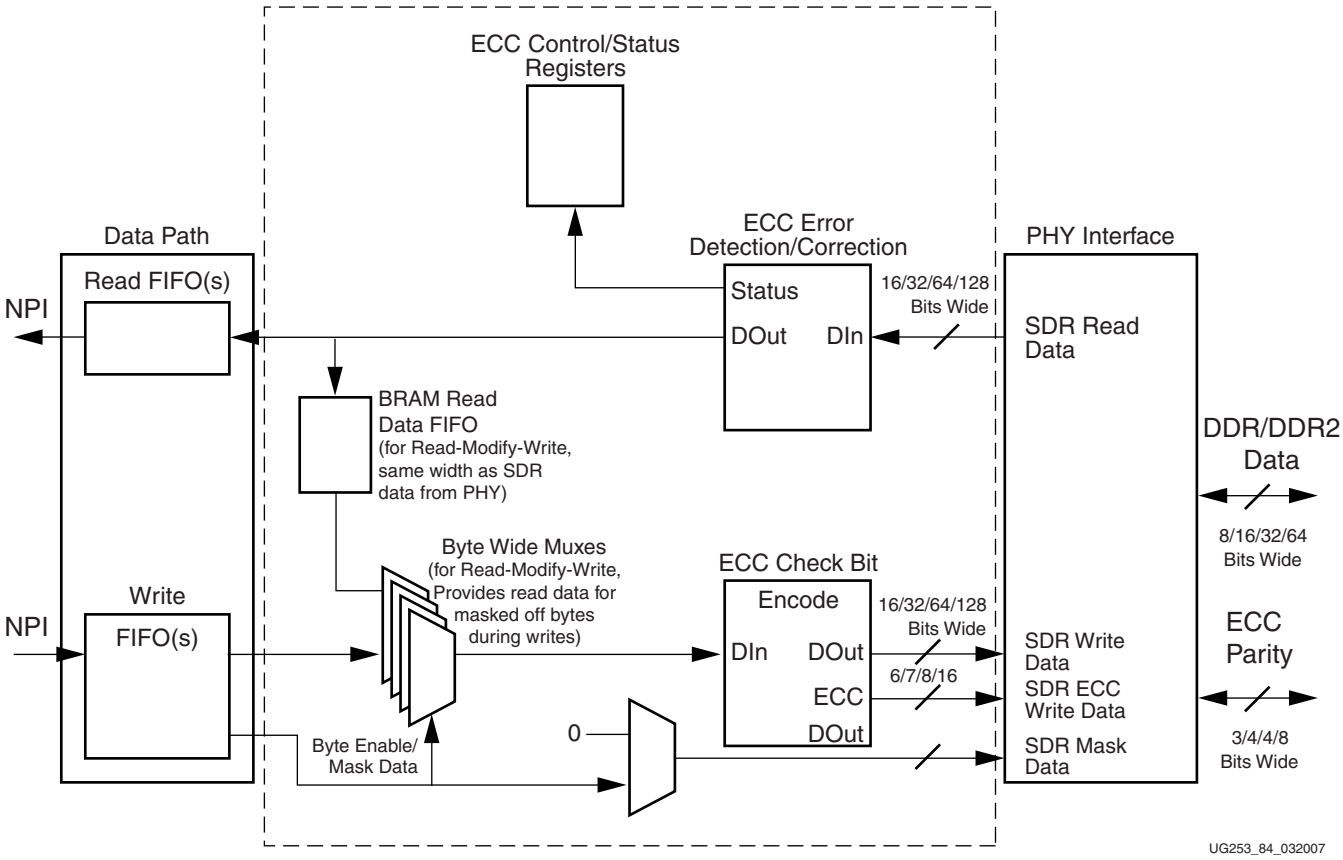


Figure A-2: **Data Path with ECC Enabled**

When ECC functionality is enabled, several blocks are turned on to implement control and status registers, ECC decode and encode, and support for read-modify-write operations (needed for handling byte enables). The blocks are inserted between the PHY and Data Path so only one instance of the ECC logic is needed even if there are multiple ports. The ECC decode and encode is performed in the SDR domain.

The ECC Control and Status registers module controls when interrupts are generated and provides control and status data access with respect to ECC.

Refer to "MPMC2 ECC Registers," page 167 for more information.

Read Data Handling

On reads, data passes through the ECC Error Detection and Correction block. The data is checked for errors. If there are no errors, data passes through as normal and goes into the read FIFO as a normal read. If a single error is detected, the error is automatically corrected (SEC) and sent to the Read FIFO to complete as a normal read. If 2 errors are detected (DED), the data is not corrected but passed through unchanged and the problem reported to the control and status registers.

Need for Read Modify Write

During writes to memory, a new set of ECC check bits must be written to memory along with write data. The ECC encoding process is handled by the ECC Check Bit Encode block. Read-Modify-Writes (RMW) are needed for write transactions where the byte enables that span the width of the ECC word on the SDR bus are not all on or off. Also, many ECC DIMMs do not have DM pins and therefore require RMW.

When byte enables do not exist or are not all on or off, the correct value of the ECC check bits are not known because not all the data across the ECC word is present.

The RMW operation first fetches the data value of the masked off byte lanes so the correct ECC check bits for the whole ECC word can be computed and written.

RMW is accomplished by taking read data (after ECC decode and correction) and storing the data in a FIFO that is the same width as the SDR data bus. Later, as write data comes out of the Write FIFO, the masked off byte enables activate a MUX that routes read data to fill in the corresponding “holes” in the write data. The result is a complete set of data with which to compute the new ECC check bits. All resultant data is written to memory. Therefore writes with any byte enables turned off result in writes with all byte enables turned on. This has the side effect that masked off write data is read, scrubbed, and written back.

The control state machine is modified during a RMW operation to perform a full read and then a full write. Because the data is on the same memory page, an optimization is to skip the precharge and activate commands between read and write. To optimize the space in the control path BRAM state machines, RMW operations will reuse the corresponding read and write state machines for the give transfer size. Special BRAM state machine bits are used to implement the chained RMW operation and to skip precharge and activate commands

The RMW process does add latency to the design because a full read must precede a write. This can double the transaction time for writes. Writes with all byte enables on or all off are faster than writes with mixed byte enables due to need for RMW. A flag bit is introduced into the NPI interface to allow the transactions to be qualified as needing RMW or not needing RMW. This flag bit, which is called MPMC2_PIM_<*PortNum*>_RdModWr, tells the MPMC if transactions require RMW (=1) or do not require RMW (=0). The RdModWr signal must be asserted with AddrReq. RdModWr should be asserted with respect to the memory burst length of 4 and the ECC word size of the memory interface being used. For NPI interfaces that do not support the RMW or are not able to set it dynamically, the value should default to one to ensure correct write operations under all conditions. Dynamic RMW allows for write performance to be optimized when RMW is known to not be needed.

Note: If all the byte enables are OFF but a RMW operation is performed, data will still be read, scrubbed, and written back. This would permit a custom NPI device to perform burst writes with byte enables all off (and RdModWr = 1) to scrub memory. MPMC2 does not require the memory to support data mask (DM pins) with ECC. DM pins can be connected if needed or left unconnected.

ECC encode functionality adds latency to writes. RMW support adds one cycle of latency to multiplex in read data (in addition to latency of the full read transaction). The control state machines support the additional cycles of latency between Write FIFO pop and data appearing at the PHY Interface.

For debugging and testing, the ECC encode process allows you to insert single or double bit errors into the data being written for test purposes. The error insertion logic can be parameterized out.

Memory Organization and ECC Word Size

Table B-1 describes the ECC word size and number of extra memory data bits needed for different organizations of memory. The `RdModWr` flag must be set according to the alignment across the ECC word size over a memory burst length of 4.

Note: For DDR/DDR2, the ECC words are based on the SDR bus size.

Table B-1: **ECC Word Size**

Memory Type	Memory Data Width	Memory ECC Width (C_NUM_ECC_BITS)	ECC Word Size	Notes
DDR/DDR2	8	4	16 + 6 Check Bits	PHY calibration, - ECC data is only carried on three bits
DDR/DDR2	16	4	32 + 7 Check Bits	
DDR/DDR2	32	4	64 + 8 Check Bits	
DDR/DDR2	64	8	2 Instances of 64 + 8 Check Bits	Requires two parallel 64 bit ECC encode/decode blocks

MPMC2 ECC Registers

Table B-2 shows the ECC related registers which are included with the ECC logic for the MPMC2 when enabled (C_INCLUDE_ECC_SUPPORT = 1).

Table B-2: **MPMC2 ECC Register Descriptions**

Summary Grouping	DCR Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
ECC Core	C_ECC_BASEADDR + 0	ECCCR(1)	R/W	00000003(3)	ECC Control Register
	C_ECC_BASEADDR + 1	ECCSR(1)	R/ROW(2)	00000000	ECC Status Register
	C_ECC_BASEADDR + 2	ECCSEC(1)	R/ROW(2)	00000000	ECC Single Error Count Register
	C_ECC_BASEADDR + 3	ECCDEC(1)	R/ROW(2)	00000000	Double Error Count Register
	C_ECC_BASEADDR + 4	ECCPEC(1)	R/ROW(2)	00000000	ECC Parity Field Error Count Register
ECC ISC	C_ECC_BASEADDR + 5	ECCADDR(1)	RO	N/A	ECC Error Address Register
	C_ECC_BASEADDR + 7	DGIE(1)	R/W	00000000	Device Global Interrupt Enable Register
	C_ECC_BASEADDR + 8	IPISR(1)	R/TOW(4)	00000000	IP Interrupt Status Register
	C_ECC_BASEADDR + 9	IPIER(1)	R/W	00000000	IP Interrupt Enable Register

Notes:

1. Only used if C_INCLUDE_ECC_SUPPORT = 1
2. ROW = Reset On Write. A write operation will reset the register.
3. Reset condition of ECCCR depends on the value of parameter C_ECC_DEFAULT_ON.
4. TOW = Toggle On Write. Writing'1' to a bit position within the register causes the corresponding bit position in the register to toggle.

ECC Control Register (ECCCR)

The ECC Control register determines if ECC check bits are generated during memory write operation and checked during a memory read operation. The ECC Control Register also defines testing modes if enabled by the parameter C_INCLUDE_ECC_TEST.

Note: This register is available only when C_INCLUDE_ECC_SUPPORT = 1

Figure B-A-3 illustrates the ECCR control register.

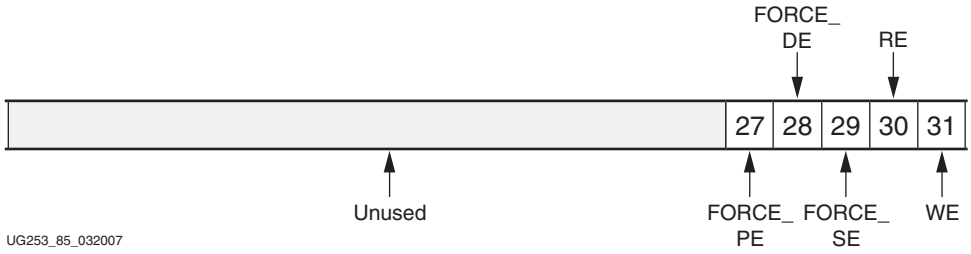


Figure A-3: **ECCR Control Register**

Table B-3 describes the bit values for the ECCCR Control register.

Table B-3: **ECCR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-26				**Reserved**
27	FORCE_PE	R/W	0	**Force Parity Field Bit Error**(1). Available for testing and determines if parity field bit errors are forced in the data stored in the memory. See ECC Testing for more information. 0 = No parity field bit errors are created 1 = Parity field bit errors are forced in stored data
28	FORCE_DE	R/W	0	**Force Double-bit Error**(1). Available for testing and determines if double-bit errors are forced in the data stored in the memory. 0 = No double-bit errors are created 1 = Double-bit errors are forced in the stored data
29	FORCE_SE	R/W	0	**Force Single-bit Error**(1). Available for testing and determines if single-bit errors are forced in the data stored in the memory. 0 = No single-bit errors are created 1 = Single-bit errors are forced in the stored data
30	RE	R/W	1(2)	**ECC Read Enable**. 0 = ECC read logic is bypassed 1 = ECC read logic is enabled
31	WE	R/W	1(2)	**ECC Write Enable**. 0 = ECC write logic is bypassed 1 = ECC write logic is enabled

Notes:

1. This bit is available only if C_INCLUDE_ECC_TEST = 1
2. Reset value is determined by parameter C_ECC_DEFAULT_ON.
   If C_ECC_DEFAULT_ON = 1 then this bit is equal to 1
   If C_ECC_DEFAULT_ON = 0, then this bit is equal to 0

ECC Status Register (ECCSR)

The ECC Status register in combination with the ECC Error Address Register (ECCADDR) records the first occurrence of an error and latches information about the error until the error is cleared by writing to the ECC Status Register (ECCSR).

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-4 describes the bit values for the ECC Status register.

Table B-4: **ECCSR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-15				**Reserved**
16-19	ECC_ERR_SIZE	R/ROW(1)	0000	**ECC Error Transaction Size**. Records the size of the NPI transaction where the error occurred
20	ECC_ERR_RNW	R/ROW(1)	0	**ECC ERROR Transaction Read/Write**. Indicates if error occurred on a read transactions or a write transaction that employed a read-modify-write operation
21-28	ECC_ERR_SYND	R/ROW(1)	00000000	**ECC Error Syndrome**. Indicates the ECC syndrome value of the most recent memory transaction in which a single-bit error was detected. The 8-bit syndrome value indicates the data bit position in which an error was detected and corrected
29	PE	R/ROW(1)	0	**Parity Field Bit Error.** During a memory transaction an error was detected in a parity field bit. 0 = No parity field bit errors detected 1 = Parity field bit error detected and corrected
30	DE	R/ROW(1)	0	**Double-Bit Error.** During a memory transaction a double-bit error was detected and is not correctable. 0 = No double-bit errors were detected 1 = Double-bit error was detected
31	SE	R/ROW(1)	0	**Single-Bit Error**. During memory transaction a single-bit error was detected and corrected. 0 = No single-bit errors were detected 1 = Single-bit error detected and corrected

Note:

1. ROW = Reset On Write. Any write operation to the ECCSR will reset the register.

ECC Single-Bit Error Count Register (ECCSEC)

The ECC Single-Bit Error Count register records the number of ECC single-bit errors that occurred during the memory transaction on the data bits only. ECC logic will correct the detected single-bit errors. When the value in this register reaches 4095 (the max count), the next single-bit error detected will not be counted. This count consumes 12-bits.

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`. Single bit errors occurring on ECC check bits are covered by the ECCDEC register.

Table B-5 describes the bit values for the ECC Single-Bit Error Count register.

Table B-5: **ECCSEC Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-19				**Reserved**
20-31	SEC	R/ROW(1)	0	**Single-Bit Error Count**. Indicates the number of single-bit errors that occurred during the pervious memory transactions. The maximum error count is 4095.

Note:

1. ROW = Reset On Write. Any write operation to the ECCSEC register will reset the register.

ECC Double-Bit Error Count Register (ECCDEC)

The ECC Double-Bit Error Count register records the number of ECC double-bit errors that occurred during the memory transaction. ECC cannot correct double-bit errors detected. When the value in this register reaches 4095 (the max count), the next double-bit error detected will not be counted.

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-6 describes the bit values for the ECC Double-Bit Error Count register.

Table B-6: **ECCDEC Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-19				**Reserved**
20-31	DEC	R/ROW(1)	0	**Double-Bit Error Count**. Indicates the number of double-bit errors that occurred during the pervious memory transactions. The maximum error count is 4095.

Note:

1. ROW = Reset On Write. Any write operation to the ECCDEC register will reset the register.

ECC Parity Field Bit Error Count Register (ECCPEC)

The ECC Parity Field Bit Error Count register records the number of bit errors that occurred in the ECC parity field during the memory transaction. ECC logic will correct detected parity field bit errors. When the value in this register reaches 4095 (the max count), the next parity field bit error detected will not be counted.

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-7 describes the bit values for the ECC Parity Bit Error Count register.

Table B-7: **ECCPEC Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-19				**Reserved**
20-31	PEC	R/ROW(1)	0	**Parity Field Bit Error Count**. Indicates the number of errors that occurred in the parity field bits during the last memory transactions. The maximum error count is 4095.

Note:

1. ROW = Reset On Write. Any write operation to the ECCPEC register will reset the register.

ECC Error Address Register (ECCADDR)

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-8 describes the bit values for the ECC Error Address register.

Table B-8: **ECCADDR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-31	ECCERRADDR	RO	N/A	**ECC Error Address.** Indicates the ECC Address corresponding to an ECC error reported by the ECC Status Register (ECCSR). This register value is valid when an error is actively reported in the ECCSR only.

ECC Interrupt Descriptions

Note: The interrupts described here are only available if `C_INCLUDE_ECC_SUPPORT = 1`

Device Global Interrupt Enable Register (DGIE)

The Device Global Interrupt Enable register is used to globally enable the final interrupt output from the ECC interrupt service.

Table B-9 describes the bit values for the Device Global Interrupt Enable register.

Table B-9: **DGIE Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0	GIE	R/W	0	**Global Interrupt Enable**. 0 = Interrupts disabled 1 = Interrupts enabled
1-31				Reserved

IP Interrupt Status Register (IPISR)

The IP Interrupt Status register is the interrupt capture register for the ECC logic.

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-10 describes the bit values for the ECC Interrupt Status register.

Table B-10: **IPISR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-28				**Reserved**
29	PE_IS	R/TOW(1)	0	**Parity Field Bit Error Interrupt Status**. Indicates a parity field bit error has occurred during the memory data transaction. In the ECC module, parity field bit errors will be corrected as data is read from memory. This interrupt is for system monitoring only and does not indicate corrupt data. 0 = Disabled 1 = Enabled
30	DE_IS	R/TOW(1)	0	**Double-Bit Error Interrupt Status**. Indicates a double-bit data error has occurred during the memory transaction. In the ECC module, double-bit errors can be detected, but not corrected. When this interrupt is asserted, the data read from memory is not valid. 0 = Disabled 1 = Enabled
31	SE_IS	R/TOW(1)	0	**Single-Bit Error Interrupt Status.** Indicates a single-bit error has been detected during the memory transaction. In the ECC module, single-bit errors will be detected and corrected. This interrupt is for system monitoring only and does not indicate corrupt data. 0 = Disabled 1 = Enabled

Note:

1. TOW is Toggle On Write.
Writing a 1 to a bit position within the register causes the corresponding bit position in the register to toggle.

IP Interrupt Enable Register (IPIER)

The IP Interrupt Enable register has an enable bit for each defined bit of the IP Interrupt Status register.

Note: This register is available only when `C_INCLUDE_ECC_SUPPORT = 1`

Table B-11 describes the bit values for the ECC IP Interrupt Enable register.

Table B-11: **IPISR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-28				**Reserved**
29	PE_IE	R/W	0	**Parity Field Bit Error Interrupt Enable**. Enables assertion of the interrupt for indicating parity field bit errors have occurred. 0 = Disabled 1 = Enabled
30	DE_IE	R/W	0	**Double-bit Error Interrupt Enable**. Enables assertion of the interrupt for indicating double-bit data errors have occurred.0 = Disabled 1 = Enabled
31	SE_IE	R/W	0	**Single-bit Error Interrupt Enable**. Enables assertion of the interrupt for indicating single-bit data errors have occurred. 0 = Disabled 1 = Enabled

ECC Testing

To enable testing on the ECC core logic, set `C_INCLUDE_ECC_TEST = 1`.

The ECC Control Register (ECCCR) described in "ECC Control Register (ECCCR)," page 168 includes control bits for enabling or disabling the test logic. The following list describes possible forcing errors combinations. If any other combination is attempted, no errors will be forced on the data written to memory.

- No bit error forcing
- Force single-bit data errors (ECC Control Register (ECCCR): `FORCE_SE = 1`)
- Force double-bit data errors (ECC Control Register (ECCCR): `FORCE_DE = 1`)
- Force parity bit errors (ECC Control Register (ECCCR): `FORCE_PE = 1`)
- Force single-bit data and single parity field bit errors (ECC Control Register (ECCCR): `FORCE_SE = 1` and `FORCE_PE = 1`)

For single-bit error testing, a mask shift register forces single-bit errors on the data written to memory. The data mask shift register has the least significant single-bit equal to one. When testing is enabled during a memory write, the shift register clocks the one towards to Most Significant Bit (MSB).

For double-bit error testing, a data mask shift register forces double-bit errors on the data written to memory. The data mask shift register has two adjacent bits equal to one (starting in the two least significant bit positions) and rotates the "11" pattern in the shift register (towards the two most significant bits) on each memory write.

When parity field bit error testing is enabled, a mask shift register forces single-bit errors on the check bits stored in memory.

The parity mask shift register has the least significant single-bit equal to one and rotates the one (toward the MSB) on each memory write.

XILINX®

Appendix C

MPMC2 Performance Monitoring

Overview

The MPMC2 Performance Monitor (PM) is an optional per-port feature that can be enabled or disabled in the configuration of MPMC2 using the PM checkboxes in the MPMC2 IP Configurator GUI. The PM provides the ability to instrument and measure the performance of the MPMC by collecting transactional statistics at the NPI level. Each PM captures the length of a transaction and stores the length value into a BRAM.

The Performance Monitor provides:

- 48-bit Dead Cycle counters and a 48-bit Global Cycle counter
- A DCR bus interface to read and write and enable the BRAMs on the PMs

Note: The MPMC2 release notes contain information about a PM reference design example which provides SW examples for accessing the PM registers.

Performance Monitor Registers

Table C-1 summarizes the registers that apply to the performance monitors.

Table C-1: **Performance Monitor Register Summary**

Grouping	DCR Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
PM_ADDRESS_REGISTER	C_PM_BASEADDR + 0	PMADDR(1,2)	R/W	00000000	Performance Monitor Address Register
PM_DATA_REGISTER	C_PM_BASEADDR + 1	PMDATA(1,2)	R/W	00000000	Performance Monitor Data Register
PM_CONTROL_REGISTER	C_PM_BASEADDR + 2	PMCTRL(1)	R/W	00000000	Performance Monitor Control Register
PM_GC_REGISTER	C_PM_BASEADDR + 3	PMGC(1,3)	R/W	00000000	Performance Monitor Global Counter Register

Notes:

1. Only available when a performance monitor is enabled.
2. Every read/write of PMDATA causes PMADDR to increment automatically.
3. Only valid when C_PM_DC_CNTR=1 or C_PM_GC_CNTR=1.

Performance Monitor Address Register (PMADDR)

PMADDR Bit Definitions

Table C-2 lists the PMADDR bit definitions.

Table C-2: **PMADDR Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-17				Reserved
18-31	ADDR	RW (1)	0	Address from which the PMDATA register will read/write. It auto increments for every read or write on the data_register. See PMDATA register description below for more information

Note:

1. Every read/write of PMDATA causes PMADDR to increment automatically.

Performance Monitor Data Register (PMDATA)

For each PM, there is one 512x36 BRAM and each 36-bit data value requires 2 DATA register reads and writes to access all 36 bits. The first read/write accesses the most significant bits (MSB), while the second read and write accesses the least significant bits (LSB). Therefore each PM has an address space to cover 512x36 bits.

In the address space following the PIMs, there are 8 48-bit Dead Cycle (DC) counters that can be accessed (if `C_PM_DC_CNTR = 0xff`). To save logic, the dead cycle counter can be excluded from the designs by setting the parameters `C_PM_DC_CNTR = 0x00` (in the `system.mhs` file).

Note: Dead Cycle counters for the Performance Monitors on ports on which there is no traffic might increment.

Note: This register is available only when a performance monitor is enabled.

PMDATA Bit Definitions

Table C-3 lists the PMDATA bit definitions.

Table C-3: **PMDATA Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-31	DATA	RW (1)	0	Count of transactions with a given type and length. Alternates between MSB and LSB of the 36 bit or 48 bit count values.

Note:

1. A read/write of PMDATA causes PMADDR to increment automatically

Performance Monitoring Register Mapping

Table C-4 shows the register mapping of PMADDR relative to PMDATA.

Table C-4: **PMADDR and PMDATA Mapping**

Register	PMADDR Range
PM0	PMADDR = 0x0000 - 0x03FF
PM1	PMADDR = 0x0400 - 0x07FF
PM2	PMADDR = 0x0800 - 0x0BFF
PM3	PMADDR = 0x0C00 - 0x0FFF
PM4	PMADDR = 0x1000 - 0x13FF
PM5	PMADDR = 0x1400 - 0x17FF
PM6	PMADDR = 0x1800 - 0x1BFF
PM7	PMADDR = 0x1C00 - 0x1FFF
DC0	PMADDR = 0x2000 - 0x2001
DC1	PMADDR = 0x2002 - 0x2003
DC2	PMADDR = 0x2004 - 0x2005
DC3	PMADDR = 0x2006 - 0x2007
DC4	PMADDR = 0x2008 - 0x2009
DC5	PMADDR = 0x200A - 0x200B
DC6	PMADDR = 0x200C - 0x200D
DC7	PMADDR = 0x200E - 0x200F

Qualifiers

Each PM is organized into qualifiers by writes and reads with 32 bins each. Each bin is 36 bits, MSB first for PM0. The following describes the register offsets definitions within the PMADDR range for a given PIM.

Qualifier 0

Write

bin 0 = Offset 0x00
bin 1 = Offset 0x02
...
bin 31 = Offset 0x3e

Read

bin 0 = Offset 0x40
bin 1 = Offset 0x42
...

Qualifier 1

Write

bin 0 = Offset 0x80
bin 0 = Offset 0x82
...
bin 31 = Offset 0BE

Read

bin 0 = Offset 0xC0
bin 1 = Offset 0xC2
...
bin 31 = Offset 0xFE
...

Qualifier 7

Write

bin 0 = Offset 0x380
bin 1 = Offset 0x382
...

Read

bin 0 = Offset 0x3C0
bin 1 = Offset 0x3C2
...
bin 31 = Offset 0x3FE

Qualifier Definitions

Table C-5 describes the qualifier definitions.

Table C-5: **Qualifier Definitions**

Qualifier	Description
0	Bytes - Double words
1	Cache Line 4
2	Cache Line 8
3	Unused
4	Burst 32
5	Burst 64
6	Unused
7	Unused

The value in the bins depends on the parameter C_PM_SHIFT_BY, which can be set in the mhs file (default value = 0). C_PM_SHIFT_BY shifts the counter for the transactions lengths to allow longer transactions to be recorded with the trade off of resolution.

With C_PM_SHIFT_BY = 0, Bin 0 = number of transactions of 0 length Bin 31 = number of transactions of 31+ length
With C_PM_SHIFT_BY = 1, Bin 0 = number of transactions of 0-1 length Bin 31 = number of transactions of 62-63+ length
With C_PM_SHIFT_BY = 2, Bin 0 = number of transactions of 0-3 length Bin 31 = number of transactions of 124-127+ length
With C_PM_SHIFT_BY = 3, Bin 0 = number of transactions of 0-7 length Bin 31 = number of transactions of 248-255+ length

Note: C_PM_SHIFT_BY has a max value of 3

Performance Monitor Control Register (PMCTRL)

The Performance Monitor Control register (PMCTRL) sets the enables for the BRAMs used by the PM and allows the enable values to be read. BRAMs should be disabled before data values are read to ensure data is not changing while being read. BRAMs can also be disabled to stop the count values from being updated. Enable BRAMs to record data on that PM. Enabling a PM BRAM also enables the Dead Cycle counter for that port. The LSB of PMCTRL enables PM0, while the next bit enables PM1, and so forth.

Note: This register is available only when a performance monitor is enabled.

PMCTRL Bit Definitions

Table C-6 describes the PMCTRL register bits.

Table C-6: **PMCTRL Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-23				Reserved
24	CTRL0	R/W	0	1=Enable PM0 0=Disable PM0
25	CTRL1	R/W	0	1=Enable PM1 0=Disable PM1
26	CTRL2	R/W	0	1=Enable PM2 0=Disable PM2
27	CTRL3	R/W	0	1=Enable PM3 0=Disable PM3
28	CTRL4	R/W	0	1=Enable PM4 0=Disable PM4
29	CTRL5	R/W	0	1=Enable PM5 0=Disable PM5
30	CTRL6	R/W	0	1=Enable PM6 0=Disable PM6
31	CTRL7	R/W	0	1=Enable PM7 0=Disable PM7

Note: For example, to enable all PMs except PM4 write 0xEF to PMCTRL

Performance Monitor Global Counter Register (PMGC)

To save logic, the global cycle counter can be excluded from the designs by setting the parameters C_PM_GC_CNTR = 0 (in the system.mhs file).

Note:
This register is available only when a performance monitor is enabled and C_PM_GC_CNTR = 0

PMGC Bit Definitions

Table C-7 describes the PMGC bit definitions.

Table C-7: **PMGC Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0-31	GC	RW (1)	0	**Global Cycle Counter Value**. Contains 48-bit value of the global cycle counter. It is enabled whenever any of the PMs are enabled in the PMCTRL. Because this register requires two reads/writes to fully access to counter value, it toggles between accessing MSB and LSB of the 48-bit value.