DesignWare DW_ahb_dmac Databook

DW_ahb_dmac – Product Code: 3889-0

2.22a
July 2018

Copyright Notice and Proprietary Information

© 2018 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at
https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Revision History . . . 9

Preface . . . 15
Organization . . . 15
Related Documentation . . . 16
Web Resources . . . 16
Customer Support . . . 16

Chapter 1
Product Overview . . . 19
1.1 DesignWare System Overview . . . 19
1.2 General Product Description . . . 21
1.2.1 DW_ahb_dmac Block Diagram . . . 21
1.3 Basic Definitions . . . 23
1.4 Features . . . 28
1.4.1 General . . . 28
1.4.2 Address Generation . . . 29
1.4.3 Channel Buffering . . . 29
1.4.4 Channel Control . . . 29
1.4.5 Transfer Initiation . . . 30
1.4.6 Flow Control . . . 30
1.4.7 Interrupts . . . 30
1.4.8 Low Power Mode . . . 30
1.5 Standards Compliance . . . 30
1.6 Verification Environment Overview . . . 30
1.7 Licenses . . . 31
1.8 Where To Go From Here . . . 31

Chapter 2
Functional Description . . . 33
2.1 Setup/Operation of DW_ahb_dmac Transfers . . . 33
2.2 Block Flow Controller and Transfer Type . . . 33
2.3 Handshaking Interface . . . 34
2.4 Basic Interface Definitions . . . 35
2.5 Memory Peripherals . . . 36
2.6 Software Handshaking . . . 36
2.7 Handshaking Interface – Peripheral Is Not Flow Controller . . . 37
2.7.1 Single Transaction Region . . . 38
2.7.2 Early-Terminated Burst Transaction . . . 39
2.7.3 Hardware Handshaking – Peripheral Is Not Flow Controller . . . 40

2.7.4 Software Handshaking – Peripheral Is Not Flow Controller 46
2.7.5 Single Transactions – Peripheral Is Not Flow Controller 47
2.8 Handshaking Interface – Peripheral Is Flow Controller 48
2.8.1 Hardware Handshaking – Peripheral Is Flow Controller 49
2.8.2 Software Handshaking – Peripheral Is Flow Controller 51
2.8.3 Single Transactions – Peripheral is Flow Controller 52
2.9 Setting Up Transfers 52
2.9.1 Transfer Operation 53
2.9.2 Peripheral Interrupt Request Interface 87
2.10 Flow Control Configurations 88
2.11 Peripheral Burst Transaction Requests 90
2.11.1 Transmit Watermark Level and Transmit FIFO Underflow 90
2.11.2 Choosing the Transmit Watermark Level 90
2.11.3 Selecting CTL*x*.DEST_MSIZE and Transmit FIFO Overflow 93
2.11.4 Receive Watermark Level and Receive FIFO Overflow 93
2.11.5 Choosing the Receive Watermark level 94
2.11.6 Selecting CTL*x*.SRC_MSIZE and Receive FIFO Underflow 94
2.12 Generating Requests for the AHB Master Bus Interface 94
2.12.1 Locked DMA Transfers 96
2.13 Arbitration for AHB Master Interface 98
2.14 Latency 99
2.15 Scatter/Gather 101
2.16 Endianness 104
2.16.1 Big Endian-Little Endian Conversion Logic 104
2.16.2 LLI Fetch, and Status and Control Write-Back 105
2.16.3 Endian Selection 105
2.16.4 Static Endian Configuration 107
2.16.5 Dynamic Endian Configuration 107
2.17 AHB Transfer Error Handling 108
2.18 Last Beat of DMA Burst Indication 109
2.18.1 Example 1 109
2.19 Low Power Modes – Global and Channel Clock Gating 111
2.19.1 Global Clock Gating 111
2.19.2 Channel Clock Gating 113
2.20 Interrupt Registers 115

Chapter 3
Parameter Descriptions 117
3.1 DMA Source Code Configuration Parameters 118
3.2 Global DMA Configuration Parameters 119
3.3 Configuration of AMBA layers Parameters 125
3.4 Channel x configuration Parameters 129

Chapter 4
Signal Descriptions 139
4.1 Slave Interface Signals 141
4.2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals 144
4.3 Test Interface Signals 148
4.4 Peripheral Handshaking Interface Signals 149

4.5 Interrupt Interface Signals 151
4.6 Debug Bus Interface Signals 153

Chapter 5
Register Descriptions 161
5.1 DMAC/Channel_x_Registers (for x = 1; x <= DMAH_NUM_CHANNELS-1) Registers 165
5.1.1 SARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 167
5.1.2 DARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 168
5.1.3 LLPx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 169
5.1.4 CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 171
5.1.5 SSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 182
5.1.6 DSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 184
5.1.7 SSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 186
5.1.8 DSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 188
5.1.9 CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 190
5.1.10 SGRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 199
5.1.11 DSRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) 201
5.2 DMAC/Interrupt Registers 203
5.2.1 RawTfr 205
5.2.2 RawBlock 206
5.2.3 RawSrcTran 207
5.2.4 RawDstTran 209
5.2.5 RawErr 210
5.2.6 StatusTfr 211
5.2.7 StatusBlock 212
5.2.8 StatusSrcTran 213
5.2.9 StatusDstTran 214
5.2.10 StatusErr 215
5.2.11 MaskTfr 216
5.2.12 MaskBlock 218
5.2.13 MaskSrcTran 220
5.2.14 MaskDstTran 222
5.2.15 MaskErr 224
5.2.16 ClearTfr 226
5.2.17 ClearBlock 227
5.2.18 ClearSrcTran 228
5.2.19 ClearDstTran 229
5.2.20 ClearErr 230
5.2.21 StatusInt 231
5.3 DMAC/Software_Handshake Registers 233
5.3.1 ReqSrcReg 234
5.3.2 ReqDstReg 236
5.3.3 SglRqSrcReg 238
5.3.4 SglRqDstReg 240
5.3.5 LstSrcReg 242
5.3.6 LstDstReg 244
5.4 DMAC/Miscellaneous Registers 246
5.4.1 DmaCfgReg 247
5.4.2 ChEnReg 248

5.4.3 DmaIdReg . . . 250
5.4.4 DmaTestReg . . . 251
5.4.5 DmaLpTimeoutReg . . . 252
5.4.6 DMA_COMP_PARAMS_6 . . . 253
5.4.7 DMA_COMP_PARAMS_5 . . . 261
5.4.8 DMA_COMP_PARAMS_4 . . . 274
5.4.9 DMA_COMP_PARAMS_3 . . . 287
5.4.10 DMA_COMP_PARAMS_2 . . . 300
5.4.11 DMA_COMP_PARAMS_1 . . . 315
5.4.12 DmaCompsID . . . 329

Chapter 6
Programming the DW_ahb_dmac . . . 331
6.1 Software Drivers . . . 331
6.2 Register Access . . . 332
6.3 Illegal Register Access . . . 332
6.4 DW_ahb_dmac Transfer Types . . . 332
6.4.1 Multi-Block Transfers . . . 333
6.4.2 Auto-Reloading of Channel Registers . . . 337
6.4.3 Contiguous Address Between Blocks . . . 337
6.4.4 Suspension of Transfers Between Blocks . . . 338
6.4.5 Ending Multi-Block Transfers . . . 338
6.5 Programing Examples . . . 339
6.5.1 Programming Example for Linked List Multi-Block Transfer . . . 341
6.6 Programming a Channel . . . 342
6.6.1 Programming Examples . . . 343
6.7 Disabling a Channel Prior to Transfer Completion . . . 367
6.7.1 Abnormal Transfer Termination . . . 368
6.8 Defined-Length Burst Support on DW_ahb_dmac . . . 368

Chapter 7
Verification . . . 369
7.1 Overview of Vera Tests . . . 369
7.2 Overview of DW_ahb_dmac Testbench . . . 370

Chapter 8
Integration Considerations . . . 373
8.1 Performance . . . 373
8.1.1 Power Consumption, Frequency, and Area Results . . . 373
8.2 1KB Boundary Crossing . . . 376
8.3 Read Accesses . . . 376
8.4 Write Accesses . . . 377
8.5 Consecutive Write-Read . . . 377
8.6 Accessing Top-level Constraints . . . 379
8.7 Coherency . . . 379
8.7.1 Writing Coherently . . . 379
8.7.2 Reading Coherently . . . 385

Appendix A
Error and Warning Messages . . . 389

A.1 Warnings During Simulation 389
A.2 Warnings During Synthesis 389

Chapter B
Internal Parameter Descriptions 391

Appendix C
Channel Locking and Deadlock 409
C.1 Hardware Detection of Deadlock 409
C.1.1 Case 1 409
C.1.2 Case 2 410
C.1.3 Case 3 411
C.1.4 Deadlock Prevention by Hardware 412
C.2 Programming Restrictions to Avoid Deadlock 412

Appendix D
DW_ahb_dmac Application Notes 415
D.1 Interoperability Between DW_ahb_dmac and PrimeCell Hardware Handshaking Interface 415
D.2 Mapping of PrimeCell Software Handshaking Registers to DW_ahb_dmac 417
D.2.1 PrimeCell Software Handshaking Registers 417
D.2.2 DW_ahb_dmac Software Handshaking Registers 418
D.2.3 Register Interface Mapping 418

Appendix E
Configuring DW_ahb_dmac to Match Arm PrimeCell PL080/PL081 423
E.1 ARM PL080 Equivalent 423
E.2 ARM PL081 Equivalent 423

Appendix F
Glossary 427

Index 431

Revision History

This table shows the revision history for the databook from release to release. This is being tracked from version 2.10b onward.

Version	Date	Description
2.22a	July 2018	**Updated**: ■ Version changed for 2018.07a release ■ "Performance" on page 373 ■ "Parameter Descriptions" on page 117, "Register Descriptions" on page 161, "Signal Descriptions" on page 139, and "Internal Parameter Descriptions" on page 391 are auto-extracted with change bars from the RTL **Removed**: ■ Chapter 2, "Building and Verifying a Component or Subsystem" and added the contents in the newly created user guide.

(Continued)

<table>
<tr><th>Version</th><th>Date</th><th>Description</th></tr>
<tr><td>2.21a</td><td>October 2016</td><td>Added:
■ “Low Power Modes – Global and Channel Clock Gating” on page 111
■ “Low Power Mode Parameters” parameters:
- DMAH_LP_EN
- DMAH_CH_LP_EN
- DMAH_LP_TIMEOUT_WIDTH
- DMAH_HC_LP_TIMEOUT_VALUE
- DMAH_LP_TIMEOUT_VALUE
■ Low Power Count Register (DmaLpTimeoutReg)
■ “Parameter Descriptions” on page 117 and “Register Descriptions” on page 161 auto-extracted from the RTL
■ Xprop directory in Table 2-1and Table 2-4
■ “Running VCS XPROP Analyzer”
■ Note in “Hardware Handshaking – Peripheral Is Not Flow Controller” on page 40 and “Hardware Handshaking – Peripheral Is Flow Controller” on page 49
Deleted:
■ “Running Leda on Generated Code with coreConsultant”, and reference to Leda directory in Table 2-1
■ “Running Leda on Generated Code with coreAssembler” section, and reference to Leda directory in Table 2-4
Moved:
■ Table 2-5 and Table 2-6 to “Transfer Operation” on page 53
■ Internal Parameter Descriptions to Appendix
■ “Interrupt Registers” on page 115 from Registers chapter
■ “Hardware Realignment of SAR/DAR Registers” on page 65 from Registers chapter to “ Functional Description” on page 33
Updated:
■ Version changed for 2016.10a release
■ Table 2-1</td></tr>
<tr><td>2.20a</td><td>June 2015</td><td>Added:
■ “Running SpyGlass® Lint and SpyGlass® CDC”
■ “Running SpyGlass on Generated Code with coreAssembler”
■ Chapter B, “Internal Parameter Descriptions”
Updated:
Chapter 4, “Signal Descriptions” auto-extracted from the RTL</td></tr>
</table>

(Continued)

Version	Date	Description
2.19a	June 2014	**Updated**: ■ Version for 2014.06a release ■ Big endian BE32 format for data transfer on AHB master interface ■ Performance section in Integration consideration ■ Default Input/Output Delay in the following Signal groups: - Slave Interface - Master *N* Interface
2.18b	May 2013	**Added:** ■ An output signal dma_wlast on the AHB interface to indicate the last write data during burst transfers to destination peripherals ■ Configuration parameter DMAH_WLAST_EN to enable the dma_wlast signal ■ Added section "Last Beat of DMA Burst Indication" to explain last write data during bursts transfers feature **Removed:** ■ Note stating that disabling the channel through software prior to completing a transfer is not supported when DW_ahb_dmac is configured to use defined length bursts; feature now supported **Updated:** ■ Corrected the sequence of bits in the DMA_COMP_PARAMS_1 register ■ Document template
2.17d	Sep 2012	Added the product code on the cover
2.17d	Mar 2012	**Updated:** ■ Instructions for setting bit 0 of DmaCfgReg register ■ Descriptions of SSTATARx and DSTATARx registers ■ Information in Early-Terminated Burst Transaction section
2.17c	Nov 2011	Version change for 2011.11a release
2.17b	Oct 2011	**Updated:** ■ Corrected dma_req and dma_single as being registered ■ SSTATARx and DSTATARx register descriptions
2.17a	Jun 2011	**Updated:** ■ System diagram in Figure 1-1 ■ "Related Documentation" section in Preface
2.17a	May 2011	Edited "Burst transaction" definition

(Continued)

Version	Date	Description
2.17a	Apr 2011	**Added:** ■ New “Latency” section ■ New DMAH_REVERSE_WB_OVERRIDE parameter ■ Information to “Memory Peripherals” section about impact of CTLx.SRC_MSIZE, CTLx. DEST_MSIZE values on burst transfers to/from memory peripherals
2.16a	Dec 2010	Version change for 2010.12a release
2.15a	Sep 2010	**Updated:** ■ Descriptions for dma_req signal ■ Corrected names of include files and vcs command used for simulation ■ Corrected defaults for DMAH_CHx_SMS and DMAH_CHx_DMS
2.14a	May 2010	Updated the DMAH_INTR_POL parameter description
2.14a	Dec 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks
2.14a	Jul 2009	Corrected equations for avoiding underflow when programming a source burst transaction
2.14a	May 2009	Removed references to QuickStarts, as they are no longer supported
2.14a	Apr 2009	**Added:** Note that multi-block transfers not supported **Updated:** Corrected DMAH_CHx_FIFO_DEPTH equation
2.14a	Nov 2008	Corrected CFG* reset values in memory map
2.14a	Oct 2008	Version change for 2008.10a release
2.12a	Jul 2008	Corrected Setting Up Transfers example #4 calculation
2.12a	Jun 2008	**Updated:** ■ Descriptions for software handshaking registers to say that channel must be enabled to allow writing to a bit ■ Descriptions for INT_EN register
2.11a	Mar 2008	**Updated:** ■ CFGx.FIFO_EMPTY default to 0x1 ■ Text for value of 1 for CFGx.FIFO_MODE

(Continued)

Version	Date	Description
2.10b	Jan 2008	**Updated:** ■ Table 23 (DMAH_CHx_CTL_WB_EN, DMAH_CHx_MULTI_BLK_TYPE) ■ Table 25 (SAR4, SGR4) ■ “Hardware Realignment of SAR/DAR Registers” section ■ Figure 71, LLPI register definition
2.10b	July 2007	Corrected available files in workspace/sim/test_name directory
2.10b	June 2007	Version change for 2007.06a release

Preface

This databook provides information that you need to interface the DesignWare AHB Central Direct Memory Access (DMA) Controller, referred to as DW_ahb_dmac throughout the remainder of this databook. This component conforms to the *AMBA Specification, Revision 2.0* from Arm®.

The information in this databook includes a functional description, signal and parameter descriptions, and a memory map. Also provided are an overview of the component testbench, a description of the tests that are run to verify the coreKit, and synthesis information for the component.

Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Product Overview" provides a system overview, a component block diagram, basic features, and an overview of the verification environment.
- Chapter 2, " Functional Description" describes the functional operation of the DW_ahb_dmac.
- Chapter 3, "Parameter Descriptions" identifies the configurable parameters supported by the DW_ahb_dmac.
- Chapter 4, "Signal Descriptions" provides a list and description of the DW_ahb_dmac signals.
- Chapter 5, "Register Descriptions" describes the programmable registers of the DW_ahb_dmac.
- Chapter 6, "Programming the DW_ahb_dmac" provides information needed to program the configured DW_ahb_dmac.
- Chapter 7, "Verification" provides information on verifying the configured DW_ahb_dmac.
- Chapter 8, "Integration Considerations" includes information you need to integrate the configured DW_ahb_dmac into your design.
- Appendix A, "Error and Warning Messages" describes errors and warnings that you may encounter when configuring, verifying, or synthesizing the DesignWare component using the coreTools GUIs.
- Appendix B, "Internal Parameter Descriptions" provides a list of internal parameter descriptions that might be indirectly referenced in expressions in the Signals, Parameters, or Registers chapters..
- Appendix C, "Channel Locking and Deadlock" explains how channel locking can cause deadlock.
- Appendix D, "DW_ahb_dmac Application Notes" describes the interoperability between DW_ahb_dmac and Arm® PrimeCell handshaking interface.

- Appendix E, "Configuring DW_ahb_dmac to Match Arm PrimeCell PL080/PL081" provides configuration parameter settings for DW_ahb_dmac so that it matches or is equivalent to the Arm® PrimeCell PL080 and PL081 devices.
- Appendix F, "Glossary" provides a glossary of general terms.

Related Documentation

- *DW_ahb_dmac Driver Kit User Guide* – Contains information on the Driver Kit for the DW_ahb_dmac; requires source code license (DWC-APB-Periph-Source)
- *Using DesignWare Library IP in coreAssembler* – Contains information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools
- *coreAssembler User Guide* – Contains information on using coreAssembler
- *coreConsultant User Guide* – Contains information on using coreConsultant

To see a complete listing of documentation within the DesignWare Synthesizable Components for AMBA 2, refer to the *Guide to Documentation for DesignWare Synthesizable Components for AMBA 2 and AMBA 3 AXI*.

Web Resources

- DesignWare IP product information: http://www.designware.com
- Your custom DesignWare IP page: http://www.mydesignware.com
- Documentation through SolvNet: http://solvnet.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): http://www.synopsys.com/keys

Customer Support

To obtain support for your product:

- First, prepare the following debug information, if applicable:
  - For environment setup problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, use the following menu entry:

    File > Build Debug Tar-file

    Check all the boxes in the dialog box that apply to your issue. This menu entry gathers all the Synopsys product data needed to begin debugging an issue and writes it to the file <*core tool startup directory*>/debug.tar.gz.
  - For simulation issues outside of coreConsultant or coreAssembler:
    - Create a waveforms file (such as VPD or VCD)
    - Identify the hierarchy path to the DesignWare instance
    - Identify the timestamp of any signals or locations in the waveforms that are not understood
- Then, contact Support Center, with a description of your question and supplying the requested information, using one of the following methods:

- *For fastest response*, use the SolvNet website. If you fill in your information as explained, your issue is automatically routed to a support engineer who is experienced with your product. The **Sub Product** entry is critical for correct routing.

  Go to http://solvnet.synopsys.com/EnterACall and click the **Open A Support Case** to enter a call.
  Provide the requested information, including:

  - **Product:** DesignWare Library IP
  - **Sub Product:** AMBA
  - **Tool Version:** *product version number*
  - **Problem Type:**
  - **Priority:**
  - **Title:** DW_ahb_dmac
  - **Description:** For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood

  After creating the case, attach any debug files you created in the previous step.

- Or, send an e-mail message to support_center@synopsys.com (your email will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):

  - Include the Product name, Sub Product name, and Tool Version number in your e-mail (as identified earlier) so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood
  - Attach any debug files you created in the previous step.

- Or, telephone your local support center:

  - North America:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:
    http://www.synopsys.com/Support/GlobalSupportCenters

1
Product Overview

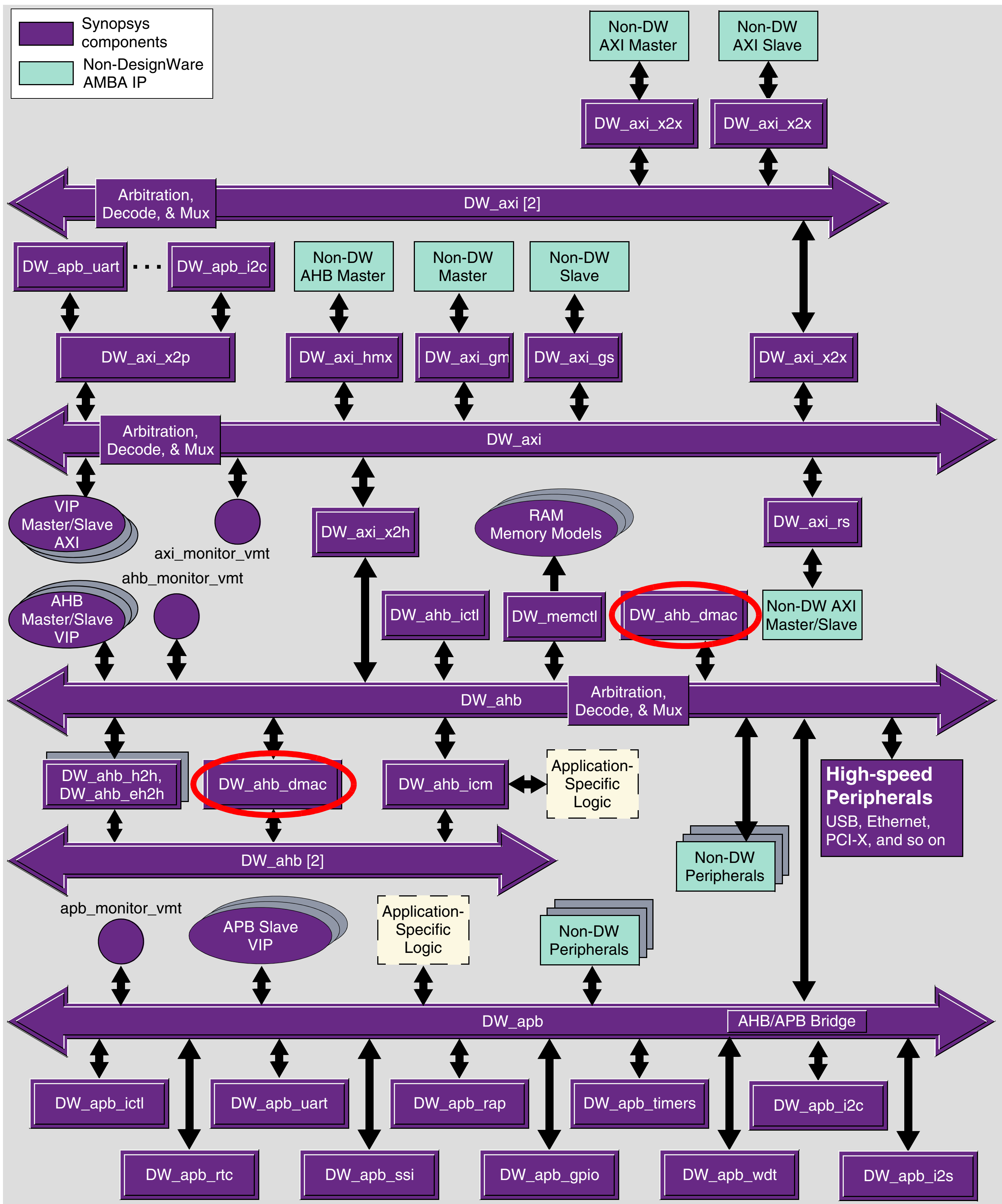
This chapter provides a basic overview of the DW_ahb_dmac, which is an AHB-Central DMA Controller core that transfers data from a source peripheral to a destination peripheral over one or more AHB bus.

1.1 DesignWare System Overview

The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing AMBA version 2.0-compliant AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus) components, and AMBA version 3.0-compliant AXI (Advanced eXtensible Interface) components.

Figure 1-1 illustrates one example of this environment, including the AXI bus, the AHB bus, and the APB bus. Included in this subsystem are synthesizable IP for AXI/AHB/APB peripherals, bus bridges, and an AXI interconnect and AHB bus fabric. Also included are verification IP for AXI/AHB/APB master/slave models and bus monitors. In order to display the databook for a DW_* component, click on the corresponding component object in the illustration.

Figure 1-1 Example of DW_ahb_dmac in a Complete System



You can connect, configure, synthesize, and verify the DW_ahb_dmac within a DesignWare subsystem using coreAssembler, documentation for which is available on the web in the *coreAssembler User Guide*.

If you want to configure, synthesize, and verify a single component such as the DW_ahb_dmac component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

1.2 General Product Description

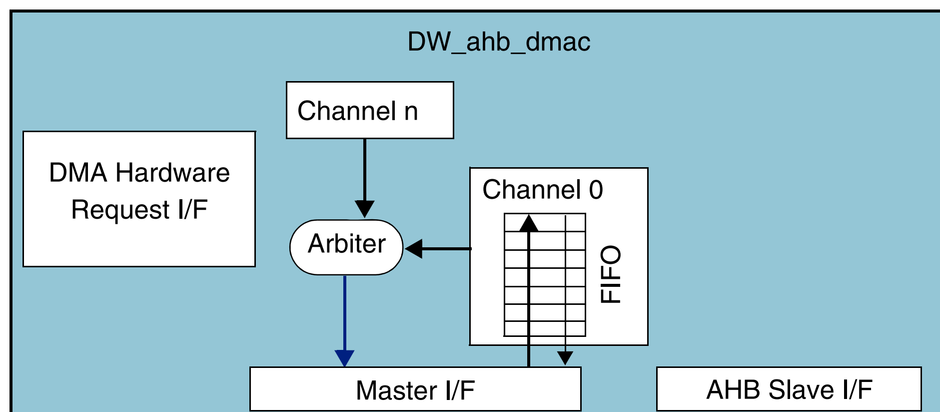
The Synopsys DW_ahb_dmac conforms to the *AMBA Specification, Revision 2.0* from ARM.

1.2.1 DW_ahb_dmac Block Diagram

Figure 1-2 shows the following functional groupings of the main interfaces to the DW_ahb_dmac block:

- DMA hardware request interface
- Up to eight channels
- FIFO per channel for source and destination
- Arbiter
- AHB master interface
- AHB slave interface

Figure 1-2 DW_ahb_dmac Block Diagram

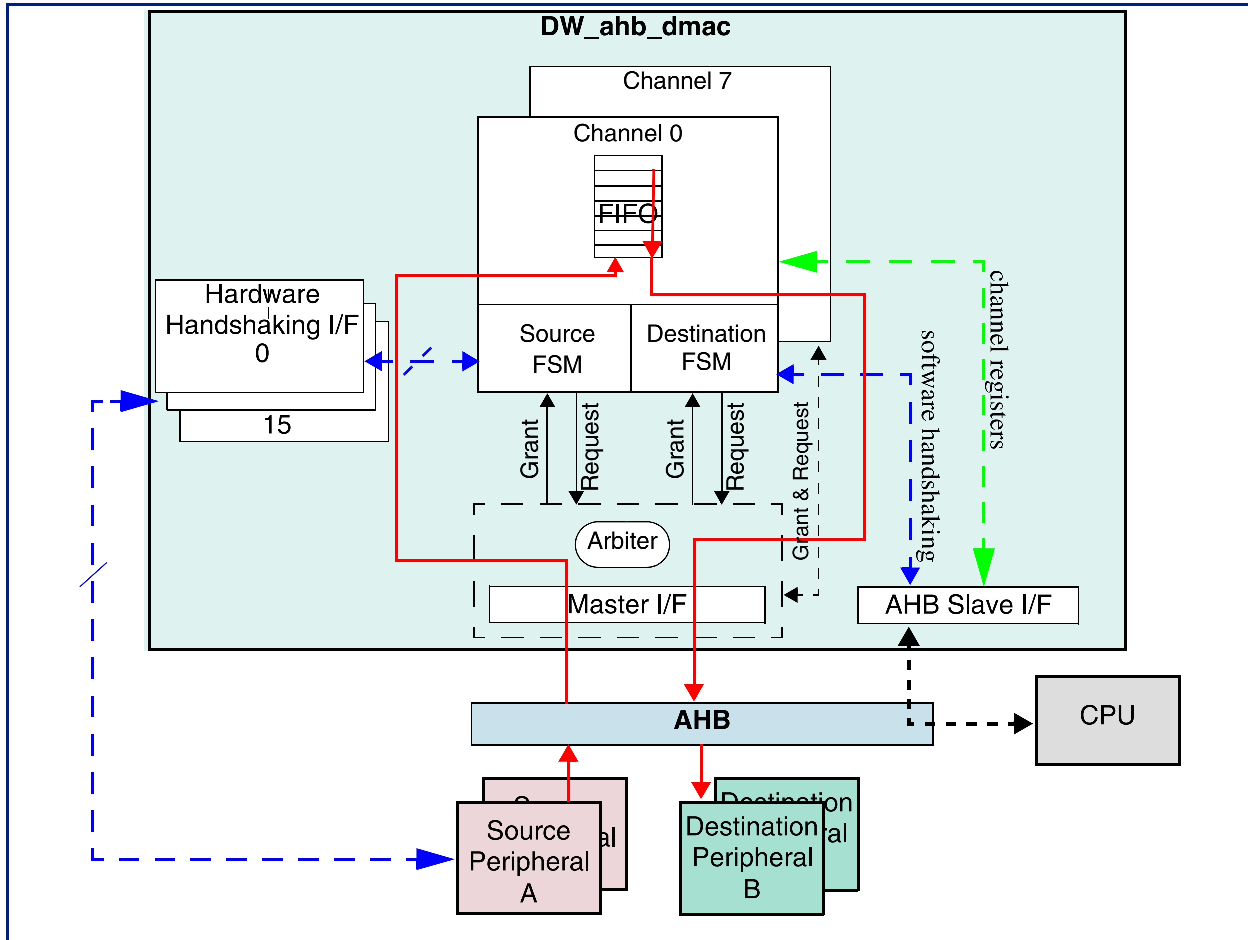


One channel of the DW_ahb_dmac is required for each source/destination pair. In the most basic configurations, as illustrated in Figure 1-3 on page 22, the DW_ahb_dmac has one master interface and one channel. The master interface reads the data from a source peripheral (A) and writes it to a destination peripheral (B). Two AHB transfers are required for each DMA data transfer; this is also known as a dual-access transfer.

Figure 1-3 illustrates a peripheral-to-peripheral DMA transfer, where peripheral A (source) uses a hardware handshaking interface, and peripheral B (destination) uses a software handshaking interface. For example, the request to send data to peripheral B is originated by the CPU, while writing to peripheral B is handled by the DW_ahb_dmac. The channel source and destination arbitrate independently for the AHB master

interface, along with other channels. For more information about the arbitration scheme, see "Arbitration for AHB Master Interface" on page 98.

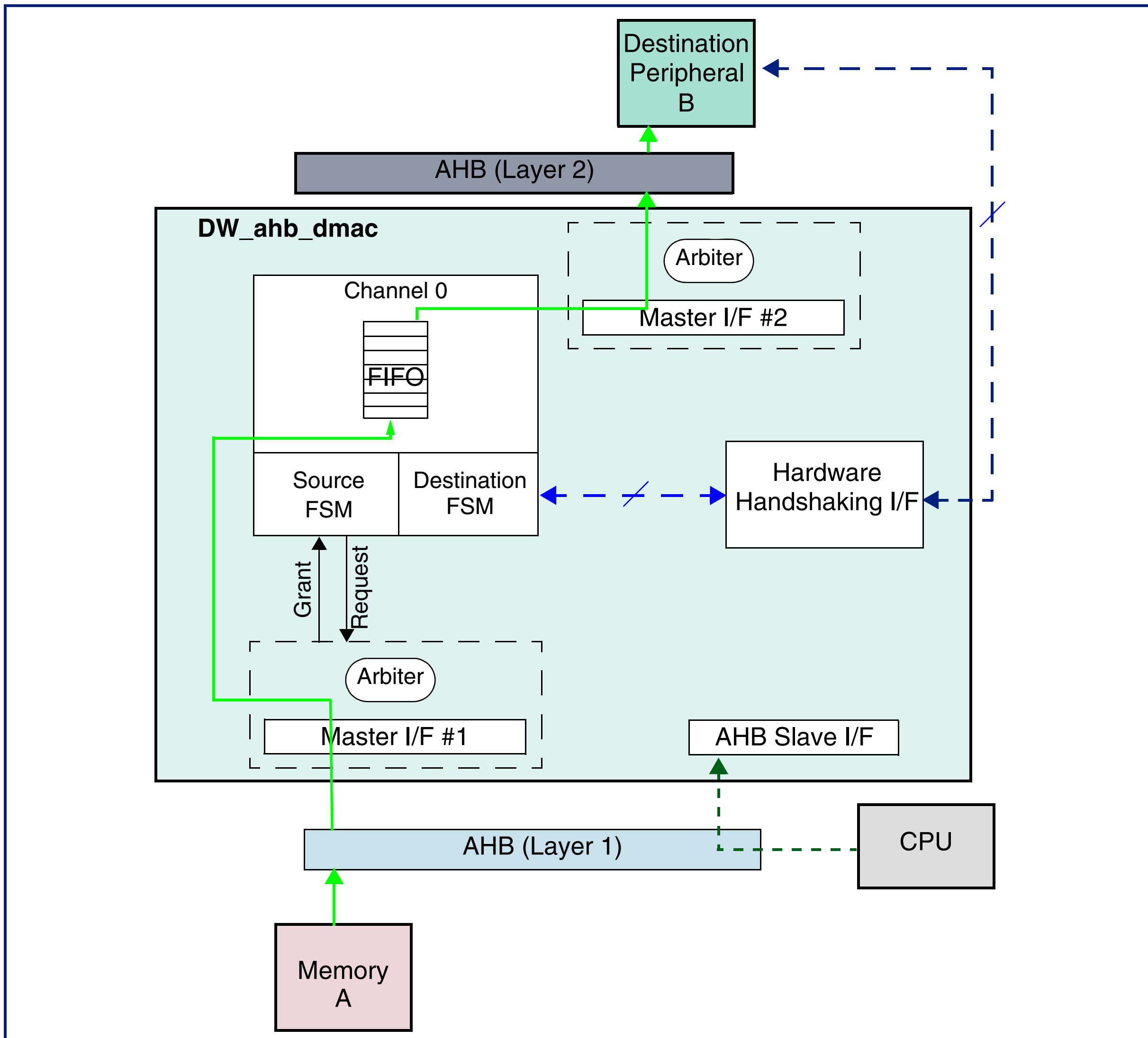
Figure 1-3 Peripheral-to-Peripheral DMA Transfer on Same AHB Layer



The DW_ahb_dmac also supports multi-layer DMA transfers when the source and destination peripherals are on different AHB layers. In this case, you must configure the DW_ahb_dmac to have more than one master interface—one per layer. Figure 1-4 on page 23 illustrates a DW_ahb_dmac with two master interfaces and a DMA transfer between a source and destination on different AHB layers. Peripheral B uses

a hardware handshaking interface. The memory does not use any handshaking interface to the DW_ahb_dmac in order to initiate DMA transfers.

Figure 1-4 Peripheral-to-Memory DMA Transfer on Separate AHB Layers



1.3 Basic Definitions

The following terms are concise definitions of the DMA concepts used throughout this databook:

- **Source peripheral** – Device on a AHB layer from which the DW_ahb_dmac reads data; the DW_ahb_dmac then stores the data in the channel FIFO. The source peripheral teams up with a destination peripheral to form a channel. The source peripheral is either an AHB or APB slave. If the source is an APB slave, it is accessed through the AHB-APB bridge.
- **Destination peripheral** – Device to which the DW_ahb_dmac writes the stored data from the FIFO (previously read from the source peripheral). The destination peripheral is either an AHB or APB slave. If the destination is an APB slave, it is accessed through the AHB-APB bridge.

- **Memory** – Source or destination that is always "ready" for a DMA transfer and does not require a handshaking interface to interact with the DW_ahb_dmac. A peripheral should be assigned as memory only if it does not insert more than 16 wait states. If more than 16 wait states are required, then the peripheral should use a handshaking interface—the default if the peripheral is not programmed to be memory—in order to signal when the peripheral is ready to accept or supply data. A memory peripheral can also generate SPLIT/RETRY responses.
- **Channel** – Read/write data path between a source peripheral on one configured AHB layer and a destination peripheral on the same or different AHB layer that occurs through the channel FIFO. If the source peripheral is not memory, then a source handshaking interface is assigned to the channel. If the destination peripheral is not memory, then a destination handshaking interface is assigned to the channel. Source and destination handshaking interfaces can be assigned dynamically by programming the channel registers.
- **Master interface** – DW_ahb_dmac is a master on the AHB bus, reading data from the source and writing it to the destination over the AHB bus. It is possible to have up to four master interfaces, which means that up to four independent source and destination channels can operate simultaneously. Each channel has to arbitrate for the master interface. You need to have more than one master interface if the source and destination peripherals reside on different AHB layers.
- **Slave interface** – The AHB interface over which the DW_ahb_dmac is programmed. The slave interface in practice can be on the same layer as any of the master interfaces, or it can be on a separate layer.
- **Handshaking interface** – A set of signals or software registers that conform to a protocol and handshake between the DW_ahb_dmac and source or destination peripheral in order to control transferring a single or burst transaction between them. This interface is used to request, acknowledge, and control a DW_ahb_dmac transaction. A channel can receive a request through one of three types of handshaking interface: hardware, software, or peripheral interrupt.
  - **Hardware handshaking interface** – Uses hardware signals to control transferring a single or burst transaction between the DW_ahb_dmac and the source or destination peripheral. For more information about this interface, see "Hardware Handshaking – Peripheral Is Not Flow Controller" on page 40 and "Hardware Handshaking – Peripheral Is Flow Controller" on page 49.
  - **Software handshaking interface**– Uses software registers to control transferring a single or burst transaction between the DW_ahb_dmac and the source or destination peripheral. No special DW_ahb_dmac handshaking signals are needed on the I/O of the peripheral. This mode is useful for interfacing an existing peripheral to the DW_ahb_dmac without modifying it. For more information about this interface, see "Flow Control Configurations" on page 88.
  - **Peripheral interrupt handshaking interface** – Simple use of the hardware handshaking interface. In this mode, the interrupt line from the peripheral is tied to the dma_req input of the hardware handshaking interface; other interface signals are ignored. For more information about this interface, see "Peripheral Interrupt Request Interface" on page 87.
- **Flow controller** – Device (either the DW_ahb_dmac, or source/destination peripheral) that determines the length of a DMA block transfer and terminates it.
  - If you know the length of a block before enabling the channel, then you should program the DW_ahb_dmac as the flow controller.

- ❑ If the length of a block is not known prior to enabling the channel, the source or destination peripheral needs to terminate a block transfer. In this mode, the peripheral is the flow controller.

For more information, see "Setup/Operation of DW_ahb_dmac Transfers" on page 33.

- **Flow control mode** (CFGx.FCMODE) – Special mode that only applies when the destination peripheral is the flow controller. It controls the data pre-fetching from the source peripheral.
- **Transfer hierarchy** – Figure 1-5 illustrates the hierarchy between DW_ahb_dmac transfers, block transfers, transactions (single or burst), and AHB transfers (single or burst) for non-memory peripherals.

Note Note that for memory peripherals, there is no DMA Transaction Level.

Figure 1-5 DW_ahb_dmac Transfer Hierarchy for Non-Memory Peripherals

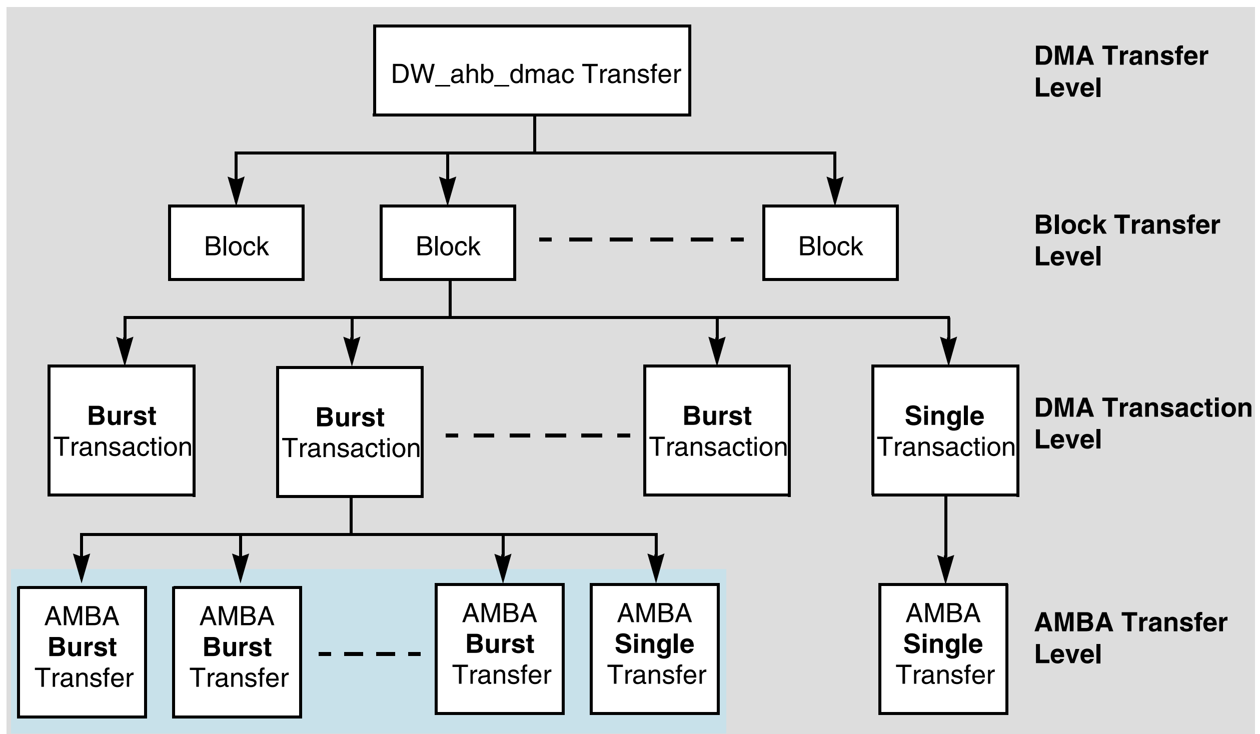
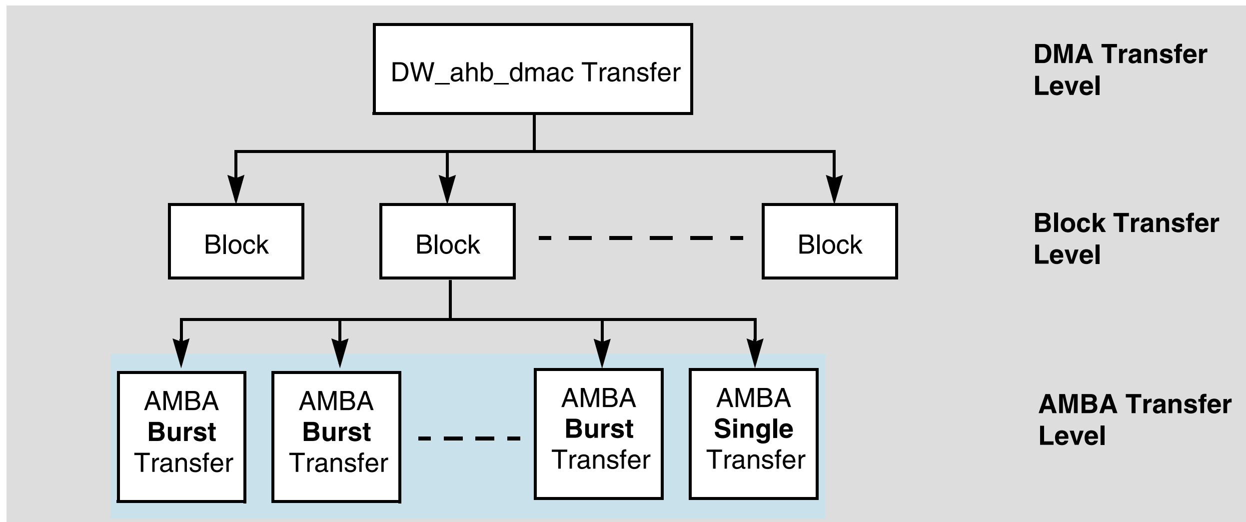


Figure 1-6 shows the transfer hierarchy for memory.

Figure 1-6 DW_ahb_dmac Transfer Hierarchy for Memory



- **Block** – Block of DW_ahb_dmac data, the amount of which is the block length and is determined by the flow controller. For transfers between the DW_ahb_dmac and memory, a block is broken directly into a sequence of bursts and single transfers. For transfers between the DW_ahb_dmac and a non-memory peripheral, a block is broken into a sequence of DW_ahb_dmac transactions (single and bursts). These are in turn broken into a sequence of AHB transfers.
- Transaction – Basic unit of a DW_ahb_dmac transfer, as determined by either the hardware or software handshaking interface. A transaction is relevant only for transfers between the DW_ahb_dmac and a source or destination peripheral if the peripheral is a non-memory device. There are two types of transactions:
  - **Single transaction** – Length of a single transaction is always 1 and is converted to a single AHB transfer.
  - **Burst transaction** – Length of a burst transaction is programmed into the DW_ahb_dmac. The burst transaction is converted into a sequence of bursts and AHB single transfers. The burst transaction length is under program control and normally bears some relationship to the FIFO sizes in the DW_ahb_dmac and in the source and destination peripherals.
- **DMA transfer** – Software controls the number of blocks in a DW_ahb_dmac transfer. Once the DMA transfer has completed, the hardware within the DW_ahb_dmac disables the channel and can generate an interrupt to signal the DMA transfer completion. You can then reprogram the channel for a new DMA transfer.
  - **Single-block DMA transfer** – Consists of a single block.
  - **Multi-block DMA transfer** – DMA transfer may consist of multiple DW_ahb_dmac blocks. Multi-block DMA transfers are supported through block chaining (linked list pointers), auto-reloading channel registers, and contiguous blocks. The source and destination can independently select which method to use.
    - **Linked lists (block chaining)** – Linked list pointer (LLP) points to the location in system memory where the next linked list item (LLI) exists. The LLI is a set of registers that describes the next block (block descriptor) and an LLP register. The DW_ahb_dmac fetches the LLI at

the beginning of every block when block chaining is enabled. For more information about linked lists and block chaining, see "Block Chaining Using Linked Lists" on page 333.

LLI accesses are always 32-bit accesses (Hsize = 2) aligned to 32-bit boundaries and cannot be changed or programmed to anything other than 32-bit, even if the AHB master interface of the LLI supports more than a 32-bit data width.

- **Auto-reloading** – DW_ahb_dmac automatically reloads the channel registers at the end of each block to the value when the channel was first enabled. For more information about this function, see "Auto-Reloading of Channel Registers" on page 337.
- **Contiguous blocks** – Address between successive blocks is selected to be a continuation from the end of the previous block. For more information, see "Contiguous Address Between Blocks" on page 337.

- **Scatter** – Relevant to destination transfers within a block. The destination address is incremented or decremented by a programmed amount when a scatter boundary is reached. The number of AHB transfers between successive scatter boundaries is under software control.
- **Gather** – Relevant to source transfers within a block. The source address is incremented or decremented by a programmed amount when a gather boundary is reached. The number of AHB transfers between successive gather boundaries is under software control.

  For more information about scatter and gather, see "Scatter/Gather" on page 101.
- **Channel locking** – Software can program a channel to keep the AHB master interface by locking arbitration of the master bus interface for the duration of a DMA transfer, block, or transaction (single or burst). For more information on channel locking, see "Channel Locking" on page 96.
- **Bus locking** – Software can program a channel to maintain control of the AHB bus by asserting hlock for the duration of a DMA transfer, block, or transaction (single or burst). At minimum, channel locking is asserted during bus locking. For more information, see "Bus Locking" on page 96.
- **FIFO mode** – Special mode to improve bandwidth. When enabled, the channel waits until the FIFO is less than half full to fetch the data from the source peripheral, and waits until the FIFO is greater than or equal to half full in order to send data to the destination peripheral. Because of this, the channel can transfer the data using bursts, which eliminates the need to arbitrate for the AHB master interface in each single AHB transfer. When this mode is not enabled, the channel waits only until the FIFO can transmit or accept a single AHB transfer before it requests the master bus interface.
- **Pseudo fly-by operation** – Typically, it takes two AHB bus cycles to complete a transfer—one for reading the source and one for writing to the destination. However, when the source and destination peripherals of a DMA transfer are on different AHB layers, it is possible for the DW_ahb_dmac to fetch data from the source and store it in the channel FIFO at the same time that the DW_ahb_dmac extracts data from the channel FIFO and writes it to the destination peripheral. This activity is known as *pseudo fly-by operation*. In order for this to occur in appropriate sequential order, the source and destination logic in the DW_ahb_dmac should first win their respective master interfaces, and then the master interface for both source and destination layers must win arbitration of their AHB layer.

1.4 Features

The DW_ahb_dmac component includes the following features.

1.4.1 General

- AMBA 2.0-compliant
- AHB slave interface – used to program the DW_ahb_dmac
- Channels
    - Up to eight channels, one per source and destination pair
    - Unidirectional channels – data transfers in one direction only
    - Programmable channel priority
- AHB master interfaces
    - Up to four independent AHB master interfaces that allows:
        - Up to four simultaneous DMA transfers
        - Masters that can be on different AHB layers (multi-layer support)
        - Source and destination that can be on different AHB layers (pseudo fly-by performance)
    - Configurable data bus width (up to 256 bits) for each AHB master interface
    - Configurable endianness for master interfaces
- Transfers
    - Support for memory-to-memory, memory-to-peripheral, peripheral-to-memory, and peripheral-to-peripheral DMA transfers
    - DW_ahb_dmac to or from APB peripherals through the APB bridge
- Configurable identification register
- Component ID parameters for configurable software driver support
- Configuration of DesignWare AHB Lite system
- DMA burst indication on the last beat
- Support for Little Endian, Address Invariant (AI) Big Endian scheme and BE-32 (Word Invariant) scheme of data access on AHB slave interface and each AHB master interface.
- Arbitration scheme to decide which of the request lines is granted access to a particular master bus interface.

Source code for this component is available on a per-project basis as a DesignWare Core. Contact your local sales office for the details.

1.4.2 Address Generation

- Programmable source and destination addresses (on AHB bus)
- Address increment, decrement, or no change
- Multi-block transfers achieved through:
  - Linked Lists (block chaining)
  - Auto-reloading of channel registers
  - Contiguous address between blocks
- Independent source and destination selection of multi-block transfer type
- Scatter/Gather

1.4.3 Channel Buffering

- Single FIFO per channel for source and destination
- Configurable FIFO depth
- D flip-flop-based FIFO
- Automatic data packing or unpacking to fit FIFO width

1.4.4 Channel Control

- Programmable source and destination for each channel
- Programmable transfer type for each channel (memory-to-memory, memory-to-peripheral, peripheral-to-memory, and peripheral-to-peripheral)
- Programmable burst transaction size for each channel
- Programmable enable and disable of DMA channel
- Support for disabling channel without data loss
- Support for suspension of DMA operation
- Support for RETRY, SPLIT, and ERROR responses
- Programmable maximum burst transfer size per channel
- Configurable maximum transaction size to allow gate optimization
- Configurable maximum block size to allow gate optimization
- Bus locking – can be programmed to be over the transaction, block, or DMA transfer level
- Channel locking – can be programmed to be over the transaction, block, or DMA transfer level
- Option to hardcode type of multi-block transfer
- Option to disable the writeback of the Channel Control register at the end of every block transfer

1.4.5 Transfer Initiation

- Handshaking interfaces for source and destination peripherals (up to 16)
  - Hardware handshaking interface
  - Software handshaking interface
  - Peripheral interrupt handshaking interface
- Handshaking interface supports single or burst DMA transactions
- Polarity control for hardware handshaking interface
- Enabling and disabling of individual DMA handshaking interfaces

1.4.6 Flow Control

- Programmable flow control at block transfer level (source, destination, or DW_ahb_dmac)
- Software control of source data pre-fetch when destination is flow controller

1.4.7 Interrupts

- Combined and separate interrupt requests
- Interrupt generation on:
  - DMA transfer (multi-block) completion
  - Block transfer completion
  - Single and burst transaction completion
  - Error condition
- Support of interrupt enabling and masking

1.4.8 Low Power Mode

- Global Clock Gating
- Channel Specific Clock Gating

1.5 Standards Compliance

The DW_ahb_dmac component conforms to the *AMBA Specification, Revision 2.0* from ARM. Readers are assumed to be familiar with this specification.

1.6 Verification Environment Overview

The DW_ahb_dmac includes an extensive verification environment, which sets up and invokes your selected simulation tool to execute tests that verify the functionality of the configured component. You can then analyze the results of the simulation.

"Verification" on page 369 discusses the specific procedures for verifying the DW_ahb_dmac.

1.7 Licenses

Before you begin using the DW_ahb_dmac, you must have a valid license. For more information, see "Licenses" in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

1.8 Where To Go From Here

At this point, you may want to get started working with the DW_ahb_dmac component within a subsystem or by itself. Synopsys provides several tools within its coreTools suite of products for the purposes of configuration, synthesis, and verification of single or multiple synthesizable IP components—coreConsultant and coreAssembler. For information on the different coreTools, see *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_ahb_dmac component, see "Overview of the coreConsultant Configuration and Integration Process" in *DesignWare Synthesizable Components for AMBA 2 User Guide*.

For more information about implementing your DW_ahb_dmac component within a DesignWare subsystem using coreAssembler, see "Overview of the coreAssembler Configuration and Integration Process" in *DesignWare Synthesizable Components for AMBA 2 User Guide*.

2
Functional Description

This chapter describes the functional details of the DW_ahb_dmac component.

There is an option to configure AHB Lite, which is the DesignWare implementation of AMBA 2.0 AHB-Lite. The DesignWare AHB Lite configuration does not include the following:

- Requesting/granting protocols to the arbiter and split/retry responses from the slaves; all slaves are made non-split capable
- No arbiter as the signals associated with the component are not used: hbusreq and hgrant
- No write data, address, or control multiplexers
- Pause mode not enabled
- Default master number changed to 1
- Number of masters is changed to 1

You use the DMAH_M1_AHB_LITE parameter to configure the DW_ahb_dmac for an AHB Lite configuration.

For more information about AHB Lite, refer to the *DesignWare DW_ahb Databook*.

2.1 Setup/Operation of DW_ahb_dmac Transfers

"Programming a Channel" on page 342 describes how to program the DW_ahb_dmac in order to perform DMA transfers. This section discusses how a single block transfer, made up of transactions, is actually performed. The relevant settings of the DW_ahb_dmac are also discussed here.

2.2 Block Flow Controller and Transfer Type

The device that controls the length of a block is known as the flow controller. Either the DW_ahb_dmac, the source peripheral, or the destination peripheral must be assigned as the flow controller.

- If the block size is known prior to when the channel is enabled, then the DW_ahb_dmac should be programmed as the flow controller. The block size should be programmed into the CTLx.BLOCK_TS field.
- If the block size is unknown when the DW_ahb_dmac channel is enabled, either the source or destination peripheral must be the flow controller.

The CTL*x*.TT_FC field indicates the transfer type and flow controller for that channel. Table 2-1 lists valid transfer types and flow controller combinations.

Table 2-1 Transfer Types and Flow Control Combinations (CTLx.TT_FC Field Decoding)

CTLx.TT_FC Field	Transfer Type	Flow Controller
000	Memory to Memory	DW_ahb_dmac
001	Memory to Peripheral	DW_ahb_dmac
010	Peripheral to Memory	DW_ahb_dmac
011	Peripheral to Peripheral	DW_ahb_dmac
100	Peripheral to Memory	Peripheral
101	Peripheral to Peripheral	Source Peripheral
110	Memory to Peripheral	Peripheral
111	Peripheral to Peripheral	Destination Peripheral

As an example, the DW_ahb_dmac can be programmed as the flow controller when a DMA block must be transferred from a receive DW_apb_ssi peripheral to memory. In a block transfer, software programs the DW_apb_ssi register – CTRLR1.NDF – with the number of source data items minus 1. Software then programs the CTL*x*.BLOCK_TS register with the same value and programs the DW_ahb_dmac as the flow controller.

The DW_apb_ssi has no built-in intelligence to signal block completion to the DW_ahb_dmac; this is not required in this case because software knows the block size prior to enabling the channel.

As another example, a peripheral can be a block flow controller when a DMA block must be transferred from an Ethernet controller to memory. In this case, the size of an ethernet packet may not be known prior to enabling the DW_ahb_dmac channel. Therefore, the ethernet controller needs built-in intelligence to indicate to the DW_ahb_dmac when a block transfer has completed.

2.3 Handshaking Interface

Handshaking interfaces are used at the transaction level to control the flow of single or burst transactions. The operation of the handshaking interface is different and depends on whether the peripheral or the DW_ahb_dmac is the flow controller.

The peripheral uses the handshaking interface to indicate to the DW_ahb_dmac that it is ready to transfer or accept data over the AHB bus.

A non-memory peripheral can request a DMA transfer through the DW_ahb_dmac using one of two types of handshaking interfaces:

- Hardware
- Software

Software selects between the hardware or software handshaking interface on a per-channel basis. Software handshaking is accomplished through memory-mapped registers, while hardware handshaking is accomplished using a dedicated handshaking interface.

Throughout the remainder of this document, references to both source and destination hardware handshaking interfaces assume an active-high interface (refer to CFG*x*.SRC(DST)_HS_POL bits in the Channel Configuration register, "CFGx"). When active-low handshaking interfaces are used, then the active level and edge are reversed from that of an active-high interface.

The type of handshaking interface depends on whether the peripheral is a flow controller or not.

Source and destination peripherals can independently select the handshaking interface type; that is, hardware or software handshaking. For more information, refer to the CFG*x*.HS_SEL_SRC and CFG*x*.HS_SEL_DST parameters in the CFGx register.

2.4 Basic Interface Definitions

In this chapter and the following equations, references to CTL*x*.SRC_MSIZE, CTL*x*.DEST_MSIZE, CTL*x*.SRC_TR_WIDTH, and CTL*x*.DST_TR_WIDTH refer to the decoded values of the parameters; for example, CTL*x*.SRC_MSIZE = 3'b001 decodes to 4, and CTL*x*.SRC_TR_WIDTH = 3'b010 decodes to 32 bits.

The following definitions are used in this chapter:

- Source single transaction size in bytes

  src_single_size_bytes = CTL*x*.SRC_TR_WIDTH/8 **(1)**

- Source burst transaction size in bytes

  src_burst_size_bytes = CTL*x*.SRC_MSIZE * *src_single_size_bytes* **(2)**

- Destination single transaction size in bytes

  dst_single_size_bytes = CTL*x*.DST_TR_WIDTH/8 **(3)**

- Destination burst transaction size in bytes

  dst_burst_size_bytes = CTL*x*.DEST_MSIZE * *dst_single_size_bytes* **(4)**

- Block size in bytes:
  - DW_ahb_dmac is flow controller – With the DW_ahb_dmac as the flow controller, the processor programs the DW_ahb_dmac with the number of data items (block size) of source transfer width (CTL*x*.SRC_TR_WIDTH) to be transferred by the DW_ahb_dmac in a block transfer; this is programmed into the CTL*x*.BLOCK_TS field. Therefore, the total number of bytes to be transferred in a block is:

    blk_size_bytes_dma = CTL*x*.BLOCK_TS * *src_single_size_bytes* **(5)**

- Source peripheral is block flow controller

  blk_size_bytes_src = (Number of source burst transactions in block * *src_burst_size_bytes*) + (Number of source single transactions in block * *src_single_size_bytes*)**(6)**

- Destination peripheral is block flow controller

  blk_size_bytes_dst = (Number of destination burst transactions in block * *dst_burst_size_bytes*) + (Number of destination single transactions in block * *dst_single_size_bytes*)**(7)**

2.5 Memory Peripherals

Figure 1-6 shows the DMA transfer hierarchy of the DW_ahb_dmac for a memory peripheral. There is no handshaking interface with the DW_ahb_dmac, and therefore the memory peripheral can never be a flow controller. Once the channel is enabled, the transfer proceeds immediately without waiting for a transaction request.

The alternative to not having a transaction-level handshaking interface is to allow the DW_ahb_dmac to attempt AHB transfers to the peripheral once the channel is enabled. If the peripheral slave cannot accept these AHB transfers, it inserts wait states onto the bus (by de-asserting hready) until it is ready; it is not recommended that more than 16 wait states be inserted onto the bus. By using the handshaking interface, the peripheral can signal to the DW_ahb_dmac that it is ready to transmit or receive data, and then the DW_ahb_dmac can access the peripheral without the peripheral inserting wait states onto the bus.

Note If a channel is used exclusively for memory-to-memory DMA transfers – that is, no transaction-level handshaking on the source or destination side – then set DMAH_MAX_MULT_SIZE to 4 in order to achieve logic optimization.

The CTLx.SRC_MSIZE and CTLx.DEST_MSIZE are properties valid only for peripherals with a handshaking interface; they cannot be used for defining the burst length for memory peripherals.

When the peripherals are memory, the DW_ahb_dmac is always the flow controller and uses DMA transfers to move blocks; thus the CTLx.SRC_MSIZE and CTLx.DEST_MSIZE values are not used for memory peripherals. The SRC_MSIZE/DEST_MSIZE limitations are used to accommodate devices that have limited resources, such as a FIFO. Memory does not normally have limitations similar to the FIFOs.

Therefore:

- Length of burst transfers *to* memory is always equal to the number of data items available in a channel FIFO or data items required to complete the block transfer, whichever is smaller.
- Length of burst transfers *from* memory is always equal to the space available in a channel FIFO or number of data items required to complete the block transfer, whichever is smaller.

Note The length of burst transfers to or from memory can be limited by software by programming the CFGx.MAX_ABRST register field.

2.6 Software Handshaking

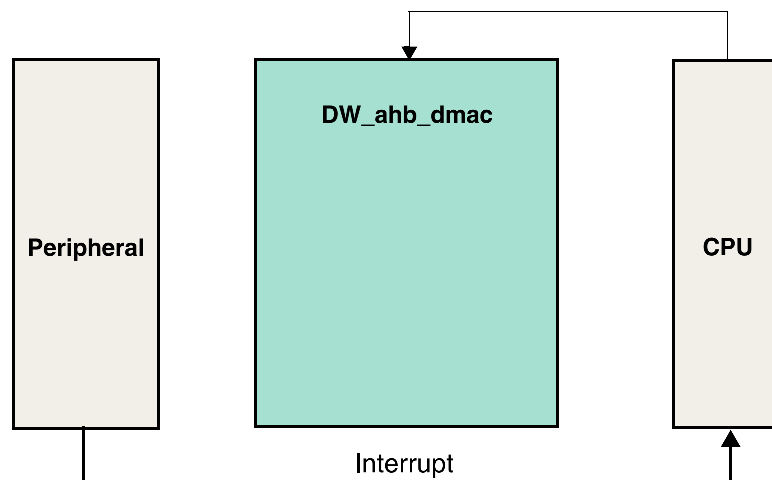
When the slave peripheral requires the DW_ahb_dmac to perform a DMA transaction, it communicates this request by sending an interrupt to the CPU or interrupt controller. The interrupt service routine then uses

the software registers detailed in "Software Handshaking Registers" to initiate and control a DMA transaction. This group of software registers is used to implement the software handshaking interface.

The HS_SEL_SRC/HS_SEL_DST bit in the CFGx channel configuration register must be set to enable software handshaking. For examples on how to use the software handshaking interface, refer to "Example 9" on page 78, "Example 10" on page 80, and "Example 11" on page 83.

Figure 2-1 Software Controlled DMA Transfers

- Program and enable channel through "Channel Registers"
- After interrupt, initiate and control DMA transaction between peripherals and DW_ahb_dmac through "Software Handshaking Registers".



The software handshaking registers are:

- ReqSrcReg– source software transaction request
- ReqDstReg – destination software transaction request
- SglReqSrcReg – single source transaction request
- SglReqDstReg – single destination transaction request
- LstSrcReg – last source transaction request
- LstDstReg – last destination transaction request

For details on how the software handshaking flow works, refer to "Software Handshaking – Peripheral Is Not Flow Controller" on page 46 and "Software Handshaking – Peripheral Is Flow Controller" on page 51.

2.7 Handshaking Interface – Peripheral Is Not Flow Controller

When the peripheral is not the flow controller, the DW_ahb_dmac tries to efficiently transfer the data using as little of the bus bandwidth as possible. Generally, the DW_ahb_dmac tries to transfer the data using burst transactions and, where possible, fill or empty the channel FIFO in single bursts – provided that the software has not limited the burst length; refer to "Example 3". The DW_ahb_dmac can also lock the arbitration for the master bus interface so that a channel is permanently granted the master bus interface.

Additionally, the DW_ahb_dmac can assert the AMBA hlock signal to lock the DW_ahb system arbiter. For more information, refer to "Locked DMA Transfers" on page 96.

Before describing the handshaking interface operation when the peripheral is not the flow controller, the following sections define the terms "Single Transaction Region" and "Early-Terminated Burst Transaction."

2.7.1 Single Transaction Region

There are cases where a DMA block transfer cannot complete using only burst transactions. Typically this occurs when the block size is not a multiple of the burst transaction length; for more information, refer to Example 4 and Example 5. In these cases, the block transfer uses burst transactions up to the point where the amount of data left to complete the block is less than the amount of data in a burst transaction. At this point, the DW_ahb_dmac samples the "single" status flag and completes the block transfer using single transactions; again refer to Example 4 and Example 5.

The peripheral asserts a single status flag to indicate to the DW_ahb_dmac that there is enough data or space to complete a single transaction from or to the source/destination peripheral.

Note

For hardware handshaking, the single status flag is a signal on the hardware handshaking interface; refer to "Hardware Handshaking – Peripheral Is Not Flow Controller" on page 40. For software handshaking, the single status flag is one of the software handshaking interface registers; refer to "Software Handshaking – Peripheral Is Not Flow Controller" on page 46.

The Single Transaction Region is the time interval where the DW_ahb_dmac uses single transactions to complete the block transfer; burst transactions are exclusively used outside this region.

Note

Burst transactions can also be used in this region; for more information, refer to ""Early-Terminated Burst Transaction" on page 39."

The Single Transaction Region applies to only a peripheral that is *not* the flow controller. The precise definition of when this region is entered is dependent on what acts as the flow controller:

- The DW_ahb_dmac is the flow controller – The source peripheral enters the Single Transaction Region when the number of bytes left to complete in the source block transfer is less than *src_burst_size_bytes*. If:

  blk_size_bytes/src_burst_size_bytes = *integer* **(8)**

  then the source never enters this region, and the source block uses only burst transactions.

  The destination peripheral enters the Single Transaction Region when the number of bytes left to complete in the destination block transfer is less than *dst_burst_size_bytes*. If:

  blk_size_bytes/dst_burst_size_bytes = *integer* **(9)**

  then the destination never enters this region, and the destination block uses only burst transactions.

The conditions mentioned earlier cause a peripheral to enter the Single Transaction Region. When the peripheral is outside the Single Transaction Region, then the DW_ahb_dmac responds to only burst transaction requests. Whether the peripheral knows that it is in the Single Transaction Region or not, it must always generate burst requests outside the Single Transaction Region, or the DMA block transfer stalls. Once in the Single Transaction Region, the DW_ahb_dmac can complete the block transfer using single transactions.

- Either the source or destination peripheral is the flow controller – The destination or source peripheral enters the Single Transaction Region when the flow control peripheral – that is, the source or destination – signals the last transaction in the block *and* when the amount of data left to be transferred in the destination/source block is less than that which is specified by *dst_burst_size_bytes*/*src_burst_size_bytes*.

2.7.2 Early-Terminated Burst Transaction

When a source or destination peripheral is in the Single Transaction Region, a burst transaction can still be requested. However, *src_burst_size_bytes*/ *dst_burst_size_bytes* is greater than the number of bytes left to complete in the source/destination block transfer at the time that the burst transaction is triggered. In this case, the burst transaction is started and “early-terminated” at block completion without transferring the programmed amount of data – that is, *src_burst_size_bytes* or *dst_burst_size_bytes* – but only the amount required to complete the block transfer. An Early-Terminated Burst Transaction occurs between the DW_ahb_dmac and the peripheral only when the peripheral is not the flow controller.

The DW_ahb_dmac never terminates defined-length bursts early.

- If the DW_ahb_dmac is configured with DMAH_INCR_BURSTS = 0 and it detects a burst request when in the Single Transaction Region, the DW_ahb_dmac only issues SINGLE type transfers to complete the burst.
- If DMAH_INCR_BURSTS = 1 and the DW_ahb_dmac detects a burst when in the Single Transaction Region, the DW_ahb_dmac issues an undefined length burst (INCR) of length only large enough to complete the block transfer. It does not transfer all programmed *src_burst_size_bytes* or *dst_burst_size_bytes*.

2.7.3 Hardware Handshaking – Peripheral Is Not Flow Controller

Figure 2-2 illustrates the hardware handshaking interface between a peripheral – whether a destination or source – and the DW_ahb_dmac when the peripheral is not the flow controller.

Figure 2-2 Hardware Handshaking Interface

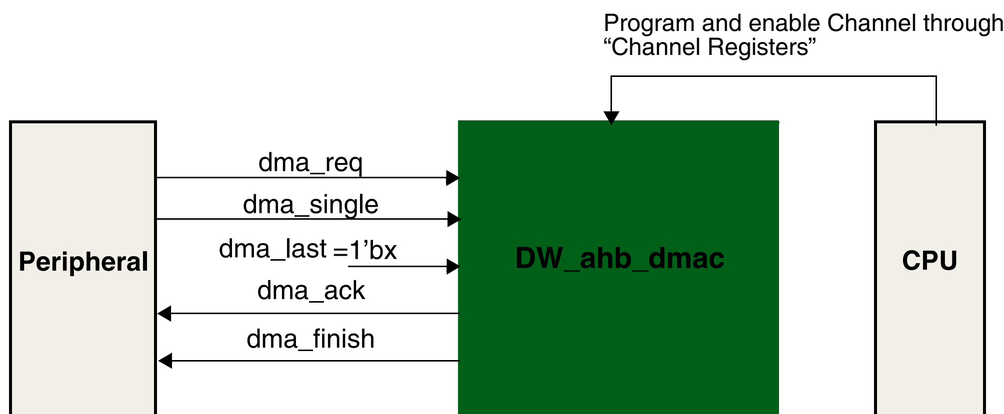


Table 2-2 describes the hardware handshaking signals in the case where the peripheral is not the flow controller; that is, where either the DW_ahb_dmac or the other peripheral is the flow controller. Signal polarity can be programmed using the CFGx.SRC_HS_POL and CFGx.DST_HS_POL fields.

Table 2-2 Hardware Handshaking Interface

Signal	Direction	Description
dma_ack	Output	**DW_ahb_dmac acknowledge signal to peripheral**. The dma_ack signal is asserted after the data phase of the last AHB transfer in the current transaction – single or burst – to the peripheral that has completed. For a single transaction, dma_ack remains asserted until the peripheral de-asserts dma_single; dma_ack is de-asserted one hclk cycle later. For a burst transaction, dma_ack remains asserted until the peripheral de-asserts dma_req; dma_ack is de-asserted one hclk cycle later.
dma_finish	Output	**DW_ahb_dmac asserts dma_finish to signal block completion**. This has the same timing as dma_ack and forms a handshaking loop with dma_req if the last transaction in the block is a burst transaction, or with dma_single if the last transaction in the block is a single transaction. There is an exception to the timing definition mentioned earlier when dma_finish interfaces with a source peripheral when the destination peripheral is the flow controller; for details, refer to "Example 7" Case 1b.
dma_last	Input	Since the peripheral is not the flow controller, dma_last is not sampled by the DW_ahb_dmac and this signal is ignored**.**

Signal	Direction	Description
dma_req	Input	**Burst transaction request from peripheral**. The DW_ahb_dmac always interprets the dma_req signal as a burst transaction request, regardless of the level of dma_single. This is a level-sensitive signal; once asserted by the peripheral, dma_req must remain asserted until the DW_ahb_dmac asserts dma_ack. Upon receiving the dma_ack signal from the DW_ahb_dmac to indicate the burst transaction is complete, the peripheral should de-assert the burst request signal, dma_req. Once dma_req is de-asserted by the peripheral, the DW_ahb_dmac de-asserts dma_ack. If an active level on dma_req is detected in the Single Transaction Region, then the block is completed using an Early-Terminated Burst Transaction**.**
dma_single	Input	**Single transfer status**. The dma_single signal is a status signal that is asserted by a destination peripheral when it can accept at least one destination data item; otherwise it is cleared. For a source peripheral, the dma_single signal is again a status signal and is asserted by a source peripheral when it can transmit at least one source data item; otherwise it is cleared. Once asserted, dma_single must remain asserted until dma_ack is asserted, at which time the peripheral should de-assert dma_single. This signal is sampled by the DW_ahb_dmac only in the Single Transaction Region of the block transfer. Outside of this region, dma_single is ignored and all transactions are burst transactions.

When the dma_req or dma_single signal is asserted, they must remain asserted until dma_ack is asserted; otherwise, the behavior of DW_ahb_dmac is not defined and data integrity is not guaranteed.

In such scenarios, the corresponding channel can be reused by disabling the channel and then by re-enabling the channel with appropriate programming required for subsequent DMA transfers.

Figure 2-3 shows the timing diagram of a burst transaction where the peripheral clock, per_clk, equals hclk. In this example, the peripheral is outside the Single Transaction Region, and therefore the DW_ahb_dmac does not sample dma_single[0].

The handshaking loop is as follows:

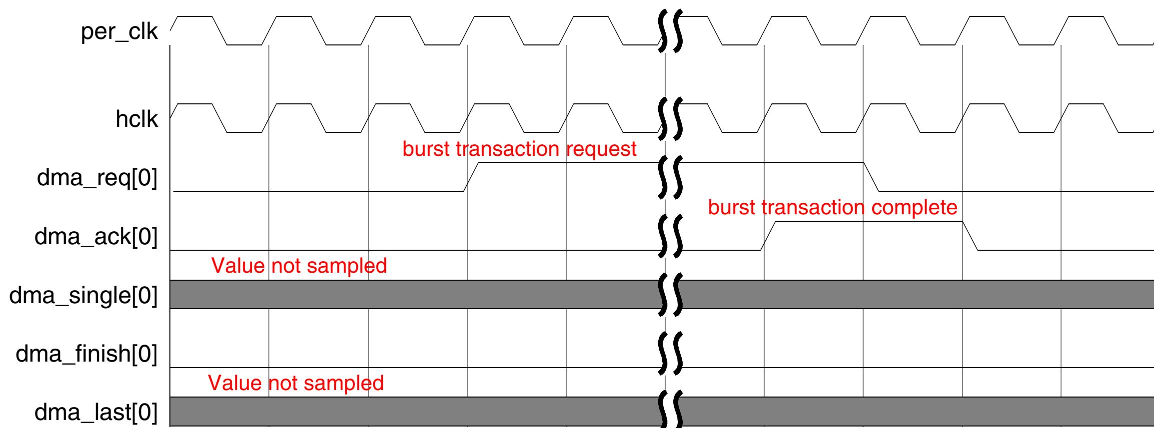
dma_req asserted by peripheral

-> dma_ack asserted by DW_ahb_dmac

-> dma_req de-asserted by peripheral

-> dma_ack de-asserted by DW_ahb_dmac.

Figure 2-3 Burst Transaction – pclk = hclk



The per_clk signal is equal to hclk if the peripheral is an AHB peripheral; it is equal to pclk if the peripheral is an APB peripheral. The burst transaction request signal, dma_req, and the single status signal, dma_single, are generated in the peripheral of per_clk and sampled by hclk in the DW_ahb_dmac. The acknowledge signal, dma_ack, is generated in the DW_ahb_dmac of hclk and sampled in the peripheral by per_clk. The handshaking mechanism between the DW_ahb_dmac and the peripheral supports quasi-synchronous clocks; that is, hclk and per_clk must be phase- aligned, and the hclk frequency must be a multiple of the per_clk frequency.

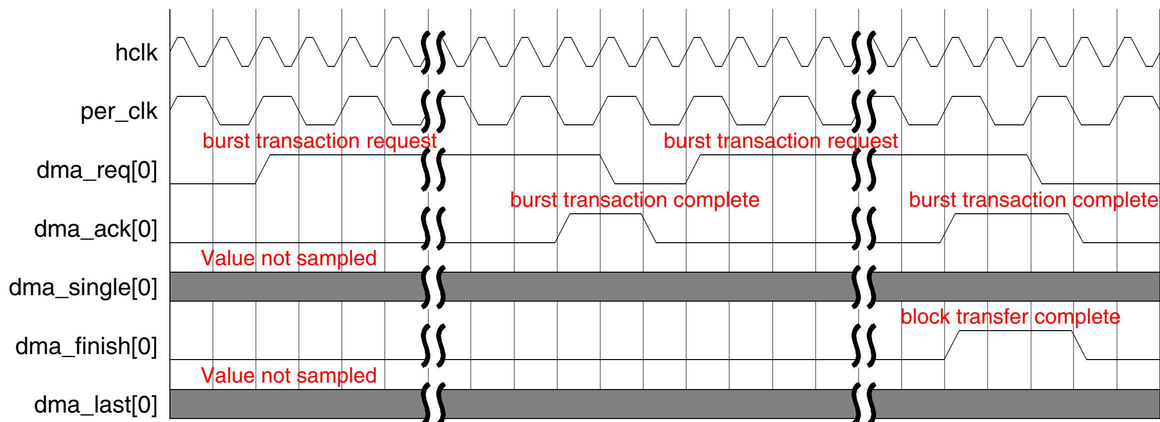
In the case where the destination peripheral is an APB peripheral and for the case of buffered writes through an APB bridge, then caution must be exercised so as not to overflow the destination peripheral FIFO. This can happen when the write is buffered in the APB bridge; that is, the write completes on the AHB before completing on the APB bus. The following scenario can cause an overflow: the DW_ahb_dmac asserts dma_ack as soon as the write transaction completes on the AHB. The APB peripheral, on sampling that the acknowledge signal is asserted, de-asserts its request signal and asserts the request signal one APB clock cycle later, as it senses that there is space in its FIFO. The issue here is that there could be space for only a single entry that the first buffered write consumes. The initiation of the second transaction may overflow the FIFO. If the write is not buffered, then the initiation of the second transaction does not occur, as the destination peripheral senses that its FIFO is full.

To avoid this, do one of the following:

1. Do not use buffered writes.
2. If using buffered writes, then ensure that dma_ack signal from the DW_ahb_dmac is delayed until the write completes to the APB peripheral. One way may be to route the dma_ack signal through the APB bridge.

Figure 2-4 shows two back-to-back burst transactions at the end of a block transfer where the hclk frequency is twice the pclk frequency; the peripheral is an APB peripheral. The second burst transaction terminates the block, and dma_finish[0] is asserted to indicate block completion.

Figure 2-4 Back-to-Back Burst Transactions – hclk = 2*per_clk



There are two things to note when designing the hardware handshaking interface:

- Once asserted, the dma_req burst request signal must remain asserted until the corresponding dma_ack signal is received, even if the condition that generates dma_req in the peripheral is False.
- The dma_req signal should be de-asserted when dma_ack is asserted, even if the condition that generates dma_req in the peripheral is True.

Figure 2-5 shows a single transaction that occurs in the Single Transaction Region. The handshaking loop is as follows:

dma_single asserted by peripheral

-> dma_ack asserted by DW_ahb_dmac

-> dma_single de-asserted by peripheral

-> dma_ack de-asserted by DW_ahb_dmac

Figure 2-5 Single Transaction

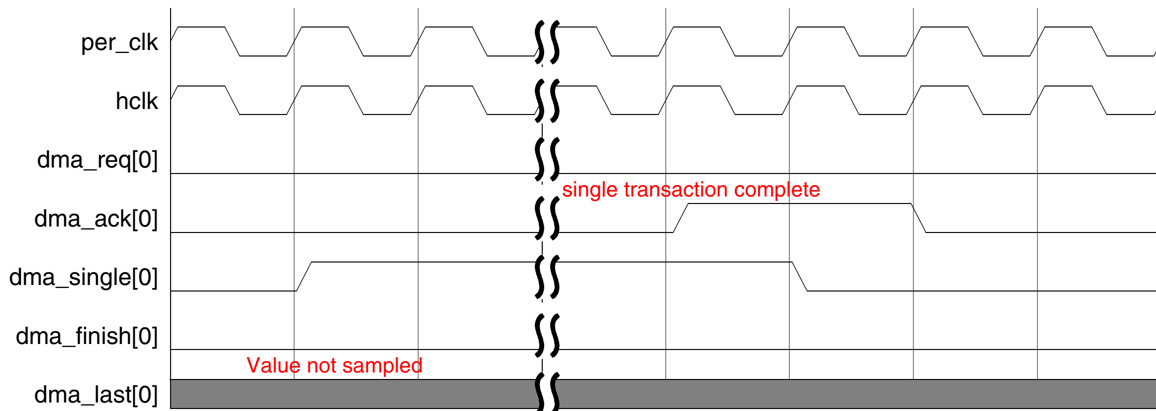
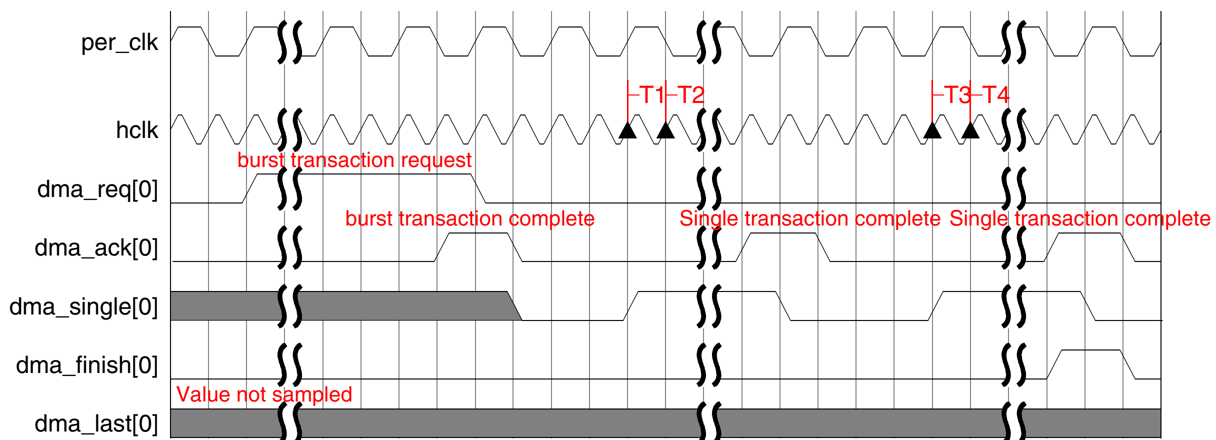


Figure 2-6 shows a burst transaction, followed by two back-to-back single transactions, where the hclk frequency is twice the per_clk frequency; handshaking interface 0 is used.

Figure 2-6 Burst Followed by Back-to-Back Single Transactions



After the first burst transaction, the peripheral enters the Single Transaction Region and the DW_ahb_dmac starts sampling dma_single[0]. On hclk edges T2 and T4, the DW_ahb_dmac samples that dma_single[0] is asserted and performs single transactions. The second single transaction terminates the block transfer; dma_finish[0] is asserted to indicate block completion.

In the Single Transaction Region, if an active level on dma_req and dma_single occur on the same cycle – or if the active level on dma_single occurs only one cycle before an active level on dma_req – then the burst transaction takes precedence over the single transaction, and the block would be completed using an

Early-Terminated Burst Transaction. If the DW_ahb_dmac samples that dma_req[0] is asserted on hclk cycles T1-T2 or T3-T4 in Figure 2-6, then the burst request takes precedence.

In Figure 2-7, an active level on dma_req[0] after time T4 takes precedence over the active level on dma_single[0] after time T3.

Figure 2-7 Early-Terminated Burst Transaction

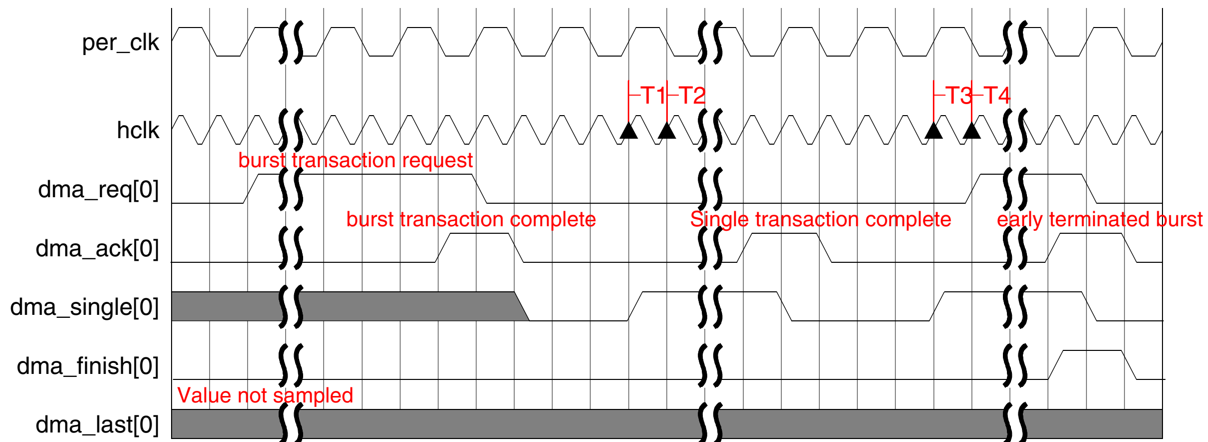
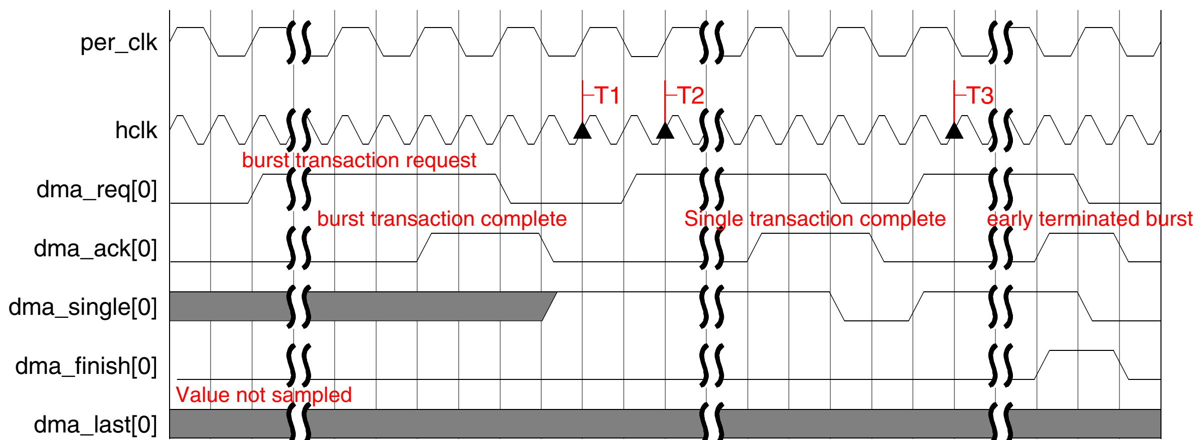


Figure 2-8 shows a burst transaction followed by a single transaction, followed by a burst transaction at the end of a block.

Figure 2-8 Burst Transaction ignored During Active Single Transaction



After the first burst transaction completes, the peripheral is in the Single Transaction Region and DW_ahb_dmac samples that dma_single[0] is asserted at T1. The dma_req[0] signal is triggered in the middle of this single transaction at time T2. This burst transaction request is ignored and is not serviced. An active edge on dma_req[0] is re-generated and sampled by DW_ahb_dmac at time T3. This burst transaction completes the block transfer using an Early-Terminated Burst Transaction.

2.7.3.1 Generating dma_req and dma_single Hardware Handshaking Signals

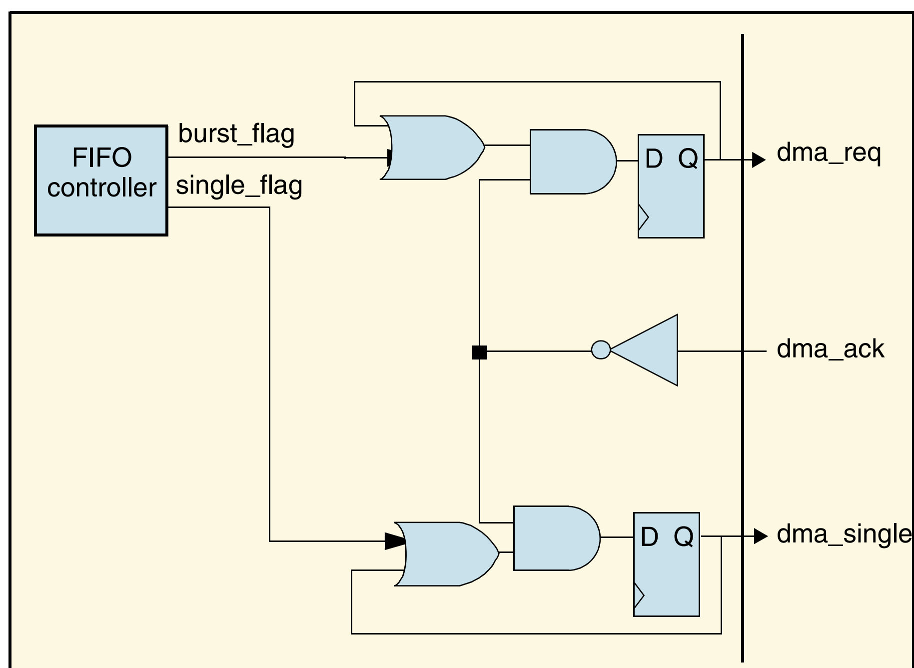
Figure 2-9 illustrates a suggested method of generating dma_req and dma_single for a source peripheral when the peripheral is not the flow controller. The single_flag signal in Figure 2-9 is asserted when the source FIFO has at least one source data item in the FIFO. The burst_flag signal in Figure 2-9 is asserted

when the source FIFO contains data items greater than or equal to some watermark-level number of data items in it.

Note Figure 2-9 shows dma_req and dma_single being de-asserted when dma_ack is asserted. It also shows how, once asserted, dma_req and dma_single remain asserted until dma_ack is asserted. The example assumes active-high handshaking.

The destination peripheral dma_req and dma_single signals can be generated in a similar fashion, but in this case the single_flag signal in Figure 2-9 is asserted when the destination FIFO has at least one free location. The burst_flag signal in Figure 2-9 is asserted when the destination FIFO contains free locations greater than or equal to some watermark-level number.

Figure 2-9 Generation of dma_req and dma_single by Source



2.7.4 Software Handshaking – Peripheral Is Not Flow Controller

When the peripheral is not the flow controller, then the last transaction registers – LstSrcReg and LstDstReg – are not used, and the values in these registers are ignored.

2.7.4.1 Operation – Peripheral Not In Single Transaction Region

Writing a 1 to the ReqSrcReg[x]/ReqDstReg[x] register is always interpreted as a burst transaction request, where x is the channel number. However, in order for a burst transaction request to start, software must write a 1 to the SglReqSrcReg[x]/ SglReqDstReg[x] register.

You can write a 1 to the SglReqSrcReg[*x*]/SglReqDstReg[*x*] and ReqSrcReg[*x*]/ReqDstReg[*x*] registers in any order, but both registers must be asserted in order to initiate a burst transaction. Upon completion of the burst transaction, the hardware clears the SglReqSrcReg[*x*]/SglReqDstReg[*x*] and ReqSrcReg[*x*]/ReqDstReg[*x*] registers.

2.7.4.2 Operation – Peripheral In Single Transaction Region

Writing a 1 to the SglReqSrcReg/SglReqDstReg initiates a single transaction. Upon completion of the single transaction, both the SglReqSrcReg/SglReqDstReg and ReqSrcReg/ReqDstReg bits are cleared by hardware. Therefore, writing a 1 to the ReqSrcReg/ReqDstReg is ignored while a single transaction has been initiated, and the requested burst transaction is not serviced.

Again, writing a 1 to the ReqSrcReg/ReqDstReg register is always a burst transaction request. However, in order for a burst transaction request to start, the corresponding channel bit in the SglReqSrcReg/SglReqDstReg must be asserted. Therefore, to ensure that a burst transaction is serviced in this region, you must write a 1 to the ReqSrcReg/ReqDstReg before writing a 1 to the SglReqSrcReg/SglReqDstReg register. If the programming order is reversed, a single transaction is started instead of a burst transaction. The hardware clears both the ReqSrcReg/ReqDstReg and the SglReqSrcReg/SglReqDstReg registers after the burst transaction request completes. When a burst transaction is initiated in the Single Transaction Region, then the block completes using an Early-Terminated Burst Transaction.

Software can poll the relevant channel bit in the SglReqSrcReg/SglReqDstReg and ReqSrcReg/ReqDstReg registers. When both are 0, then either the requested burst or single transaction has completed. Alternatively, the IntSrcTran or IntDstTran interrupts can be enabled and unmasked in order to generate an interrupt when the requested source or destination transaction has completed.

Note The transaction-complete interrupts are triggered when both single and burst transactions are complete. The same transaction-complete interrupt is used for both single and burst transactions.

2.7.5 Single Transactions – Peripheral Is Not Flow Controller

When the source peripheral is not the flow controller, the source peripheral can hardcode dma_single to an inactive level (hardware handshaking), or software will never need to initiate single transactions from the source (software handshaking). This can happen if either of the following is true:

- Block size is a multiple of the burst transaction length.
  - If DW_ahb_dmac is the flow controller

    blk_size_bytes_dma/src_burst_size_bytes = *integer*
  - If the destination peripheral is the flow controller

    blk_size_bytes_dst/src_burst_size_bytes = *integer*
- Block size is not a multiple of the burst transaction length, but the peripheral can dynamically adjust the watermark level that triggers a burst request in order to enable block completion.

When the destination peripheral is not a flow controller, then the destination peripheral may hardcode dma_single to an inactive level (hardware handshaking), or software will never need to initiate single transactions to the destination (software handshaking). This can happen when any of the following are true:

- Block size is a multiple of the burst transaction length.
  - If DW_ahb_dmac is the flow controller

    block_size_bytes_dma/dst_burst_size_bytes = *integer*

- If the source peripheral is flow controller

  block_size_bytes_src/dst_burst_size_bytes = *integer*

- The destination peripheral can dynamically adjust the watermark level upwards so that a burst request is triggered in order to enable a destination block completion.
- It is guaranteed that data at some point will be extracted from the destination FIFO in the "Single transaction region" in order to trigger a burst transaction.

Note The destination peripheral requires data to be extracted in order to empty the destination FIFO below a watermark level for triggering a destination burst request. If it is guaranteed that data at some point will be extracted from the destination FIFO in the Single Transaction Region in order to trigger a dma_req, then in this case, the dma_single signal from the destination can be tied to an inactive level, and the destination block completes with an Early-Terminated Burst Transaction.

If none of the above are true, then a series of burst transactions followed by single transactions is needed to complete the source/destination block transfer; for more information, refer to "Example 4" and "Example 5".

2.8 Handshaking Interface – Peripheral Is Flow Controller

When the peripheral is the flow controller, it controls the length of the block and must communicate to the DW_ahb_dmac when the block transfer is complete. The peripheral does this by telling the DW_ahb_dmac that the current transaction – burst or single – is the last transaction in the block. When the peripheral is the flow controller and the block size is not a multiple of the CTL*x*.SRC_MSIZE/CTL*x*.DEST_MSIZE, then the peripheral must use single transactions to complete a block transfer.

Note Since the peripheral can terminate the block on a single transaction, there is no notion of a Single Transaction Region such as there is when the peripheral is not the flow controller.

When the peripheral is the flow controller, it indicates directly to DW_ahb_dmac which type of transaction – single or burst – to perform. Where possible, the DW_ahb_dmac uses the maximum possible burst length. It can also lock the arbitration for the master bus so that a channel is permanently granted the master bus interface. The DW_ahb_dmac can also assert the hlock signal to lock the DW_ahb system arbiter. For more information, refer to "Locked DMA Transfers" on page 96.

2.8.1 Hardware Handshaking – Peripheral Is Flow Controller

Figure 2-10 shows the hardware handshaking interface between a destination or source peripheral and the DW_ahb_dmac when the peripheral is the flow controller.

Figure 2-10 Hardware Handshaking Interface

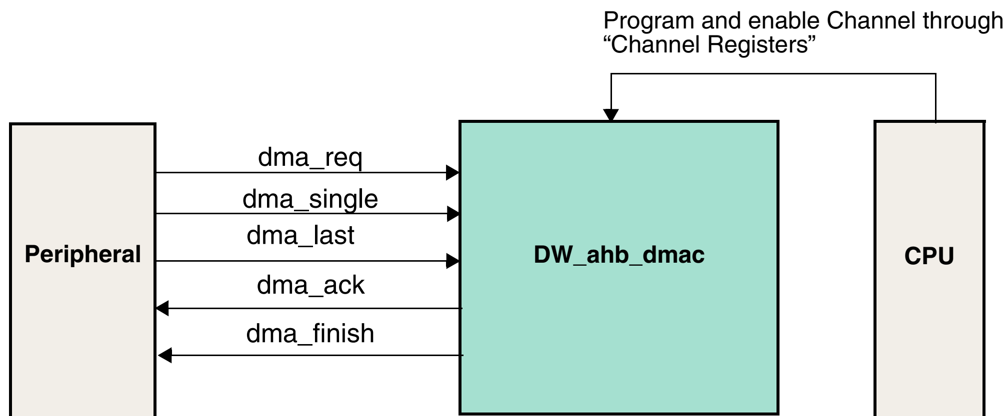


Table 2-3 describes the operation of the hardware handshaking interface signals when the peripheral is the flow controller; timing diagrams are illustrated in Figure 2-11 and Figure 2-12.

Table 2-3 Hardware Handshaking Interface

Signal	Direction	Description
dma_ack	Output	**DW_ahb_dmac acknowledge signal to the peripheral**. This is asserted after the data phase of the last AHB transfer in the current transaction (single or burst) to the peripheral has completed. It forms a handshaking loop with dma_req and remains asserted until the peripheral de-asserts dma_req (de-asserted one hclk cycle later).
dma_finish	Output	**DW_ahb_dmac block transfer complete signal**. The DW_ahb_dmac asserts dma_finish in order to signal block completion. This uses the same timing as dma_ack and forms a handshaking loop with dma_req.
dma_last	Input	**Last transaction in block**. When the peripheral is the flow controller, it asserts dma_last on the same cycle as dma_req is asserted in order to signal that this transaction request is the last in the block; the block transfer is complete after this transaction is complete. If dma_single is high in the same cycle, then the last transaction is a single transaction. If dma_single is low in the same cycle, then the last transaction is a burst transaction.
dma_req	Input	**Transaction request from peripheral**. An active level on dma_req initiates a transaction request. The type of transaction – single or burst – is qualified by dma_single. Once dma_req is asserted, it must remain asserted until dma_ack is asserted. When the peripheral that is driving dma_req determines that dma_ack is asserted, it must de-assert dma_req.

Signal	Direction	Description
dma_single	Input	**Single or burst transaction request**. If dma_single is de-asserted in the same clock cycle as a rising edge on dma_req, a burst transaction is requested by the peripheral. If asserted, the peripheral requests a single transaction.

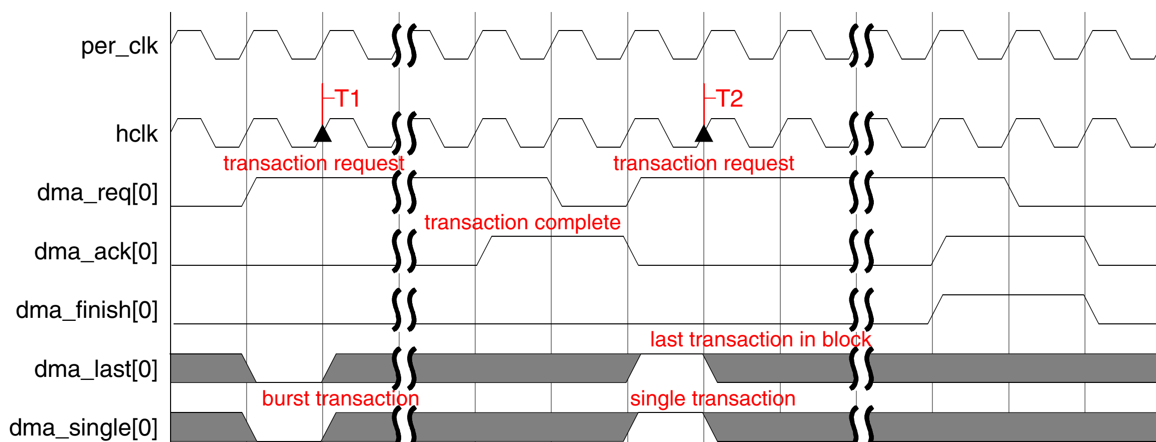
Note When the dma_req or dma_single signal is asserted, they must remain asserted until dma_ack is asserted; otherwise, the behavior of DW_ahb_dmac is not defined and data integrity is not guaranteed.

In such scenarios, the corresponding channel can be reused by disabling the channel and then by re-enabling the channel with appropriate programming required for subsequent DMA transfers.

The following timing diagrams assume that handshaking interface 0 is active-high.

Figure 2-11 shows a burst transaction followed by a single transaction, where the single transaction is the last in the block. On clock edge T1, DW_ahb_dmac samples that dma_req[0] is asserted, dma_single[0] is de-asserted, and dma_last[0] is de-asserted. This is a request for a burst transaction, which is not the last transaction in the block.

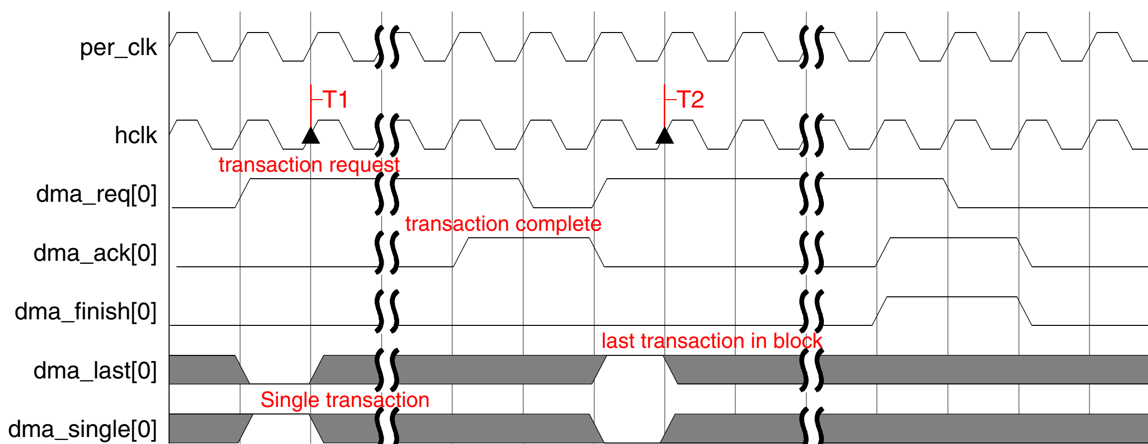
Figure 2-11 Burst Transaction Followed by Single Transaction that Terminates Block



On clock edge T2, the DW_ahb_dmac samples that dma_req[0], dma_single[0], and dma_last[0] are all asserted. This is a request for a single transaction, which is the last transaction in the block. The dma_last[0] and dma_single[0] signals need only be valid on the same clock cycle that dma_req is generated.

Similarly, Figure 2-12 shows a single transaction followed by a burst transaction, where the burst transaction is the last transaction in the block.

Figure 2-12 Single Transaction Followed by Burst Transaction that Terminates Block



2.8.2 Software Handshaking – Peripheral Is Flow Controller

Writing a 1 to the Source/Destination Software Transaction Request initiates a transaction; refer to "ReqSrcReg" and "ReqDstReg", respectively. The type of transaction – single or burst – depends on the state of the corresponding channel bit in the Single Source/Destination Transaction Request register; refer to "SglReqSrcReg" or "SglReqDstReg", respectively.

If SglReqSrcReg[n]/SglReqDstReg[n] = 1 when a 1 is written to the ReqSrcReg[n]/ReqDstReg[n] register, this means that software is requesting a single transaction on channel *n*, or a burst transaction otherwise.

The request is the last in the block if the corresponding channel bit in the Last Source/Destination Request register is asserted; refer to "LstSrcReg" and "LstDstReg", respectively.

If LstSrcReg[n]/LstDstReg[n] = 1 when a 1 is written to the ReqSrcReg[n]/ReqDstReg[n] register, this means that software is requesting that this transaction is the last transaction in the block. The SglReqSrcReg/SglReqDstReg and LstSrcReg/LstDstReg registers must be written to before the ReqSrcReg/ReqDstRegregisters.

On completion of the transaction – single or burst – the relevant channel bit in the ReqSrcReg/ReqDstRegregister is cleared by hardware. Software can therefore poll this bit in order to determine when the requested transaction has completed. Alternatively, the IntSrcTran or IntDstTran interrupts can be enabled and unmasked in order to generate an interrupt when the requested transaction – single or burst – has completed.

When the peripheral is the flow controller and the block size is not a multiple of the CTL*x*.SRC_MSIZE/CTL*x*.DEST_MSIZE, then software must use single transactions to complete the block transfer.

2.8.3 Single Transactions – Peripheral is Flow Controller

When the source peripheral is the flow controller, then it can hardcode dma_single to an inactive level (hardware handshaking), or software will never need to initiate single transactions from the source (software handshaking). This occurs when:

$$block_size_bytes_src / src_burst_size_bytes = integer \quad (10)$$

When the destination peripheral is the flow controller, then the destination peripheral can hardcode dma_single to an inactive level (hardware handshaking), or software will never need to initiate single transactions to the destination (software handshaking) when:

$$block_size_bytes_dst / dst_burst_size_bytes = integer \quad (11)$$

2.9 Setting Up Transfers

Transfers are set up by programming fields of the CTLx and CFGx registers for that channel. As shown in Figure 1-5, a single block is made up of numerous transactions – single and burst – which are in turn composed of AHB transfers. A peripheral requests a transaction through the handshaking interface to the DW_ahb_dmac; for more information, refer to "Handshaking Interface" on page 34. The operation of the handshaking interface is different and depends on what is acting as the flow controller.

Table 2-4 lists the parameters that are investigated in the following examples. The effects of these parameters on the flow of the block transfer are highlighted. In addition to the software parameters, it includes the channel FIFO depth, DMAH_CH*x*_FIFO_DEPTH, which is configurable only in coreConsultant.

For definitions of the register parameters, refer to "Register Descriptions" on page 161; for definitions of DW_ahb_dmac parameters in coreConsultant, refer to"Parameter Descriptions" on page 117.

Table 2-4 Parameters Used in Transfer Examples

Parameter	Description
DMAH_CH*x*_FIFO_DEPTH	Channel *x* FIFO depth in bytes
CTL*x*.TT_FC	Transfer type and flow control
CTL*x*.BLOCK_TS	Block transfer size
CTL*x*.SRC_TR_WIDTH	Source transfer width
CTL*x*.DST_TR_WIDTH	Destination transfer width
CTL*x*.SRC_MSIZE	Source burst transaction length
CTL*x*.DEST_MSIZE	Destination burst transaction length
CFG*x*.MAX_ABRST	Maximum AMBA burst length
CFG*x*.FIFO_MODE	FIFO mode select
CFG*x*.FCMODE	Flow-control mode

2.9.1 Transfer Operation

The following examples show the effect of different settings of each parameter from Table 2-4 on a DMA block transfer. In all examples, it is assumed that no bursts are early-terminated by the DW_ahb system arbiter, unless otherwise stated. Example 1 through Example 8 use hardware handshaking on both the source and destination side. Example 9 through Example 11 use software handshaking on both the source and destination side.

The following is a brief description of each of the examples:

- "Example 1" – Block transfer when the DW_ahb_dmac is the flow controller.
- "Example 2" – Effect of DMAH_CH*x*_FIFO_DEPTH on a block transfer.
- "Example 3" – Effect of maximum AMBA burst length, CFG*x*.MAX_ABRST, on a block transfer.
- "Example 4" – Block transfer when the DW_ahb_dmac is the flow controller and the source peripheral enters the Single Transaction Region.
- "Example 5" – Block transfer when the DW_ahb_dmac is the flow controller and the destination peripheral enters the Single Transaction Region. Also demonstrates channel FIFO flushing to the destination peripheral at the end of a block transfer.
- "Example 6" – Effect of CFG*x*.FIFO_MODE on a block transfer.
- "Example 7" – Block transfer when the destination peripheral is the flow controller and data pre-fetching from the source is enabled; CFG*x*.FCMODE = 0.
- "Example 8" – Block transfer when the destination peripheral is the flow controller and data pre-fetching from the source is disabled; CFG*x*.FCMODE = 1.
- "Example 9" – Block transfer when the DW_ahb_dmac is the flow controller and software handshaking is used on both the source and destination side.
- "Example 10" – Block transfer when the DW_ahb_dmac is the flow controller and software handshaking is used on both the source and destination side; the source enters the Single Transaction Region.
- "Example 11" – Block transfer when the source peripheral is the flow controller and software handshaking is used on both the source and destination side; the destination peripheral enters the Single Transaction Region.

The DW_ahb_dmac is programmed with the number of data items that are to be transferred for each burst transaction request, CTL*x*.SRC_MSIZE /CTL*x*.DEST_MSIZE (see Table 2-5). Similarly, the width of each data item in the transaction is set by the CTL*x*.SRC_TR_WIDTH and CTL*x*.DST_TR_WIDTH fields (see Table 2-6).

Table 2-5 shows the number of data items that can be transferred for different CTLx.SRC_MSIZE and CTLx.DEST_MSIZE values.

Table 2-5 CTLx.SRC_MSIZE and DEST_MSIZE Decoding

CTLx.SRC_MSIZE / CTLx.DEST_MSIZE	Number of data items to be transferred (of width CTLx.SRC_TR_WIDTH or CTLx.DST_TR_WIDTH)
000	1
001	4
010	8
011	16
100	32
101	64
110	128
111	256

Table 2-6 lists the block transfer size for different CTLx.SRC_TR_WIDTH and CTLx.DST_TR_WIDTH values.

Table 2-6 CTLx.SRC_TR_WIDTH and CTLx.DST_TR_WIDTH Decoding

CTLx.SRC_TR_WIDTH / CTLx.DST_TR_WIDTH	Size (bits)
000	8
001	16
010	32
011	64
100	128
101	256
11x	256

2.9.1.1 Example 1

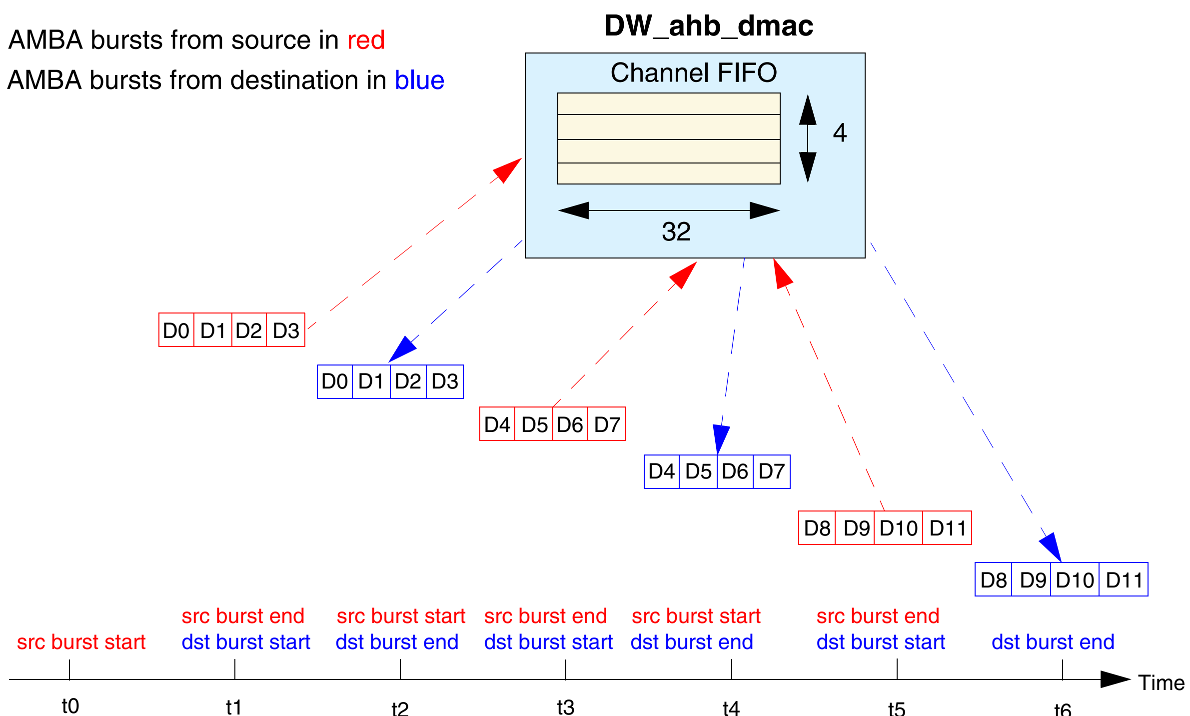
Scenario: Example block transfer when the DW_ahb_dmac is the flow controller. This example is the same for both software and hardware handshaking interfaces. Table 2-7 lists the DMA parameters for this example.

Table 2-7 Parameters in Transfer Operation – Example 1

Parameter	Description
CTL*x*.TT_FC = 3'b011	Peripheral-to-peripheral transfer with DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 12	–
CTL*x*.SRC_TR_WIDTH = 3'b010	32 bit
CTL*x*.DST_TR_WIDTH = 3'b010	32 bit
CTL*x*.SRC_MSIZE = 3'b 001	Source burst transaction length = 4
CTL*x*.DEST_MSIZE = 3'b 001	Destination burst transaction length = 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 16 bytes	–

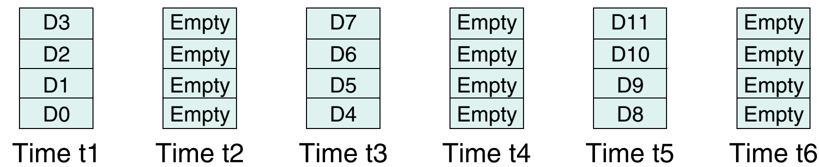
Using equation (5), a total of 48 bytes are transferred in the block; that is, *blk_size_bytes_dma* = 48. As shown in Figure 2-13, this block transfer consists of three bursts of length 4 from the source, interleaved with three bursts, again of length 4, to the destination.

Figure 2-13 Breakdown of Block Transfer



The channel FIFO is alternatively filled by a burst from the source and emptied by a burst to the destination until the block transfer has completed, as shown in Figure 2-14.

Figure 2-14 Channel FIFO Contents at Times Indicated in Figure 2-13



Burst transactions are completed in one burst. Additionally, because (8) and (9) are both true, neither the source or destination peripherals enter their Single Transaction Region at any stage throughout the DMA transfer, and the block transfer from the source and to the destination consists of burst transactions only.

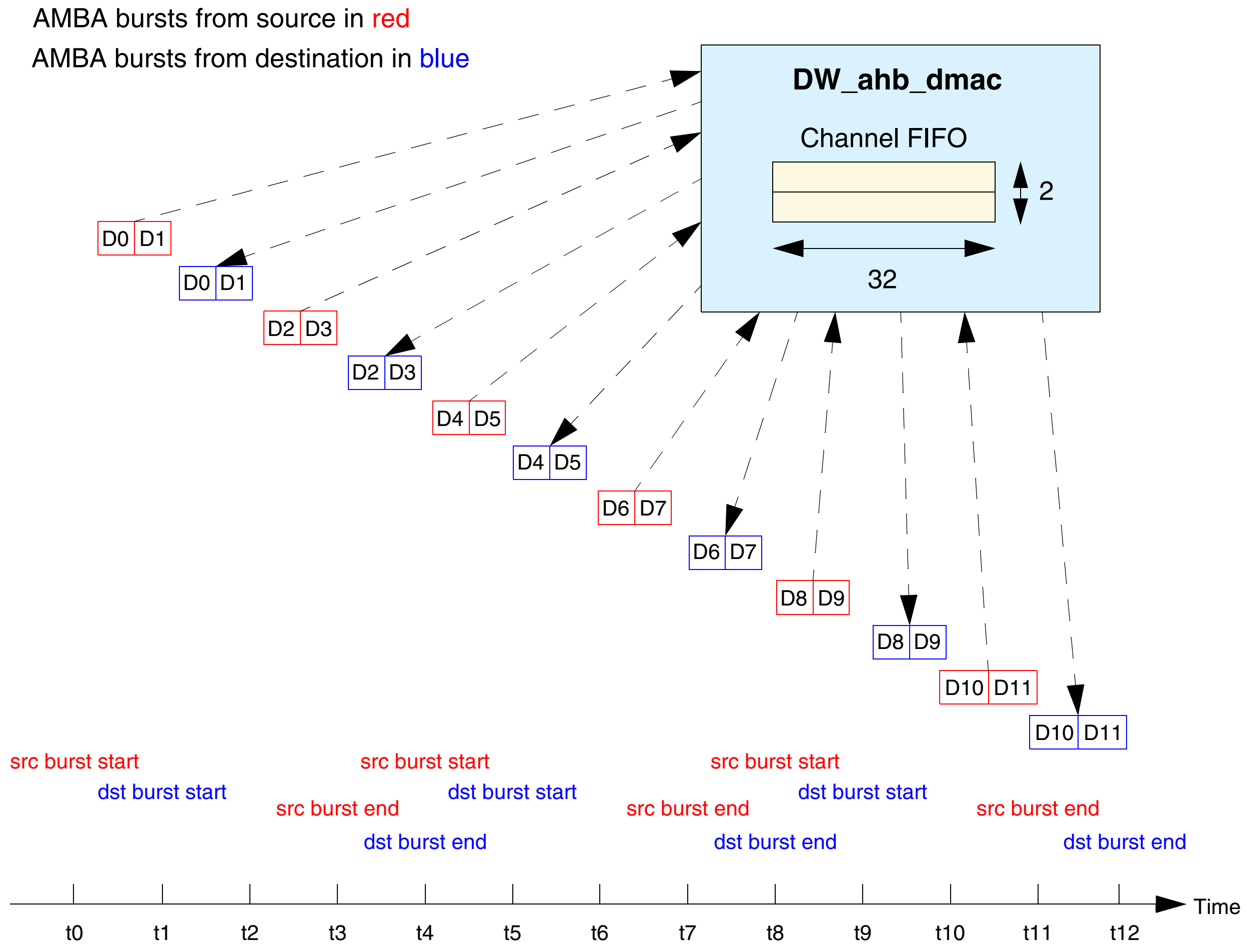
2.9.1.2 Example 2

Scenario: Effect of DMAH_CH*x*_FIFO_DEPTH on block transfers. This example is the same for both software and hardware handshaking interfaces.

In this example, the coreConsultant DMAH_CH*x*_FIFO_DEPTH parameter is changed to 8 bytes, and all other parameters are left unchanged from Example 1, Table 2-7.

Example 1 shows the source and destination burst transactions completing in a single burst. In general, a burst transaction may take multiple bursts to complete. With the DMAH_CH*x*_FIFO_DEPTH parameter set to 8 bytes instead of 16 bytes, the block transfer would look like that shown in Figure 2-15.

Figure 2-15 Breakdown of Block Transfer for DMAH_CH_FIFO_DEPTH=8



The block transfer consists of six bursts of length 2 from the source, interleaved with six bursts – again of length 2 – to the destination, as shown in Figure 2-13. The channel FIFO is alternatively filled by a burst from the source and emptied by a burst to the destination, until the block transfer has completed. In this example, a transfer of each source or destination burst transaction is made up of two bursts, each of length 2.

Therefore, Example 2 has twice the number of bursts per block than Example 1.

> **Recommendation:** To allow a burst transaction to complete in a single burst, the DW_ahb_dmac channel FIFO depth should be large enough to accept an amount of data equal to an entire burst transaction. Therefore, in order to allow both source and destination burst transactions to complete in one burst:
>
> DMAH_CH*x*_FIFO_DEPTH >= max(2**src_burst_size_bytes*, 2**dst_burst_size_bytes*) **(12)**

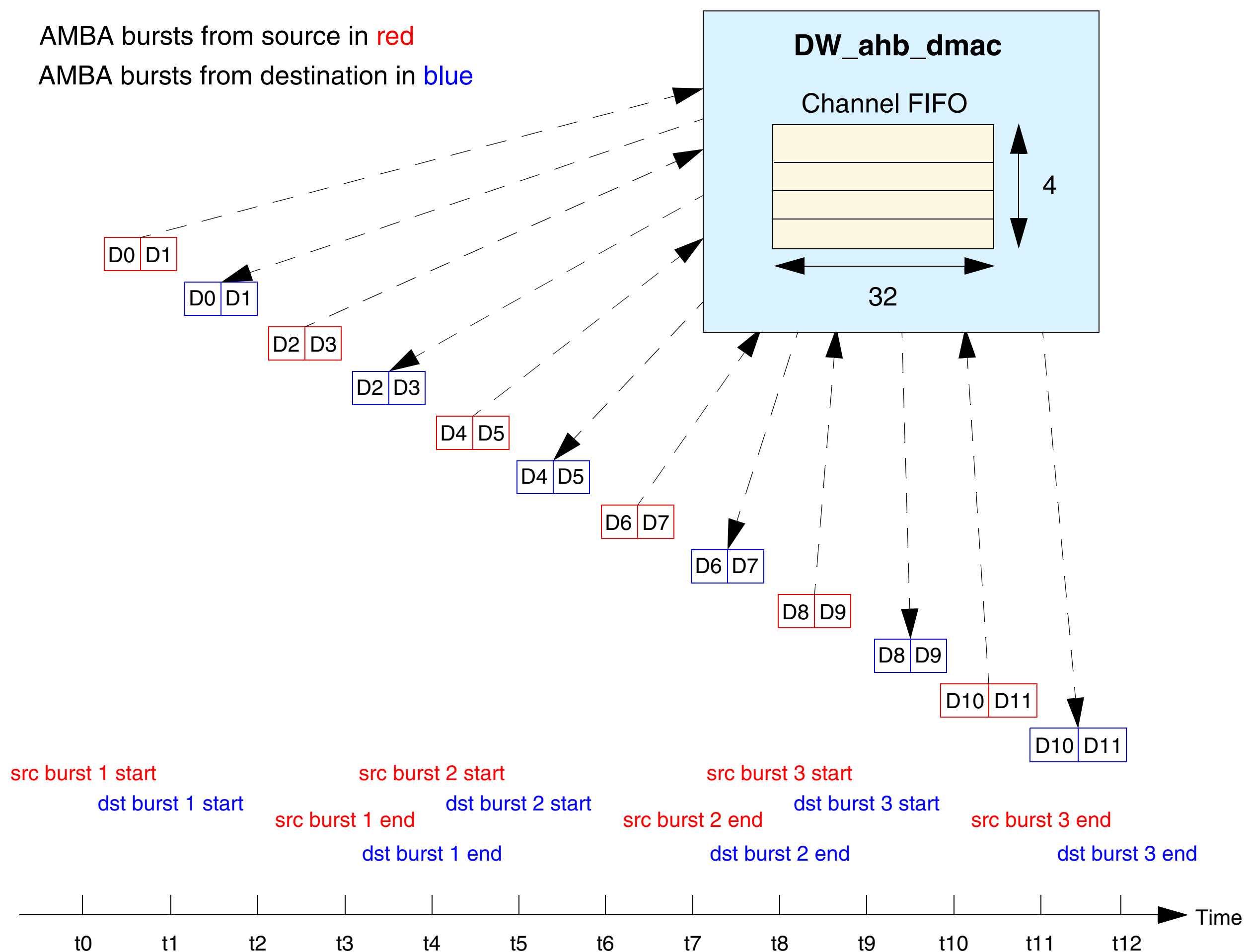
Adhering to the recommendation results in a reduced number of bursts per block, which in turn results in improved bus utilization and lower latency for block transfers.

2.9.1.3 Example 3

Scenario: Effect of the maximum AMBA burst length, CFG*x*. MAX_ABRST. This example is the same for both software and hardware handshaking interfaces.

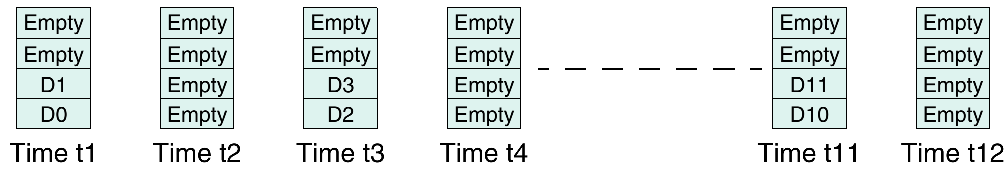
If the CFG*x*. MAX_ABRST = 2 parameter and all other parameters are left unchanged from Example 1, Table 2-7, then the block transfer would look like that shown in Figure 2-16.

Figure 2-16 Breakdown of Block Transfer where max_abrst = 2, Case 1



The channel FIFO is alternatively half filled by a burst from the source, and then emptied by a burst to the destination until the block transfer has completed; this is illustrated in Figure 2-17.

Figure 2-17 Channel FIFO Contents at Times Indicated in Figure 2-16



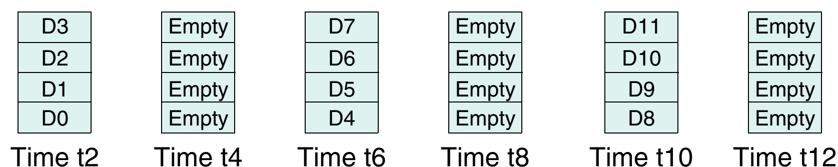
In this example block transfer, each source or destination burst transaction is made up of two bursts, each of length 2. As Figure 2-17 illustrates, the top two channel FIFO locations are redundant for this block transfer. However, this is not the general case. The block transfer could proceed as indicated in Figure 2-18.

Figure 2-18 Breakdown of Block Transfer where max_abrst = 2, Case 2

AMBA bursts from source in red
AMBA bursts from destination in blue
DW_ahb_dmac
Channel FIFO
4
32
D0 D1
D2 D3
D1 D0
D2 D3
D4 D5
D6 D7
D4 D5
D6 D7
D8 D9
D10 D11
D8 D9
D10 D11
src burst 1 start
dst burst 1 start
src burst 1 end
dst burst 1 end
src burst 2 start
dst burst 2 start
src burst 2 end
dst burst 2 end
src burst 3 start
dst burst 3 start
src burst 3 end
dst burst 3 end
Time
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12

This depends on the timing of the source and destination transaction requests, relative to each other. Figure 2-19 illustrates the channel FIFO status for Figure 2-18.

Figure 2-19 Channel FIFO Contents at Times Indicated in Figure 2-18



Recommendation: To allow a burst transaction to complete in a single burst, the following should be true:

CFG*x*.MAX_ABRST >= max(*src_burst_size_bytes*, *dst_burst_size_bytes*)

Adhering to the recommendation results in a reduced number of bursts per block, which in turn results in improved bus utilization and lower latency for block transfers. Limiting a burst to a maximum length prevents the DW_ahb_dmac from saturating the AHB bus when the DW_ahb system arbiter is configured to only allow changing of the grant signals to bus masters at the end of an undefined length burst. It also prevents a channel from saturating a DW_ahb_dmac master bus interface. For more information, refer to "Arbitration for AHB Master Interface" on page 98.

2.9.1.4 Example 4

Scenario: Source peripheral enters Single Transaction Region; the DW_ahb_dmac is the flow controller.

This example is the same for both hardware and software handshaking and demonstrates how a block from the source can be completed using a series of single transactions. It also demonstrates how the watermark level that triggers a burst request in the source peripheral can be dynamically adjusted so that the block transfer from the source completes with an Early-Terminated Burst Transaction. Table 2-8 lists the parameters used in this example.

Table 2-8 Parameters in Transfer Operation – Example 4

Parameter	Comment
CTL*x*.TT_FC = 3'b011	Peripheral-to-peripheral transfer with DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 12	–
CTL*x*.SRC_TR_WIDTH = 3'b010	32 bit
CTL*x*.DST_TR_WIDTH = 3'b010	32 bit
CTL*x*.SRC_MSIZE = 3'b010	Source burst transaction length = 8
CTL*x*.DEST_MSIZE = 3'b001	Destination burst transaction length = 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 16 bytes	–

In this case, BLOCK_TS is not a multiple of the source burst transaction length, CTL*x*.SRC_MSIZE, so near the end of a block transfer from the source, the amount of data left to be transferred is less than *src_burst_size_bytes*.

In this example, the block size is a multiple of the destination burst transaction length:

$$blk_size_bytes_dma/dst_burst_size_bytes = 48/16 = integer$$

The destination block is made up of three burst transactions to the destination and does not enter the Single Transaction Region.

The block size is not a multiple of the source burst transaction length:

$$blk_size_bytes_dma/src_burst_size_bytes = 48/32 \mathrel{!=} integer$$

Consider the case where the watermark level that triggers a source burst request in the source peripheral is equal to CTL*x*.SRC_MSIZE = 8; that is, eight entries or more need to be in the source peripheral FIFO in order to trigger a burst request.

Figure 2-20 shows how this block transfer is broken into burst and single transactions, and bursts and single transfers.

Figure 2-20 Breakdown of Block Transfer

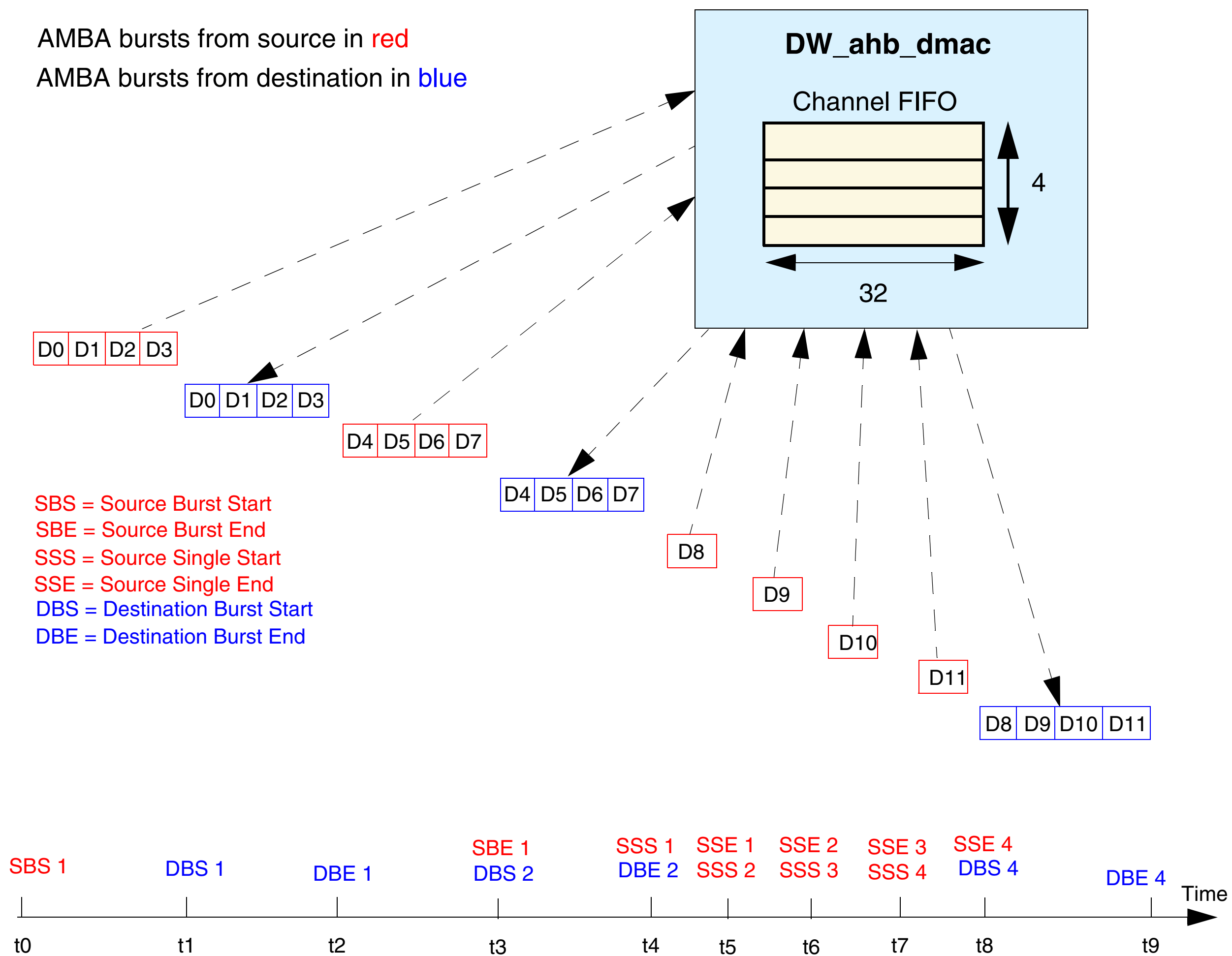
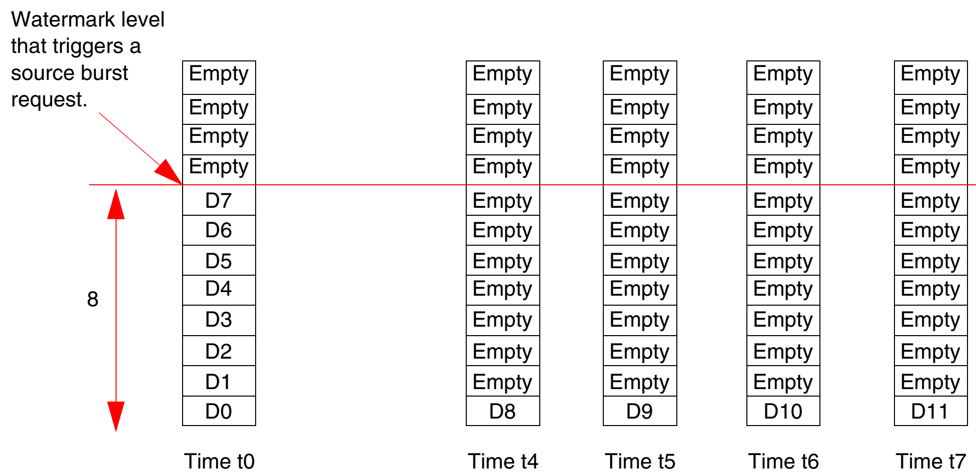


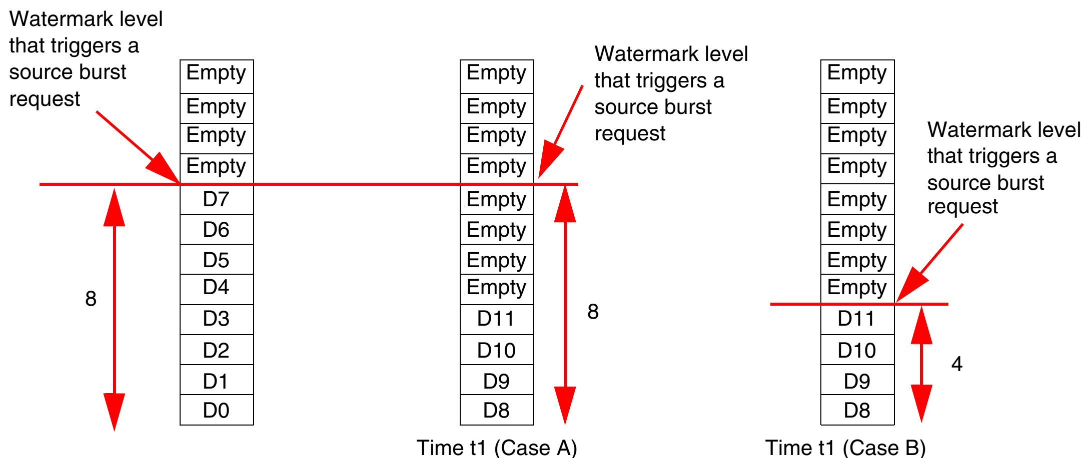
Figure 2-21 shows the status of the source FIFO at various times throughout the source block transfer.

Figure 2-21 Source FIFO Contents at Time Indicated in Figure 2-20



As shown in Figure 2-22, if the DW_ahb_dmac does not perform single transactions, the source FIFO contains four entries at time t1. However, the source has no more data to send. Therefore, if the watermark level remains at 8 (at time t1, Case A in Figure 2-22), the watermark level is never reached and a new burst request is never triggered.

Figure 2-22 Source FIFO Contents where Watermark Level is Dynamically Adjusted



The source peripheral, not knowing the length of a block and only able to request burst transactions, sits and waits for the FIFO level to reach a watermark level before requesting a new burst transaction request. This region, where the amount of data left to transfer in the source block is less than *src_burst_size_bytes*, is known as the Single Transaction Region.

In the Single Transaction Region, the DW_ahb_dmac performs single transactions from the source peripheral until the source block transfer has completed. In this example, the DW_ahb_dmac completes the source block transfer using four single transactions from the source.

Now consider Case B in Figure 2-22, where the source peripheral can dynamically adjust the watermark level that triggers a burst transaction request near the end of a block. After the first source burst transaction

completes, the source peripheral recognizes that it has only four data items left to complete in the block and adjusts the FIFO watermark level that triggers a burst transaction to 4. This triggers a burst request, and the block completes using a burst transaction. However, CTL*x*.SRC_MSIZE = 8, and there are only four data items left to transfer in the source block. The DW_ahb_dmac terminates the last source burst transaction early and fetches only four of the eight data items in the last source burst transaction. This is called an Early-Terminated Burst Transaction.

Observation: Under certain conditions, it is possible to hardcode dma_single from the source peripheral to an inactive level (hardware handshaking). Under the same conditions, it is possible for software to complete a source block transfer without initiating single transactions from the source. For more information, refer to "Single Transactions – Peripheral Is Not Flow Controller" on page 47.

2.9.1.5 Example 5

Scenario: The destination peripheral enters the Single Transaction Region while the DW_ahb_dmac is the flow controller. This example also demonstrates how the DW_ahb_dmac channel FIFO is flushed at the end of a block transfer to the destination; this example is the same for both hardware and software handshaking.

Consider the case with the parameters set to values listed in Table 2-9.

Table 2-9 Parameters in Transfer Operation – Example 5

Parameter	Comment
CTL*x*.TT_FC = 3'b011	Peripheral-to-peripheral transfer with DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 44	–
CTL*x*.SRC_TR_WIDTH = 3'b000	8 bit
CTL*x*.DST_TR_WIDTH = 3'b 011	64bit
CTL*x*.SRC_MSIZE = 3'b001	Source burst transaction length = 4
CTL*x*.DEST_MSIZE = 3'b001	Destination burst transaction length = 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 32 bytes	–

In this example, the block size is a multiple of the source burst transaction length:

blk_size_bytes_dma/src_burst_size_bytes = (44 * 1)/4 = 11 = *integer*

The source block transfer is completed using only burst transactions, and the source does not enter the Single Transaction Region.

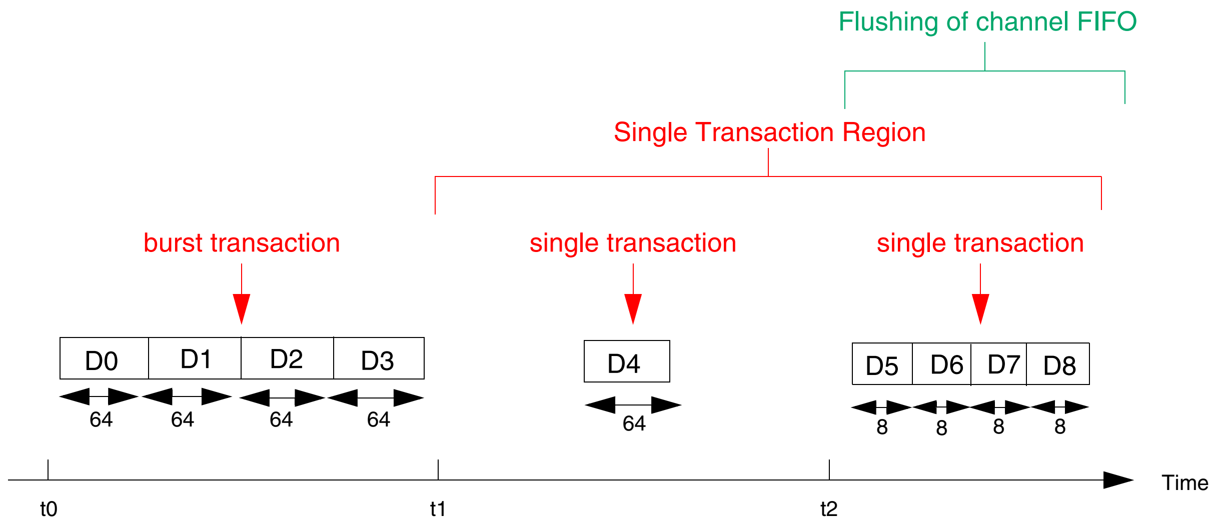
The block size is not a multiple of the destination burst transaction length:

blk_size_bytes_dma/dst_burst_size_bytes 44/32 != *integer*

So near the end of the block transfer to the destination, the amount of data left to be transferred is less than *dst_burst_size_bytes* and the destination enters the Single Transaction Region.

Figure 2-23 shows one way in which the block transfer to the destination can occur.

Figure 2-23 Block Transfer to Destination



After the first 32 bytes (*dst_burst_size_bytes = 32*) of the destination burst transaction have been transferred to the destination, there are 12 bytes (*blk_size_bytes_dma -dst_burst_size_bytes* = 44 - 32) left to transfer. This is less then the amount of data that is transferred in a destination burst (*dst_burst_size_bytes* = 32). Therefore, the destination peripheral enters the Single Transaction Region where the DW_ahb_dmac can complete a block transfer to the destination using single transactions.

Note

- In the Single Transaction Region, asserting dma_single initiates a single transaction for hardware handshaking. Writing a 1 to the relevant channel bit of the SglReqDstReg register initiates a single transaction for software handshaking.
- The destination peripheral, not knowing the length of a block and only able to request burst transactions, sits and waits for the FIFO to fall below a watermark level before requesting a new burst transaction request.

At time t2 in Figure 2-23, a single transaction to the destination has been completed. There are now only four bytes (12 - *dst_single_size_bytes* = 12 - 8) left to transfer in the destination block. However, CTL*x*.DST_TR_WIDTH implies 64-bit AHB transfers to the destination (*dst_single_size_bytes* = 8 byte); therefore, the DW_ahb_dmac cannot form a single word of the specified CTL*x*.DST_TR_WIDTH.

The DW_ahb_dmac channel FIFO has four bytes in it that must be flushed to the destination. The DW_ahb_dmac switches into a "FIFO flush mode," where the block transfer to the destination is completed by changing the AHB transfer width to the destination to be equal to that of the CTL*x*.SRC_TR_WIDTH; that is, byte AHB transfers in this example. Thus the last single transaction in the destination block is made up of a burst of length 4 and CTL*x*.SRC_TR_WIDTH width.

When the DW_ahb_dmac is in FIFO flush mode, the address on haddr is incremented by the value of hsize on the bus; that is, CTL*x*.SRC_TR_WIDTH, and not CTL*x*.DST_TR_WIDTH. In cases where the DARx is selected to be contiguous between blocks (refer to Table 6-1 and "Programming Examples" on page 343), the DAR*x* will need re-alignment at the start of the next block, since it is aligned to CTLx.SRC_TR_WIDTH and

not CTLx.DST_TR_WIDTH at the end of the previous block. This is handled by hardware. For more information, refer to "Hardware Realignment of SAR/DAR Registers" on page 65.

In general, channel FIFO flushing to the destination occurs if all three of the following are true:

- DW_ahb_dmac or the Source peripheral are flow control peripherals
- CTL*x*.DST_TR_WIDTH > CTL*x*.SRC_TR_WIDTH
- Flow control device:
  - If DW_ahb_dmac is flow controller:

    blk_size_bytes_dma/dst_single_size_bytes != *integer*
  - If source is flow controller:

    blk_size_bytes_src/dst_single_size_bytes != *integer*

Note When not in FIFO flush mode, a single transaction is mapped to a single AHB transfer. However, in FIFO flush mode, a single transaction is mapped to multiple AHB transfers of CTL*x*.SRC_TR_WIDTH width. The cumulative total of data transferred to the destination in FIFO flush mode is less than *dst_single_size_bytes*.

In this example, a burst request is not generated in the Single Transaction Region. If a burst request is generated at time t1 in Figure 2-23, then the burst transaction proceeds until there is not enough data left in the destination block to form a single data item of CTL*x*.DST_TR_WIDTH width. The burst transaction would then be early-terminated. In this example, only one data item of the four requested (decoded value of DEST_MIZE = 4) would be transferred to the destination in the burst transaction. This is referred to as an Early-Terminated Burst Transaction. If a burst request is generated at time t2 in Figure 2-23, then the destination block is completed (four byte transfers to the destination to flush the DW_ahb_dmac channel FIFO) and this burst request is again be early-terminated at the end of the destination block.

Observation: If the source transfer width – CTL*x*.SRC_TR_WIDTH in the channel control register (CTL*x*) – is less than the destination transfer width (CTL*x*.DST_TR_WIDTH), then the FIFO may need to be flushed at the end of the block transfer. This is done by setting the AHB transfer width of the last few AHB transfers of the block to the destination so that it is equal to CTL*x*.SRC_TR_WIDTH and not the programmed CTL*x*.DST_TR_WIDTH.

2.9.1.5.1 Hardware Realignment of SAR/DAR Registers

In a particular circumstance, during contiguous multi-block DMA transfers, the destination address can become misaligned between the end of one block and the start of the next block. When this situation occurs, DW_ahb_dmac re-aligns the destination address before the start of the next block.

Consider the following example. If the block length is 9, the source transfer width is 16 (halfword), and the destination transfer width is 32 (word)—the destination is programmed for contiguous block transfers—then the destination performs four word transfers followed by a halfword transfer to complete the block transfer to the destination. At the end of the destination block transfer, the address is aligned to a 16-bit transfer as the last AMBA transfer is halfword. This is misaligned to the programmed transfer size of 32 bits for the destination. However, for contiguous destination multi-block transfers, DW_ahb_dmac re-aligns the DAR address to the nearest 32-bit address (next 32-bit address upwards if address control is incrementing or next address downwards if address control is decrementing).

The destination address is automatically realigned by the DW_ahb_dmac in the following DMA transfer setup scenario:

- Contiguous multi-block transfers on destination side, AND
- DST_TR_WIDTH > SRC_TR_WIDTH, AND
- (BLOCK_TS * SRC_TR_WIDTH)/DST_TR_WIDTH != integer (where SRC_TR_WIDTH, DST_TR_WIDTH is byte width of transfer)

2.9.1.6 Example 6

Scenario: In all examples presented so far, none of the bursts have been early-terminated by the DW_ahb system arbiter. Referring to Example 1, the AHB transfers on the source and destination side look somewhat symmetric. In the examples presented so far, where the bursts are not early-terminated by the DW_ahb system arbiter, the traffic profile on the AHB bus would be the same, regardless of the value of CFG*x*.FIFO_MODE. This example, however, considers the effect of CFG*x*.FIFO_MODE; it is the same for both hardware and software handshaking.

CFG*x*.FIFO_MODE: Determines how much space or data needs to be available in the FIFO before a burst transaction request is serviced.

0 = Space/data available for single AHB transfer of the specified transfer width.

1 = Data available is greater than or equal to half the FIFO depth for destination transfers and space available is greater than half the fifo depth for source transfers. The exceptions are at the end of a burst transaction request or at the end of a block transfer.

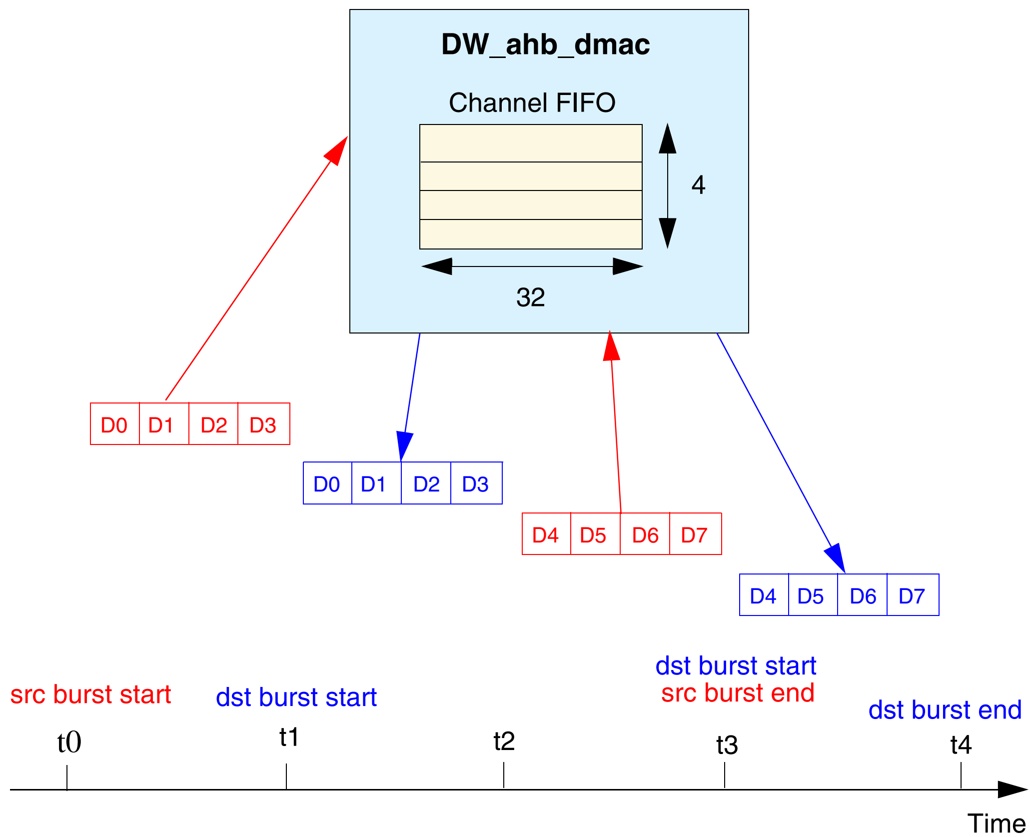
Table 2-10 lists the parameters used in this example.

Table 2-10 Parameters in Transfer Operation – Example 6

Parameter	Comment
CTL*x*.TT_FC = 3’b011	Peripheral-to-peripheral transfer with DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 32	–
CTL*x*.SRC_TR_WIDTH = 3’b010	32 bit
CTL*x*.DST_TR_WIDTH = 3’b010	32 bit
CTL*x*.SRC_MSIZE = 3’b010	Decoded value = 8
CTL*x*.DEST_MSIZE = 3’b001	Decoded value = 4
CFG*x*.MAX_ABRST = 1’b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 16 bytes	–

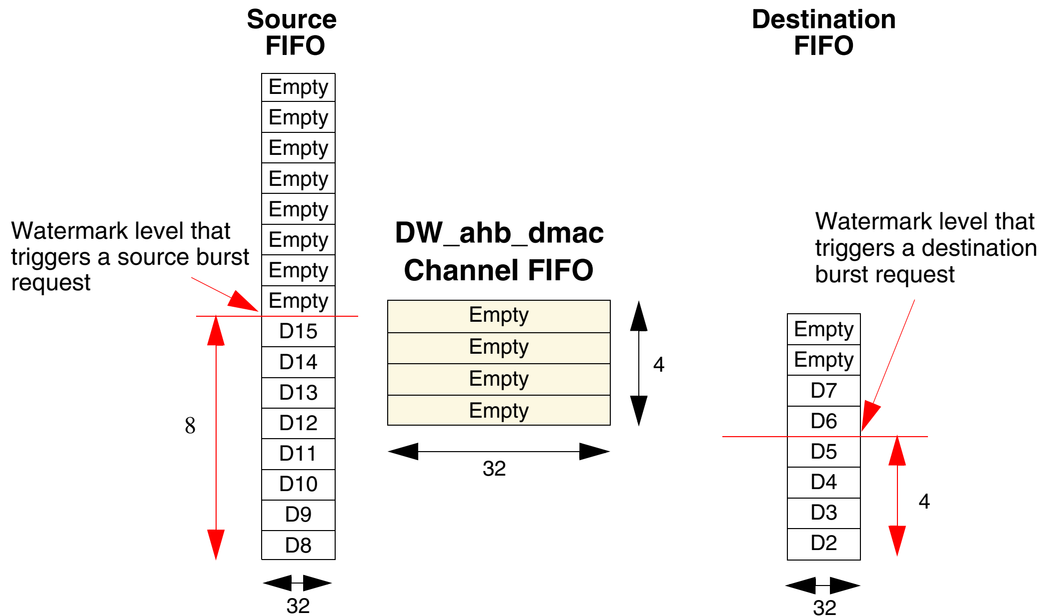
The block transfer may proceed by alternately filling and emptying the DW_ahb_dmac channel FIFO. Up to time t4, the transfer might proceed like that shown in Figure 2-24.

Figure 2-24 Block Transfer Up to Time "t4"



At time t4, the src, channel, and destination FIFOs might look like that shown in Figure 2-25

Figure 2-25 Source, DW_ahb_dmac Channel and Destination FIFOs at Time ‘t4’ in Figure 2-22



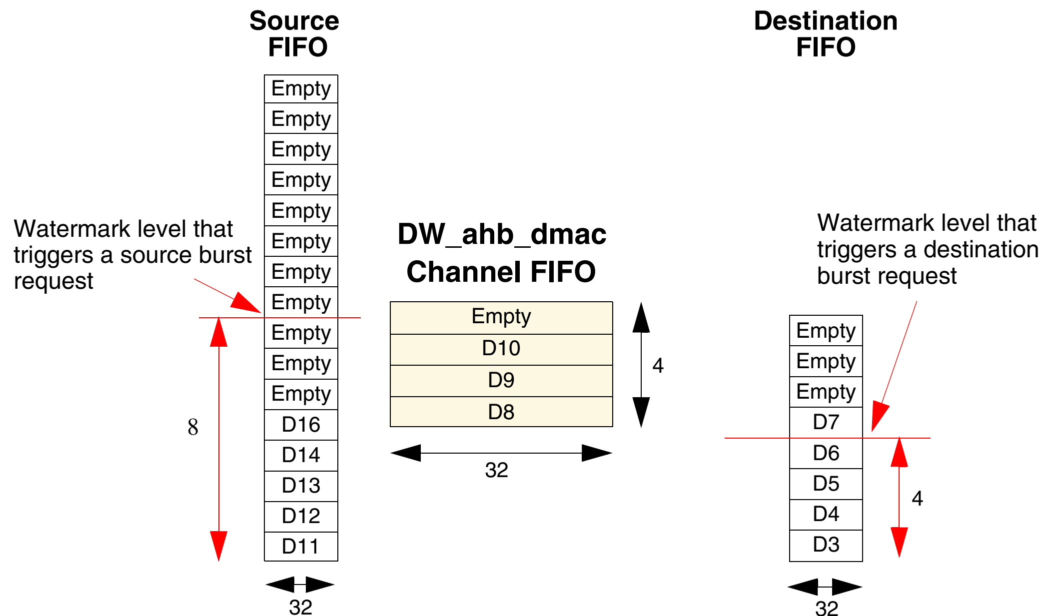
At time t4, a source burst transaction is requested, and the DW_ahb_dmac attempts a burst of length 4. Suppose that this burst is early-burst terminated after three AHB transfers. The FIFO status after this burst might look like that shown in Figure 2-26.

Referring to Figure 2-26, notice that a burst request from the destination is not triggered, since the destination FIFO contents are above the watermark level. The DW_ahb_dmac has space for one data item in the channel FIFO. So, does the DW_ahb_dmac initiate a single AHB transfer from the source peripheral to fill the channel FIFO?

The answer depends on the value of CFG*x*.FIFO_MODE. If CFG*x*.FIFO_MODE = 0, then the DW_ahb_dmac attempts to perform a single AHB transfer in order to fill the channel FIFO. If CFG*x*.FIFO_MODE = 1, then

the DW_ahb_dmac waits until the channel FIFO is less than half-full before initiating a burst from the source, as illustrated in Figure 2-26.

Figure 2-26 FIFO Status After Early-Terminated Burst



Observation: When CFG*x*.FIFO_MODE = 1, the number of bursts per block is less than when CFG*x*.FIFO_MODE = 0 and, hence, the bus utilization will improve. This setting favors longer bursts. However, the latency of DMA transfers may increase when CFG*x*.FIFO_MODE = 1, since the DW_ahb_dmac waits for the channel FIFO contents to be less than half the FIFO depth for source transfers, or greater than or equal to half the FIFO depth for destination transfers. Therefore, system bus occupancy and usage can be improved by delaying the servicing of multiple requests until there is sufficient data/space available in the FIFO to generate a burst (rather than multiple single AHB transfers); this comes at the expense of transfer latency. For reduced block transfer latency, set CFG*x*.FIFO_MODE = 0. For improved bus utilization, set CFG*x*.FIFO_MODE = 1.

2.9.1.7 Example 7

Scenario: Example block transfer when the destination is the flow controller; the effect of data pre-fetching (CFG*x*.FCMODE = 0) and possible data loss. This example uses a hardware handshaking interface, but the same scenario can be explained using a software handshaking interface.

Note Data pre-fetching is when data is fetched from the source before the destination requests it.

Flow Control Mode. Determines when source transaction requests are serviced when the Destination Peripheral is the flow controller.

0 = Source transaction requests are serviced when they occur. Data pre-fetching is enabled.

1 = Source transaction requests are not serviced until a destination transaction request occurs. In this mode, the amount of data transferred from the source is limited such that it is guaranteed to be

transferred to the destination prior to block termination by the destination. Data pre-fetching is disabled.

Table 2-11 lists the parameters used in this example.

Table 2-11 Parameters in Transfer Operation – Example 7

Parameter	Description
CTL*x*.TT_FC=3'b111	Peripheral-to-peripheral transfer with destination as flow controller
CTL*x*.BLOCK_TS = *x*	–
CTL*x*.SRC_TR_WIDTH = 3'b010	32 bit
CTL*x*.DEST_TR_WIDTH=3'b010	32 bit
CTL*x*.SRC_MSIZE = 3'b010	Decoded value = 8
CTL*x*.DEST_MSIZE = 3'b001	Decoded value = 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 32	–
CFG*x*.FCMODE = 0	Data pre-fetching enabled
CFG*x*.SRC_PER = 0	Source assigned handshaking interface 0
CFG*x*.DEST_PER = 1	Destination assigned handshaking interface 1
CFG*x*.MAX_ABRST = 7	–

Consider a case where the destination block is made up of one burst transaction, followed by one single transaction.

$$blk_size_bytes_dst = dst_burst_size_bytes = 16 + 4 = 20 \text{ bytes}$$

There are a number of different cases that can arise when CFG*x*.FCMODE = 0:

1. When the destination peripheral signals a last transaction, there is enough data in the DW_ahb_dmac channel FIFO to complete the last transaction to the destination. Therefore, the DW_ahb_dmac stops transferring data from the source, and the block transfer from the source completes; any surplus data that has been fetched from the source is effectively lost. Two cases arise when the destination peripheral signals the last transaction in a block:
   a. Active burst transaction request on source side
   b. No active burst request on source side

2. When the destination peripheral signals a last transaction, there is not enough data in the channel FIFO to complete the last transaction to the destination. The DW_ahb_dmac fetches just enough data to complete the block transfer. Two cases arise.
   a. Source enters Single Transaction Region when destination peripheral signals last transaction.
   b. Source does not enter Single Transaction Region when destination peripheral signals last transaction.

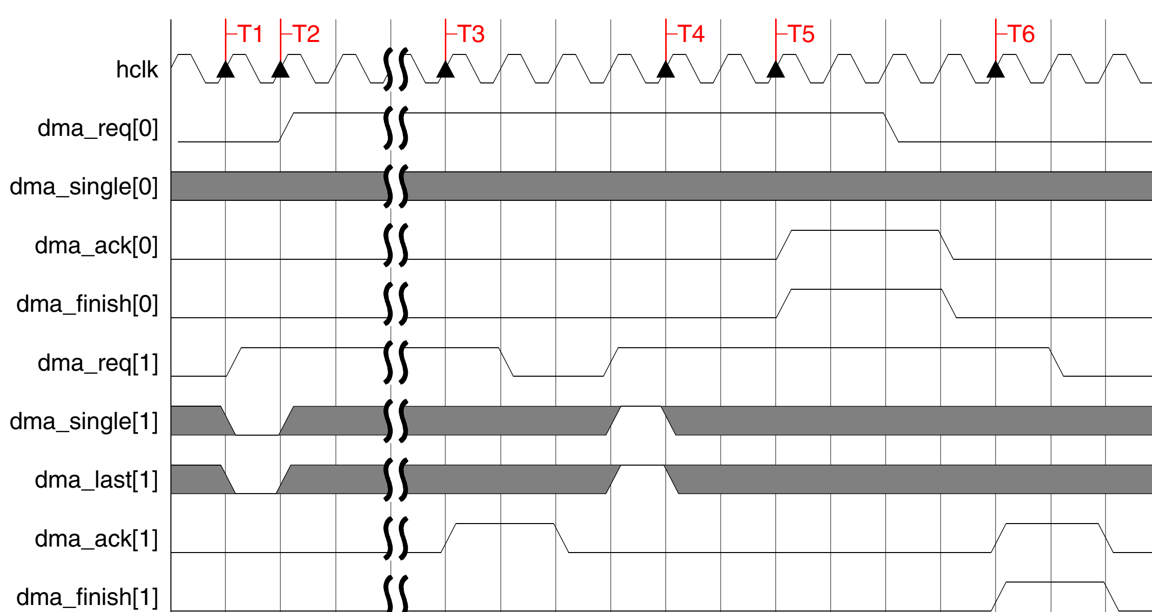
Setting CFG*x*.FCMODE is pertinent only when the destination peripheral is the flow controller. When CFG*x*.FCMODE = 0, scenarios arise where not all the data that has been pre-fetched from the source is required to complete the block transfer to the destination. This excess data is not transferred to the destination peripheral and is effectively lost for a read-sensitive source peripheral. In this example, assume a read-sensitive source peripheral.

Case a and Case b highlight instances where data pre-fetching is enabled and data is lost. Case a and Case b highlight instances are data pre-fetching is enabled, but no data loss occurs.

In this example, handshaking interface 0 is assigned to the source peripheral, and handshaking interface 1 is assigned to the destination peripheral.

Consider the block transfer shown in Figure 2-27, where the destination is the flow controller and data pre-fetching is enabled (CFG*x*.FCMODE = 0).

Figure 2-27 Data Loss when Pre-Fetching is Enabled



The source requests a burst transaction at time T2. The destination requests a burst transaction at time T1 and completes this burst request at time T3. At time T4, the destination requests a single transaction, which is to be the last in the block transfer. Suppose that at time T4 the DW_ahb_dmac has fetched seven words from the source and written four words to the destination. Therefore, the DW_ahb_dmac has three words in the channel FIFO, but the destination requires only one word to complete the block transfer.

The DW_ahb_dmac, recognizing that it has enough data in the channel FIFO to complete the block transfer to the destination, fetches no more data from the source and early-terminates the source burst transaction

(only seven of the eight data items in the source burst transaction have been fetched from the source) – Early-Terminated Burst Transaction. The DW_ahb_dmac asserts dma_finish[0] to the source at time T5, and this has the same timing as dma_ack[0], as shown in Figure 2-27.

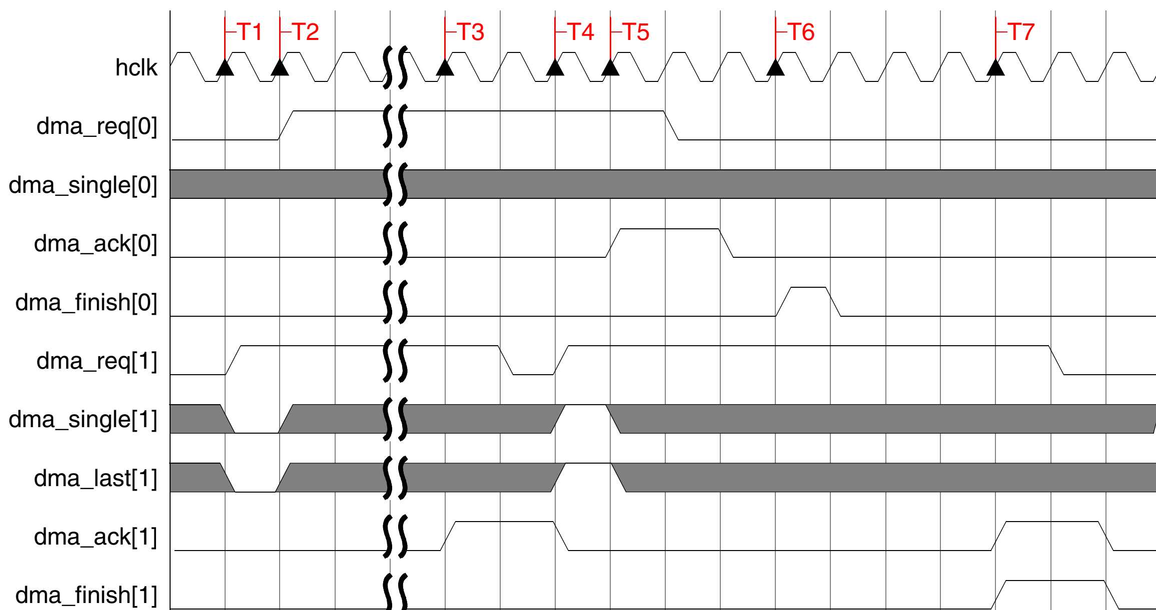
At time T6, the last single transaction to the destination has completed, which has removed one of the remaining three data words in the channel FIFO. At this time, both the source and destination block transfers have completed, and there remain two data words in the channel FIFO that have been fetched from the source. These two data words are lost because they do not form the start of the next block transfer for multi-block transfers, since the channel FIFO is cleared between blocks for multi-block transfers.

There is an exception to Case a. If the last AHB transfer to the source received a SPLIT/RETRY response over the AHB bus, then dma_finish is not asserted until the AHB transfer that received the SPLIT/RETRY response is retried and an OKAY response is received over the hresp AHB bus; to do otherwise would be a violation of the AMBA protocol. This additional word that is fetched is effectively lost, since it is not transferred to the destination.

2.9.1.7.1 Case 1b – Timing exception on dma_finish to the source when data pre-fetching is enabled

Consider the block transfer shown in Figure 2-28, where the destination is the flow controller and data pre-fetching is enabled (CFG*x*.FCMODE = 0).

Figure 2-28 Timing Exception on dma_finish to Source Peripheral



The source requests a burst transaction at time T2 and completes the burst transaction at time T5. The destination requests a burst transaction at time T1 and completes this burst request at time T3. At time T4, the destination requests a single transaction, which is to be the last in the block transfer. At time T5, the DW_ahb_dmac has completed the burst transaction from the source.

At time T5, the DW_ahb_dmac has fetched eight words from the source and written four words to the destination, which means that the DW_ahb_dmac has four words in the channel FIFO. However, the destination requires only one word to complete the block transfer. The DW_ahb_dmac, recognizing that it

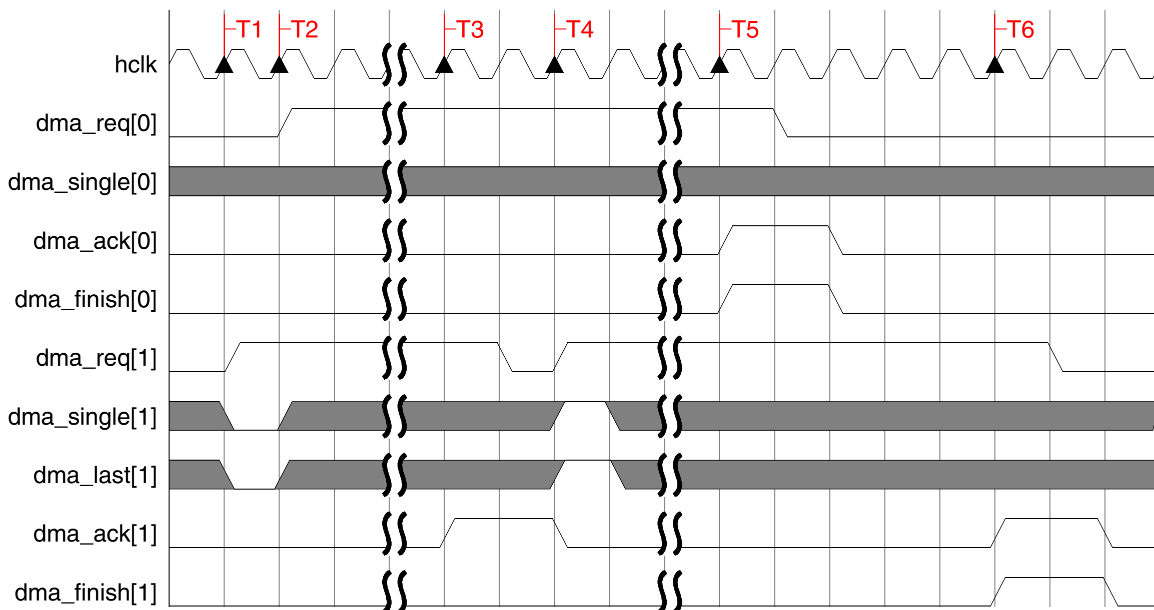
has enough data in the channel FIFO to complete the block transfer to the destination, fetches no more data from the source and signals a source block completion by asserting dma_finish[0] for a single cycle at time T6. Since there is no active transaction on the source side – that is, the previous source burst transaction has completed and there has been no new burst request from the source – the dma_finish[0] cannot form a handshaking loop with dma_req[0] (there is no active burst request) and therefore is asserted for only a single cycle.

Similar to Case a, when both the source and destination block transfers have completed at time T7, there are three data items left in the channel FIFO that are effectively lost.

2.9.1.7.2 Case 2a – Data pre-fetching enabled but no data loss. Source enters Single Transaction Region when destination signals last transaction.

Consider the block transfer as shown in Figure 2-29, where the destination is a flow controller and data pre-fetching is enabled (CFG*x*.FCMODE = 0). The transfer parameters are the same as case 1a, Table 2-11.

Figure 2-29 Case of No Data Loss When Pre-Fetching is Enabled



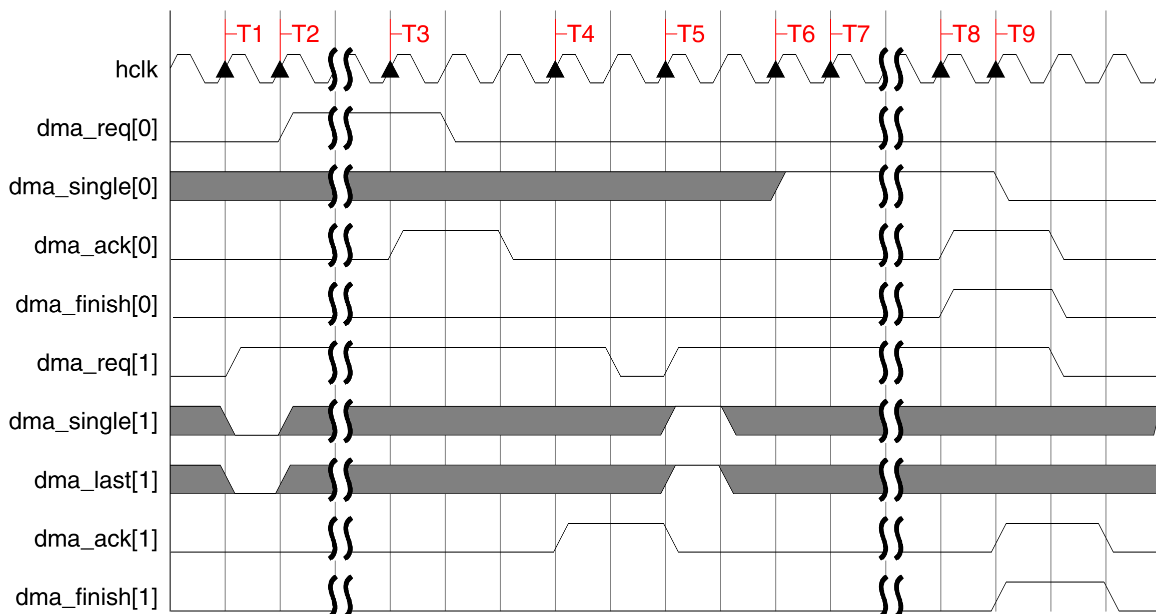
In this scenario, when dma_last[1] is asserted by the destination peripheral at time T4, there is not enough data in the channel FIFO to complete the last single transaction. Assume that the DW_ahb_dmac has fetched four data items from the source peripheral at time T4. In this case, the DW_ahb_dmac fetches one more data item from the source peripheral and then early terminates the source burst using an Early-Terminated Burst Transaction. The DW_ahb_dmac signals block completion to the source by asserting dma_finish[0] at T5, which forms a handshaking loop with dma_req[0]. In this case, there is no data loss, and all data that is fetched from the source has been transferred to the destination.

Consider the case where the transfer parameters are as given in Table 2-12.

Table 2-12 Transfer Parameters

Parameter	Description
CTL*x*.TT_FC=3'b111	Peripheral-to-peripheral transfer with destination as flow controller
CTL*x*.BLOCK_TS = *x*	
CTL*x*.SRC_TR_WIDTH = 3'b010	32 bit
CTL*x*.DEST_TR_WIDTH=3'b010	32 bit
CTL*x*.SRC_MSIZE = 3'b001	Decoded value = 4
CTL*x*.DEST_MSIZE = 3'b001	Decoded value = 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 32	–
CFG*x*.FCMODE = 0	Data pre-fetching enabled
CFG*x*.SRC_PER = 0	Source assigned handshaking interface 0
CFG*x*.DEST_PER = 1	Destination assigned handshaking interface 1
CFG*x*.MAX_ABRST = 7	–

Figure 2-30 Source Enters Single Transaction Region when Destination Asserts dma_last[1]



As illustrated in Figure 2-30, the source requests a burst transaction at time T2 and completes the burst transaction at time T3. The destination requests a burst transaction at time T1 and completes this burst request at time T4. The destination requests a last single transaction at time T5; the channel FIFO is empty at this time. The amount of data left to complete a source block transfer – 4 bytes – is less than the following:

src_burst_size_bytes = 4 * 4 = 16 bytes

Therefore, the source enters the Single Transaction Region at time T6. At time T7, the DW_ahb_dmac samples that dma_single[0] from the source peripheral is asserted and initiates a single transaction.

Note If an active level on dma_req[0] is triggered at time T6 or T7, a source burst transaction takes precedence over the source single transaction. Upon completion of the source block, this burst transaction would have been early-terminated using an Early-Terminated Burst Transaction.

When this transaction completes at time T8, the DW_ahb_dmac recognizes that enough data has been fetched from the source peripheral to complete the block transfer to the destination. The DW_ahb_dmac asserts dma_finish[0] to the source peripheral at time T8; this has the same timing as dma_ack[0]. The destination block transfer completes as previously described. No data loss occurs.

2.9.1.7.3 Case 2b – Data pre-fetching enabled but no data loss.

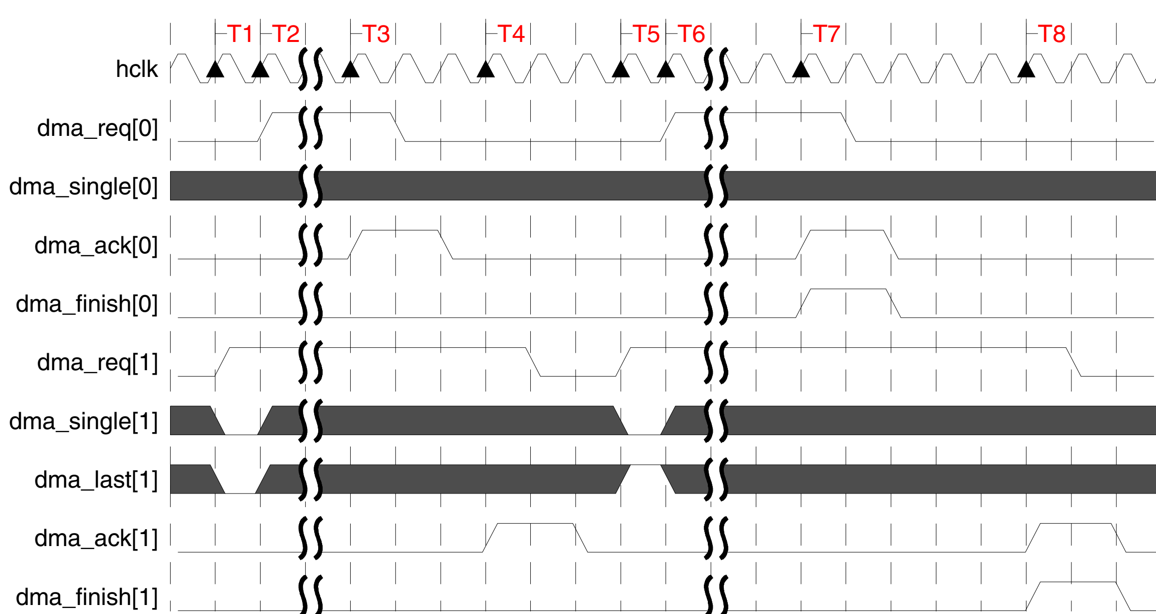
In this case, the source does not enter the Single Transaction Region when the destination signals the last transaction. This case uses the parameters listed in Table 2-12

The destination block is made up of two burst transactions.

blk_size_bytes_dst = 2 * (4 * 4) = 32 bytes

As illustrated in Figure 2-31, the source requests a burst transaction at time T2 and completes the burst transaction at time T3. The destination requests a burst transaction at time T1 and completes this burst request at time T4.

Figure 2-31 Case where Source Does Not Enter *Single transaction region* when Destination Asserts dma_last[1]



At time T5 the destination peripheral requests the last burst transaction in the block transfer to the destination. At this point, the channel FIFO is empty. The number of bytes that must be fetched from the source peripheral to complete the block transfer to the destination is equal to 4 * 4 = 16 bytes. Since 16 bytes is not less than *src_msize_bytes* (16 bytes), the source does not enter the Single Transaction Region. The DW_ahb_dmac waits for a burst request from the source peripheral, which occurs at time T6. Upon

completion of this burst request at time T7, the DW_ahb_dmac signals a source block transfer completion and asserts dma_finish[0]. Upon completion of the last destination burst transaction at time T8, the DW_ahb_dmac signals a destination block transfer completion and asserts dma_finish[1] to the destination.

Note that when data pre-fetching is enabled, CFG*x*.FCMODE = 0, the maximum amount of data that can be lost depends on whether the last transaction in the block transfer to the destination is a single transaction or burst transaction. In the worst case scenario, the DW_ahb_dmac has pre-fetched enough data from the source to fill the channel FIFO when the last transaction is signalled by the destination peripheral.

The maximum amount of data that can be lost is:

- Last transaction in block transfer is a single transaction:

  DMAH_CH*x*_FIFO_DEPTH - *dst_single_size_bytes* [refer to equation (1)]

- Last transaction in block transfer is a burst transaction:

  DMAH_CH*x*_FIFO_DEPTH - *dst_burst_size_bytes* [refer to equation (2)]
  If this equation is <= 0, then no data is lost.

Thus, if the last transaction in the block is a burst transaction and equation (2) is less than zero, then no data can be lost when CFG*x*.FCMODE = 0. There is one exception to this, as outlined in Example 8.

Enabling data pre-fetching may reduce the latency of the DMA transfer when the destination is the flow controller.

Observation: For a source peripheral that is not read-sensitive (such as memory), data pre-fetching should be enabled – that is, CFG*x*.FCMODE = 0 – in order to reduce the transfer latency when the destination is the flow controller. If the source peripheral is a read-sensitive device (such as a source FIFO), then data pre-fetching should be disabled – that is, CFG*x*.FCMODE = 1 – when the destination peripheral is the flow controller.

2.9.1.8 Example 8

Scenario: Data loss when destination is flow controller and data pre-fetching is disabled; CFG*x*.FCMODE = 1.

This example uses a hardware handshaking interface, but the same scenario can be explained using a software handshaking interface. Two scenarios arise when CFG*x*.FCMODE = 1:

- CTL*x*.SRC_TR_WIDTH <= CTL*x*.DST_TR_WIDTH
- CTL*x*.SRC_TR_WIDTH > CTL*x*.DST_TR_WIDTH

2.9.1.8.1 Case 1 – CTL*x*.SRC_TR_WIDTH <= CTL*x*.DST_TR_WIDTH

In this case, the DW_ahb_dmac controls the transfer of data from the source, such that at any time there is at most enough data to complete the current transaction – single or burst – to the destination. If there is currently no active transaction to the destination, then the channel FIFO is empty and no data is pre-fetched from the source, even if the source has an active transaction request. If both the source and destination are requesting, then the DW_ahb_dmac fetches only enough data from the source to complete the current destination transaction, and no more. Therefore, there can never be any data loss.

2.9.1.8.2 Case 2 – CTL*x*.SRC_TR_WIDTH > CTL*x*.DST_TR_WIDTH

In this example, assume the parameters in Table 2-13.

Table 2-13 Parameters in Transfer Operation – Example 7, Case 2b

Parameter	Description
CFG*x*.FCMODE = 1	Data pre-fetching disabled
CTL*x*.BLOCK_TS = *x*	–
CTL*x*.SRC_MSIZE = 3'b001	Decoded value = 4
CTL*x*.DEST_MSIZE = 3'b010	Decoded value = 8
CTL*x*.SRC_TR_WIDTH = 3'b011	64-bit
CTL*x*.DST_TR_WIDTH=3'b010	32-bit
CTL*x*.TT_FC=3'b111	Peripheral to Peripheral transfer with destination as flow controller
DMAH_CH*x*_FIFO_DEPTH = 32	–
CFG*x*.SRC_PER = 0	Source assigned handshaking interface 0
CFG*x*.DEST_PER = 1	Destination assigned handshaking interface 1
CFG*x*.MAX_ABRST = 8	–

Consider the case where the destination block is made up of a burst transaction, followed by one single transaction:

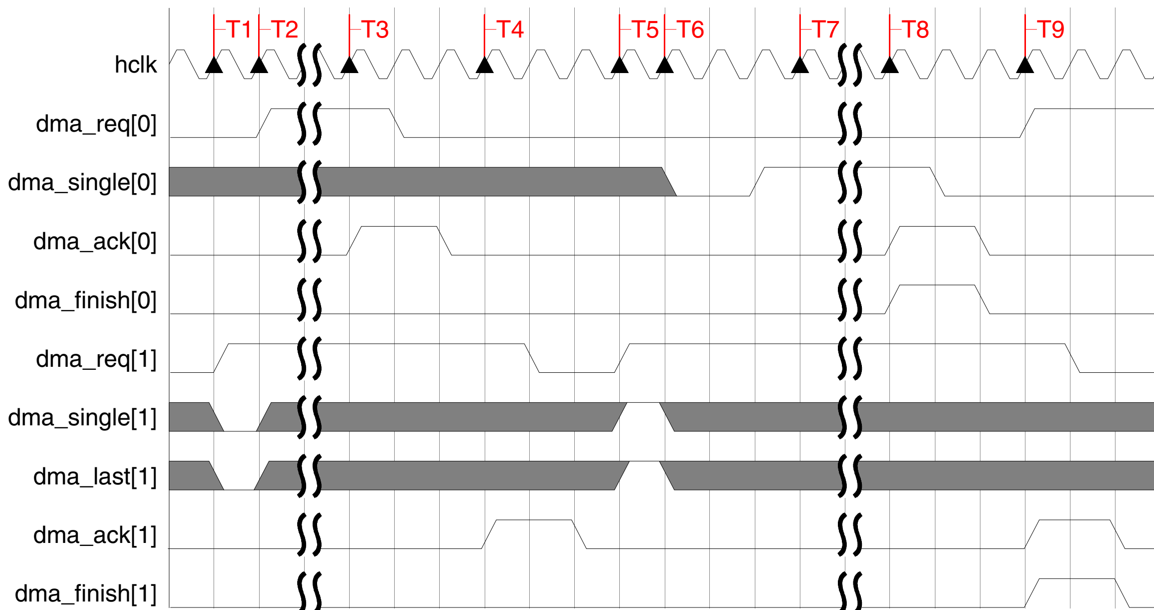
blk_size_bytes_dst = (8 * 4) + 4 = 36 bytes

src_single_size_bytes = 8
dst_single_size_bytes = 4
src_burst_size_bytes = 4 * 8 = 32
dst_burst_size_bytes = 8 * 4 = 32

As illustrated in Figure 2-32, the source requests a burst transaction at time T2, and completes the burst transaction at time T3. The destination requests a burst transaction at time T1 and completes this burst

request at time T4. The destination requests a last single transaction at time T5. The channel FIFO is empty at this time.

Figure 2-32 Data Loss when Data Pre-Fetching is Disabled



The amount of data left to complete a source block transfer, 4 bytes, is less than *src_burst_size_bytes* (32 bytes). Therefore, the source enters the Single Transaction Region. At time T7, the DW_ahb_dmac samples that dma_single[0] is asserted and initiates a single transaction from the source.

The DW_ahb_dmac fetches a single source data item from the source peripheral and stores the eight bytes in the channel FIFO. This single transaction completes at time T8; the source block also completes at time T8. However, the destination requires only four of these eight bytes to complete the block transfer to the destination (the block transfer to destination completes at time T9). The remaining four bytes are lost. Thus, when the destination is the flow controller, data loss occurs when CFG*x*.FCMODE = 1 if both of the following are true:

- SRC_TR_WITDT > CTL*x*.DST_TR_WIDTH
- *blk_size_bytes_dst/src_single_size_bytes* != integer.

The amount of data lost is:

src_single_size_bytes - dst_single_size_bytes [refer to equations (2) and (3)]

Observation: Data loss can occur when the destination is the flow controller, even if data pre-fetching is disabled, CFG*x*.FCMODE = 1.

2.9.1.9 Example 9

Scenario: This scenario demonstrates how to use software handshaking on both the source and destination sides when the DW_ahb_dmac is the flow controller. The DW_apb_ssi is used as the source and destination peripherals; for more information on the DW_apb_ssi, refer to the *DesignWare DW_apb_ssi Databook*. This example uses the parameters listed in Table 2-14.

Table 2-14 Parameters in Transfer Operation – Example 9

Parameter	Description
CTL*x*.TT_FC = 3'b011	Peripheral-to-peripheral with DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 8	
CTL*x*.SRC_TR_WIDTH = 3'b001	16 bit
CTL*x*.DST_TR_WIDTH = 3'b001	16 bit
CTL*x*.SRC_MSIZE = 3'b 001,	Source burst transaction of length 4
CTL*x*.DEST_MSIZE = 3'b 010	Destination burst transaction of length 8
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 16 bytes	–
CFG*x*.HS_SEL_SRC = 1	Source software handshaking
CFG*x*.HS_SEL_DST = 1	Destination software handshaking

All SSI parameters are prefixed with "SSI".

The following are some definitions of DW_apb_ssi parameters that are used in this example:

- SSI.TXFTLR: Transmit FIFO Threshold – Controls the level of entries (or below) at which the DW_apb_ssi transmit FIFO controller triggers an ssi_txe_intr interrupt.
- SSI.TXFLR: Transmit FIFO Level – Contains the number of valid data entries in the DW_apb_ssi transmit FIFO.
- SSI.IMR.TXEIM– Writing a 0 to this field masks the ssi_txe_intr interrupt.
- SSI.RXFTLR: Receive FIFO Threshold – Controls the level of entries (or below) at which the DW_apb_ssi transmit FIFO controller triggers an ssi_rxf_intr interrupt.
- SSI.RXFLR: Receive FIFO Level – Contains the number of valid data entries in the DW_apb_ssi transmit FIFO.
- SSI.IMR.RXFIM – Writing a 0 to this field masks the ssi_rxf_intr interrupt.

In this example:

SSI.TXFTLR = CTL*x*.DEST_MSIZE
SSI.RXFTLR + 1 = CTL*x*.SRC_MSIZE

The block transfer takes place on channel 0.

It is assumed that the transaction complete interrupts, IntDstTran and IntSrcTran, are enabled and unmasked at the beginning of the block transfer. For example:

MaskSrcTran[0] = 1
MaskDstTran[0] = 1
CTL*x*.INT_EN = 1

The block size is a multiple of the source burst transaction length:

blk_size_bytes_dma/src_burst_size_bytes = 16/8 = 2 = *integer*

It is also a multiple of the destination burst transaction length:

blk_size_bytes_dma/dst_burst_size_bytes 16/16 = 1 = *integer*

This block consists of two source burst transactions and one destination burst transaction. Neither the source nor destination enter the Single Transaction Region. The block transfer proceeds as follows:

1. SSI.TXFLR is initially equal to 0 so that an ssi_txe_intr interrupt is generated by the DW_apb_ssi. The Interrupt Service Routine (ISR) for this interrupt writes hex 0101 to both the ReqDstReg and SglReqDstReg registers; the order is not important. This generates a destination burst transaction request. Before exiting, this ISR software should write a 0 to the SSI.IMR.TXEIM register in order to mask any further ssi_txe_intr interrupts, since these are level-sensitive interrupts.
2. When the DW_apb_ssi receive FIFO contains greater than or equal to SSI.RXFTLR + 1 half-words, an ssi_rxf_intr interrupt is generated by the DW_apb_ssi. The Interrupt Service Routine (ISR) for this interrupt writes hex 0101 to both the ReqSrcReg and SglReqSrcReg registers; the order is not important. This generates a source burst transaction request. Before exiting this ISR, software should write a 0 to the SSI.IMR.RXFIM register in order to mask any further ssi_rxf_intr interrupts, since this is a level-sensitive interrupt.
3. Upon completion of the source burst transaction, the DW_ahb_dmac clears ReqSrcReg[0] and SglReqSrcReg[0], and generates an IntSrcTran interrupt. The ISR for this interrupt should write a 1 to the SSI.IMR.RXFIM register in order to unmask the generation of ssi_rxf_intr interrupts.
4. Same as step 2.
5. Same as step 3.
6. Upon completion of the destination burst transaction, the DW_ahb_dmac clears ReqDstReg[0] and SglReqDstReg[0], and generates an IntDstTran interrupt. The ISR for this interrupt should write a 1 to the SSI.IMR.TXEIM register to unmask the generation of ssi_rxf_intr interrupts.

Note

An alternative to using interrupts is for software to poll the DW_apb_ssi FIFO levels, SSI.TXFLR/SSI.RXFLR, until they equal CTL*x*.DEST_MSIZE/CTL*x*.SRC_MIZE in place of ssi_txe_intr/ssi_rxf_intr interrupts. Also, in place of IntSrcTran/IntDstTran interrupts, software could poll the ReqSrcReg[0]/ReqDstReg[0] registers until cleared by hardware.

2.9.1.10 Example 10

Scenario: This scenario demonstrates how to use software handshaking on both the source and destination side when DW_ahb_dmac is the flow controller and the source peripheral enters the Single Transaction Region. The DW_apb_ssi is used as the source and destination peripherals; for more information on the DW_apb_ssi, refer to the *DesignWare DW_apb_ssi Databook*.

This example uses the parameters listed in Table 2-15.

Table 2-15 Parameters in Transfer Operation – Example 10

Parameter	Description
CTL*x*.TT_FC = 3'b011	Peripheral-to-peripheral DW_ahb_dmac as flow controller
CTL*x*.BLOCK_TS = 12	–
CTL*x*.SRC_TR_WIDTH = 3'b00	16 bit
CTL*x*.DST_TR_WIDTH = 3'b001	16 bit
CTL*x*.SRC_MSIZE = 3'b 010	Source burst transaction of length 8
CTL*x*.DEST_MSIZE = 3'b 001	Destination burst transaction of length 4
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 8 bytes	–
CFG*x*.HS_SEL_SRC = 1	Source software handshaking
CFG*x*.HS_SEL_DST = 1	Destination software handshaking

The block size is not a multiple of the source burst transaction length:

blk_size_bytes_dma/src_burst_size_bytes = 24/16 != *integer*

Therefore, the block transfer from the source enters the Single Transaction Region near the end of a block.

The block size is a multiple of the destination burst transaction length:

blk_size_bytes_dma/dst_burst_size_bytes 24/8 = 3 = *integer*

Therefore, the block transfer to the destination consists of three burst transactions, and the destination does not enter the Single Transaction Region.

In this example:

SSI.TXFTLR = CTL*x*.DEST_MSIZE

2.9.1.10.1 ssi_rxf_intr Interrupt Service Routine (ISR)

1. Read the SSI.RXFTLR register.

   If SSI.RXFTLR = 0:

   a. Write hex 0101 to the SglReqSrcReg register.

   b. Write 7 (CTL*x*.SRC_MSIZE - 1) into the SSI.RXFTLR register, which triggers a new ssi_rxf_intr interrupt when the data items in the source FIFO are greater than or equal to 8.

   else (SSI.RXFTLR = 7)

   c. Write 0 to the SSI.IMR.RXFIM register in order to mask any further ssi_rxf_intr interrupts in the ISR for the ssi_rxf_intr interrupt.

   d. Write hex 0101 to the ReqSrcReg register.

2.9.1.10.2 IntSrcTran Interrupt Service Routine (ISR)

1. Write 0 into the SSI.RXFTLR register, which triggers an ssi_rxf_intr interrupt when a single data item is in the DW_apb_ssi receive FIFO.
2. Write a 1 to the SSI.IMR.RXFIM register in order to unmask the generation of ssi_rxf_intr interrupts.

The DMA block transfer might proceed as follows:

1. Software writes a value of 0 into the SSI.RXFTLR register, which triggers an ssi_rxf_intr interrupt when a single data item is in the DW_apb_ssi receive FIFO.
2. SSI.TXFLR is initially equal to 0, so an ssi_txe_intr interrupt is generated by DW_apb_ssi. The Interrupt Service Routine (ISR) for this interrupt writes hex 0101 to both the ReqDstReg and SglReqDstReg; the order is not important. This generates a destination burst transaction request. Before exiting this ISR, software should write a 0 to the SSI.IMR.TXEIM register in order to mask any further ssi_txe_intr interrupts, since this is a level-sensitive interrupt.
3. When the DW_apb_ssi receive FIFO contains one source data item, the DW_apb_ssi generates an ssi_rxf_intr interrupt. The ssi_rxf_intr ISR is called with SSI.RXFTLR = 0.
4. When the DW_apb_ssi receive FIFO contains half-words greater than or equal to SSI.RXFTLR + 1 = 8, an ssi_rxf_intr interrupt is generated by the DW_apb_ssi. The ssi_rxf_intr ISR is called with SSI.RXFTLR = 7, which generates a source burst transaction request, as the SglReqSrcReg has already been written to in step 3.
5. On completion of the destination burst transaction, the DW_ahb_dmac clears ReqDstReg[0] and SglReqDstReg[0], and generates an IntDstTran interrupt. The ISR for this interrupt should write a 1 to the SSI.IMR.TXEIM register in order to unmask the generation of ssi_txe_intr interrupts.
6. Same as step 2, except the ssi_txe_intr interrupt is generated when the DW_apb_ssi transmit FIFO drops to or below the watermark level.
7. Same as step 5.
8. Upon completion of the source burst transaction, the DW_ahb_dmac clears ReqSrcReg[0] and SglReqSrcReg[0], and generates an IntSrcTran interrupt.

Note The source peripheral has entered the Single Transaction Region, since there are only eight bytes left to complete the source block; however, *src_burst_size_bytes* = 16 bytes.

9. Same as 6.
10. Same as 3.
11. The DW_ahb_dmac performs a single transaction from the source. Upon completion of the source single transaction, the DW_ahb_dmac clears ReqSrcReg[0] and SglReqSrcReg[0], and generates an IntSrcTran interrupt.
12. Steps 10 and 11, performed three more times. The block transfer from the source is now complete.
13. Same as step 7. The block transfer to the destination is now complete.

Note

The block transfer could proceed by polling the SSI.TXFLR/SSI.RXFLR registers in place of ssi_txe_intr/ssi_rxf_intr interrupts. Also, in place of IntSrcTran/IntDstTran interrupts, software could poll the ReqSrcReg[0]/ReqDstReg[0] registers.

When the ssi_rxf_intr ISR has no knowledge of when the source peripheral is inside or outside the Single Transaction Region, as for this example, then the ssi_rxf_intr ISR has to be invariant to this. If the ssi_rxf_intr ISR knows when the Single Transaction Region has been entered, then it can dynamically adjust the threshold level that triggers a source burst transaction, as explained in "Example 4" on page 60. The source block then completes on an Early-Terminated Burst Transaction. The ssi_rxf_intr and IntSrcTran Interrupt Service Routines in this case would be:

2.9.1.10.3 ssi_rxf_intr Interrupt Service Routine (ISR)

1. Write hex 0101 to the ReqSrcReg register, followed by a write of hex 0101 to the SglReqSrcReg register. This generates a source burst transaction request.
2. Before exiting this ISR, software should write a 0 to the SSI.IMR.RXFIM register in order to mask any further ssi_rxf_intr interrupts, since this is a level-sensitive interrupt.

2.9.1.10.4 IntSrcTran Interrupt Service Routine (ISR)

1. If the source has entered the Single Transaction Region after this burst transaction, then the ISR writes a value of 3 into the SSI.RXFTLR register. This triggers an ssi_rxf_intr interrupt when four data items are in the DW_apb_ssi receive FIFO.
2. Write a 1 to the SSI.IMR.TXEIM register in order to unmask the generation of ssi_txe_intr interrupts.

Note

Knowing CTL*x*.BLOCK_TS and the number of source burst transactions completed, or by reading CTL*x*.BLOCK_TS – which reads the total number of data items read from the source peripheral up to this time – software can calculate when the amount of data left to fetch in the source block transfer is less than CTL*x*.SRC_MSIZE. Therefore, software can calculate when the source peripheral has entered the Single Transaction Region.

2.9.1.11 Example 11

Scenario: This scenario demonstrates how to use software handshaking on both the source and destination side when the source peripheral is the flow controller and the destination peripheral enters the Single Transaction Region. The DW_apb_ssi is used as the destination peripheral; for more information on the DW_apb_ssi, refer to the *DesignWare DW_apb_ssi Databook*. This example uses the parameters listed in Table 2-16.

Table 2-16 Parameters in Transfer Operation – Example 11

Parameter	Description
CTL*x*.TT_FC = 3'b101	Peripheral-to-peripheral with source as flow controller
CTL*x*.BLOCK_TS = *x*	–
CTL*x*.SRC_TR_WIDTH = 3'b001	16 bit

Parameter	Description
CTL*x*.DST_TR_WIDTH = 3'b001	16 bit
CTL*x*.SRC_MSIZE = 3'b 001,	Source burst transaction length = 4
CTL*x*.DEST_MSIZE = 3'b 010	Destination burst transaction length = 8
CFG*x*.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CH*x*_FIFO_DEPTH = 16 bytes	–
CFG*x*.HS_SEL_SRC = 1	Source software handshaking
CFG*x*.HS_SEL_DST = 1	Destination software handshaking

In this example, it is assumed that the source peripheral generates an interrupt when the FIFO level is greater than or equal to some watermark level. For the purposes of this example, assume that the watermark level that triggers an interrupt named *src_burst_intr* is equal to CTL*x*.SRC_MSIZE. Also assume that this interrupt is level-sensitive and can be masked by writing to a software register in the source peripheral.

Consider the case where the source block is made up of a three-burst transaction:

blk_size_bytes_src = 3 * *src_burst_size_bytes* = 3 * 8 = 24 bytes

The block size is not a multiple of the destination burst transaction length:

blk_size_bytes_src/dst_burst_size_bytes 24/16 != *integer*

Therefore, the block transfer to the destination enters the Single Transaction Region near the end of the block.

Attention

Block transfer when *one* of the following is true:

- It is guaranteed that data at some point will be extracted from the destination FIFO in the "Single transaction region" in order to trigger a burst transaction.
- When the watermark level that triggers a burst transaction to the destination can be dynamically adjusted near the end of block.

The DMA block transfer might proceed as follows:

1. Program SSI.TXFTLR = CTL*x*.DEST_MSIZE
2. SSI.TXFLR is initially 0, so an ssi_txe_intr interrupt is generated by the DW_apb_ssi. The Interrupt Service Routine (ISR) for this interrupt writes hex 0101 to both the ReqDstReg and SglReqDstReg; the order is not important. This generates a destination burst transaction request. Before exiting this ISR, software should write a 0 to the SSI.IMR.TXEIM register in order to mask any further ssi_txe_intr interrupts, since these are level-sensitive interrupts.
3. The source peripheral generates an interrupt when the watermark level is reached or exceeded. The Interrupt Service Routine (ISR) for this interrupt writes hex 0100 to the SglReqSrcReg and LstSrcReg

registers, followed by hex 0101 to the ReqSrcReg. This generates a source burst transaction request, which is not the last in the block. Software should now mask the *src_burst_intr* interrupt.

4. Upon completion of the source burst transaction, the DW_ahb_dmac clears ReqSrcReg[0] and generates an IntSrcTran interrupt. The ISR for this interrupt should unmask the generation of the *src_burst_intr* interrupt in the source peripheral.
5. Repeat step 3.
6. Repeat step 4.
7. Upon completion of the destination burst transaction, the DW_ahb_dmac clears ReqDstReg[0] and SglReqDstReg[0], and generates an IntDstTran interrupt. The ISR for this interrupt should write a 1 to the SSI.IMR.TXEIM register in order to unmask generation of ssi_rxf_intr interrupts.
8. The source peripheral generates an interrupt when the watermark level is reached or exceeded. The Interrupt Service Routine (ISR) for this interrupt writes hex 0100 to the SglReqSrcReg register, followed by hex 0101 to the LstSrcReg register, followed by hex 0101 to the ReqSrcReg register. This generates a source burst transaction request, which is the last in the block. Software should now mask the *src_burst_intr* interrupt.
9. Repeat 4. The block transfer from the source is now complete.
10. The block transfer to the destination may proceed as follows:
    a. If it is guaranteed that data at some point will be extracted from the destination FIFO near the end of a block in order to trigger a burst transaction, an ssi_txe_intr interrupt is generated by the destination DW_apb_ssi peripheral. The Interrupt Service Routine (ISR) for this interrupt writes hex 0101 to the ReqDstReg register, followed by a write hex 0101 to the SglReqDstReg; the order *is* important. This generates a destination burst transaction request, which is an Early-Terminated Burst Transaction. Before exiting this ISR, software should write a 0 to the SSI.IMR.TXEIM register in order to mask any further ssi_txe_intr interrupts, since these are level-sensitive interrupts. The block transfer to the destination is now complete.
    b. If it is *not* guaranteed that data at some point will be extracted from the destination FIFO near the end of a block in order to trigger a burst transaction, and if software can determine that the destination has entered the Single Transaction Region, then it can write a value of 4 into the SSI.TXFTLR register. This triggers an ssi_txe_intr interrupt when four free locations are present in the DW_apb_ssi transmit FIFO. On receipt of this interrupt, software can then write hex 0101 to the ReqDstReg register, followed by a write of hex 0101 to the SglReqDstReg register; the order is important. This generates a destination burst transaction request, which is an Early-Terminated Burst Transaction. Hardware clears ReqDstReg[0] and SglReqDstReg[0] upon block completion, which occurs after four data items have been transferred to the destination in this burst transaction, and an IntDstTran interrupt is generated. The block transfer to the destination is now complete.

Note

Software can determine when it has entered the Single Transaction Region, since it knows that a last-burst transaction has been requested by the source, and can calculate the number of bytes left to complete in the destination block. It can then compare this to *dst_burst_size_bytes.* For more information, refer to “Single Transaction Region” on page 38.

Attention

Block transfer when *both* of the following are true:

- It is *not* guaranteed that data at some point will be extracted from the destination FIFO in the "Single transaction region" in order to trigger a burst transaction.
- When the watermark level that triggers a burst transaction to the destination *cannot* be dynamically adjusted near the end of block.

The ssi_txe_intr and IntDstTran Interrupt Service Routines in this case would be:

- ssi_txe_intr Interrupt Service Routine (ISR)
  a. Read the SSI.TXFTLR register.
     If SSI.TXFTLR = 1
     i. Write hex 0101 to the SglReqDstReg register.
     ii. Write 8 (CTL*x*.DEST_MSIZE) to the SSI.TXFTLR register. This will trigger a new ssi_txe_intr interrupt when the number of data items in the destination FIFO is less than or equal to 8.
     else (SSI.TXFTLR = 8)
     i. Write 0 to the SSI.IMR.TXFIM register in order to mask any further ssi_txe_intr interrupts in the ISR for the ssi_txe_intr interrupt.
     ii. Write hex 0101 to the ReqDstReg register.
- IntDstTran Interrupt Service Routine (ISR)
  a. Write 1 to the SSI.TXFTLR register, which triggers an ssi_txe_intr interrupt when a single free location is available in the DW_apb_ssi transmit FIFO.
  b. Write a 1 to the SSI.IMR.TXFIM register in order to unmask the generation of ssi_txe_intr interrupts.

The DMA block transfer might proceed as follows:

1. Software writes a value of 1 into the SSI.TXFTLR register, which triggers an ssi_txe_intr interrupt when a single free location is in the DW_apb_ssi transmit FIFO.
2. SSI.TXFLR is initially 0, so an ssi_txe_intr interrupt is generated by the DW_apb_ssi. The ssi_txe_intr ISR is called when SSI.TXFTLR = 1.
3. SSI.TXFLR is initially 0, so an ssi_txe_intr interrupt is generated by the DW_apb_ssi. The ssi_txe_intr ISR is called with SSI.TXFTLR = 8. A destination burst transaction request is generated, as the SglReqDstReg register has been written to in step 2.
4. The source peripheral generates an interrupt when the watermark level is reached or exceeded. The Interrupt Service Routine (ISR) for this interrupt writes hex 0100 to the SglReqSrcReg and LstSrcReg registers, followed by a write of hex 0101 to the ReqSrcReg. This generates a source burst transaction request, which is not the last in the block. Software should now mask the *src_burst_intr* interrupt.
5. Upon completion of the source burst transaction, the DW_ahb_dmac clears ReqSrcReg[0] and generates an IntSrcTran interrupt. The ISR for this interrupt should unmask generation of the *src_burst_intr* interrupt in the source peripheral.
6. Repeat step 4.

7. Repeat step 5.
8. Upon completion of the destination burst transaction, the DW_ahb_dmac clears ReqDstReg[0] and SglReqDstReg[0], and generates an IntDstTran interrupt.
9. The source peripheral generates an interrupt when the watermark level is reached or exceeded. The Interrupt Service Routine (ISR) for this interrupt writes hex 0100 to the SglReqSrcReg register, followed by a write of hex 0101 to the LstSrcReg register, followed by a write of hex 0101 to the ReqSrcReg register. This generates a source burst transaction request, which is the last in the block. Software should now mask the *src_burst_intr* interrupt.
10. Repeat step 5. The block transfer from the source is now complete.
11. When a free location exists in the destination FIFO, an ssi_txe_intr interrupt is generated by the DW_apb_ssi. The ssi_txe_intr ISR is called with SSI.TXFTLR = 1.
12. The DW_ahb_dmac performs a single transaction to the destination. Upon completion of the destination single transaction, the DW_ahb_dmac clears ReqDstReg[0] and SglReqDstReg[0], and generates an IntDstTran interrupt.
13. Repeat steps 11 and 12 four more times. The block transfer to the destination is now complete.

Note "Example 11" on page 83 outlines a DW_ahb_dmac block transfer using software handshaking when interrupts are used to control the block transfer. The same could be achieved using register polling.

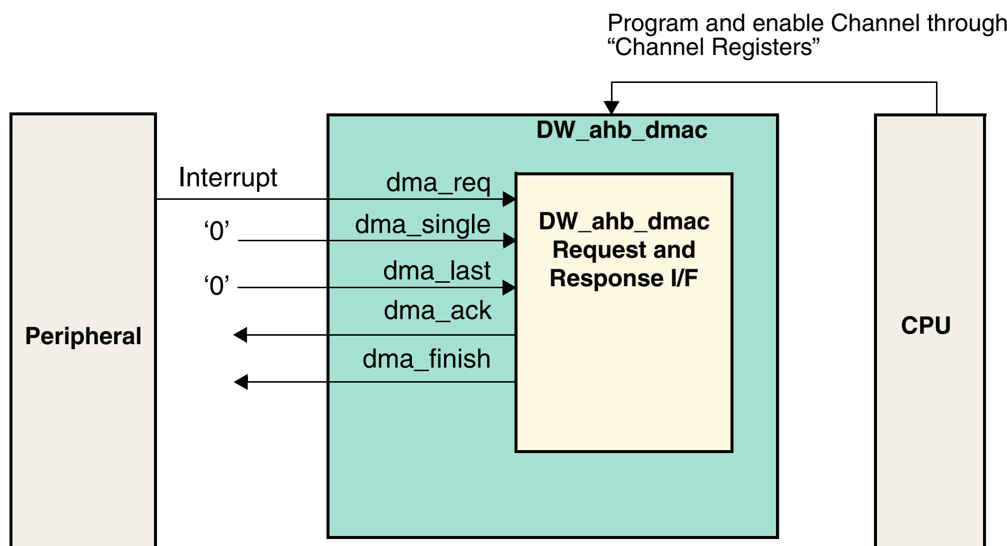
2.9.2 Peripheral Interrupt Request Interface

The interface illustrated in Figure 2-33 is a simplified version of the hardware handshaking interface. In this mode:

- The interrupt line from the peripheral is tied to the dma_req input.
- The dma_single input is tied low.
- All other interface signals are ignored.

This interface can be used where the slave peripheral does not have hardware handshaking signals. To the DW_ahb_dmac, this is the same "Hardware Handshaking – Peripheral Is Not Flow Controller" on page 40.

Figure 2-33 Transaction Request Through Peripheral Interrupt



The peripheral can never be the flow controller, since it cannot connect to the dma_last signal. The interrupt line from the peripheral is tied to the dma_req line, as shown in Figure 2-33. The timing of the interrupt line from the peripheral must be the same as the dma_req line, as discussed in "Hardware Handshaking – Peripheral Is Not Flow Controller" on page 40.

Since the dma_ack line is not sampled by the peripheral, the handshaking loop is as follows:

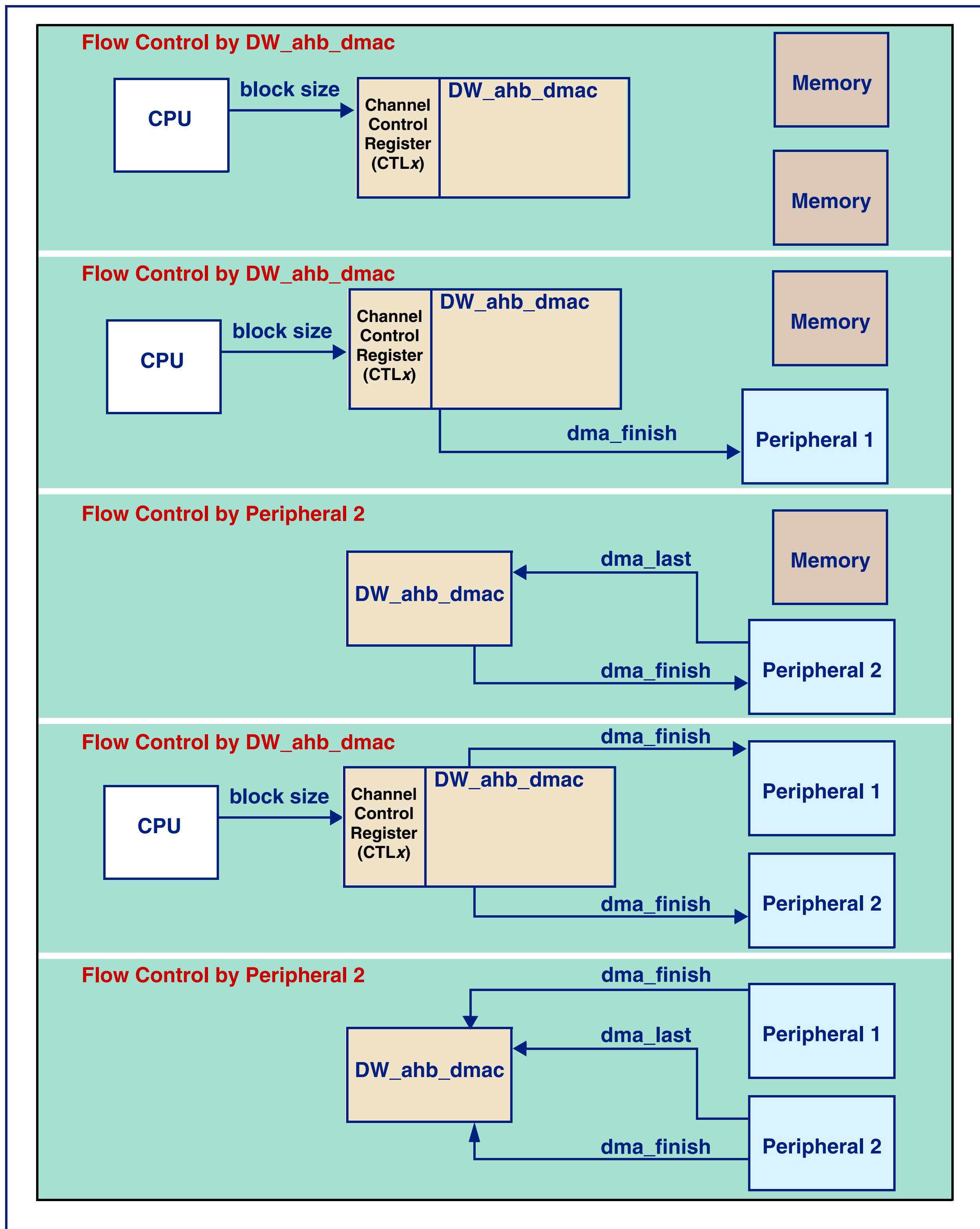
1. Peripheral generates an interrupt that asserts dma_req.
2. DW_ahb_dmac completes the burst transaction and generates an end-of-burst transaction interrupt, IntSrcTran/IntDstTran. Interrupts must be enabled and the transaction complete interrupt unmasked.
3. The interrupt service routine clears the interrupt in the peripheral so that the dma_req is de-asserted.

Notice that dma_single is hardcoded to an inactive level. For conditions where the source/destination peripheral can tie dma_single to an inactive level, refer to "Single Transactions – Peripheral Is Not Flow Controller" on page 47.

2.10 Flow Control Configurations

Figure 2-34 indicates five different flow control configurations using hardware handshaking interfaces – a simplified version of the interface is shown. These scenarios can also be used for software handshaking, which uses software registers instead of signals.

Figure 2-34 Flow Control Configurations



2.11 Peripheral Burst Transaction Requests

For a source FIFO, an active edge is triggered on dma_req when the source FIFO exceeds some watermark level. For a destination FIFO, an active edge is triggered on dma_req when the destination FIFO drops below some watermark level.

This section investigates the optimal settings of these watermark levels on the source and destination peripherals and their relationship to, respectively:

- Source transaction length, CTL*x*.SRC_MSIZE
- Destination transaction length, CTL*x*.DEST_MSIZE

For demonstration purposes, a receive DW_apb_ssi is used as a source peripheral, and a transmit DW_apb_ssi is used as a destination peripheral. For more information on the DW_apb_ssi, refer to the *DesignWare DW_apb_ssi Databook*.

Note Throughout this section, DW_apb_ssi-related parameters are prefixed with "SSI." DW_ahb_dmac-related parameters are prefixed with "DMA."

2.11.1 Transmit Watermark Level and Transmit FIFO Underflow

During DW_apb_ssi serial transfers, DW_apb_ssi transmit FIFO requests are made to the DW_ahb_dmac whenever the number of entries in the DW_apb_ssi transmit FIFO is less than or equal to the DW_apb_ssi Transmit Data Level Register (SSI.DMATDLR) value. This is known as the watermark level. The DW_ahb_dmac responds by writing a burst of data to the DW_apb_ssi transmit FIFO buffer, of length DMA.CTL*x*.DEST_MSIZE.

Data should be fetched from the DW_ahb_dmac often enough for the DW_apb_ssi transmit FIFO to continuously perform serial transfers; that is, when the DW_apb_ssi transmit FIFO begins to empty, another burst transaction request should be triggered. Otherwise the DW_apb_ssi transmit FIFO runs out of data (underflow). To prevent this condition, you must set the watermark level correctly.

2.11.2 Choosing the Transmit Watermark Level

Consider an example with the following assumption:

DMA.CTL*x*.DEST_MSIZE = SSI_TX_FIFO_DEPTH - SSI.DMATDLR

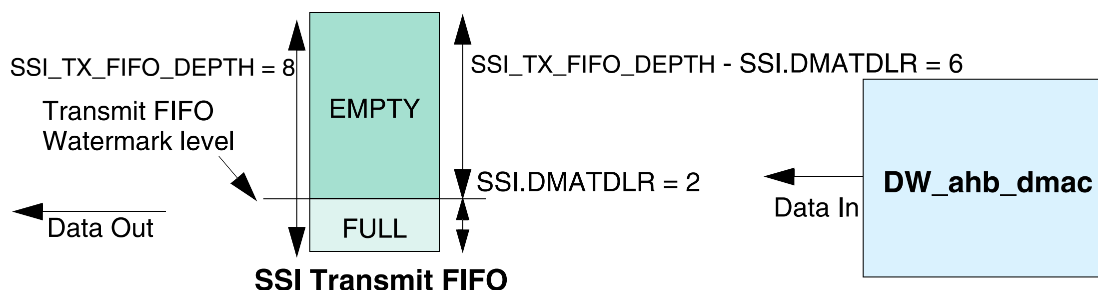
Note SSI_TX_FIFO_DEPTH is the DW_apb_ssi transmit FIFO depth. SSI.DMATDLR controls the level at which a DW_ahb_dmac destination burst request is made by the DW_apb_ssi transmit logic. It is equal to the watermark level; that is, a destination burst request is generated (active-edge of dma_req triggered) when the number of valid data entries in the DW_apb_ssi transmit FIFO is equal to or below this field value.

In this situation, the number of data items to be transferred in a DW_ahb_dmac burst is equal to the empty space in the DW_apb_ssi transmit FIFO. Consider two different watermark level settings.

2.11.2.1 Case 1: SSI.DMATDLR = 2

Figure 2-35 illustrates the watermark levels in Case 1 where SSI.DMATDLR = 2.

Figure 2-35 Case 1 Watermark Levels where SSI.DMATDLR = 2



Case 1 uses the parameters listed in Table 2-17.

Table 2-17 Transmit Watermark Level – Case 1

Parameter	Comment
SSI.DMATDLR = 2	DW_apb_ssi transmit FIFO watermark level
DMA.CTL*x*.DEST_MSIZE = SSI_TX_FIFO_DEPTH - SSI.DMATDLR = 6	DMA.CTL*x*.DEST_MSIZE is equal to the empty space in the transmit FIFO at the time the burst request is made.
SSI _TX_FIFO_DEPTH = 8	DW_apb_ssi transmit FIFO depth
DMA.CTL*x*.BLOCK_TS = 30	Block size

The number of burst transactions that are needed equals the block size divided by the number of data items per burst:

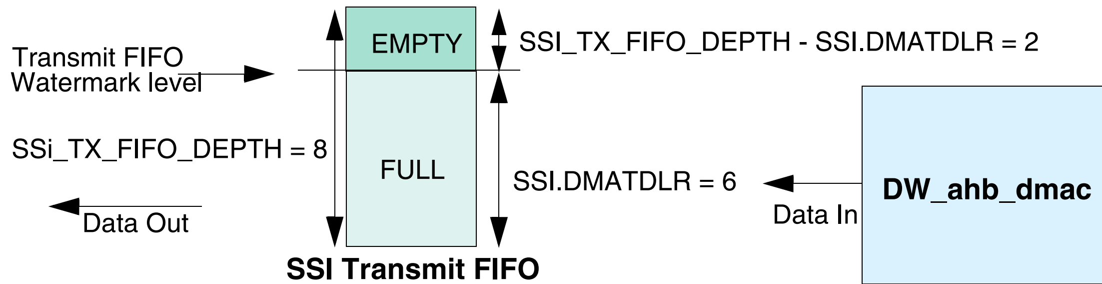
DMA.CTL*x*.BLOCK_TS/DMA.CTL*x*.DEST_MSIZE = 30/6 = 5

The number of burst transactions in the DW_ahb_dmac block transfer is 5, but the watermark level, SSI.DMATDLR, is quite low. Therefore, the probability of an SSI underflow is high where the SSI serial transmit line needs to transmit data, but where there is no data left in the transmit FIFO. This occurs because the DW_ahb_dmac has not had time to service the DW_ahb_dmac request before the DW_apb_ssi transmit FIFO becomes empty.

2.11.2.2 Case 2: SSI.DMATDLR = 6

Figure 2-36 illustrates the watermark levels in Case 2 where SSI.DMATDLR = 6.

Figure 2-36 Case 2 Watermark Levels where SSI.DMATDLR = 6



Case 2 uses the parameters listed in Table 2-18.

Table 2-18 Transmit Watermark Level – Case 2

Parameter	Description
SSI.DMATDLR = 6	DW_apb_ssi transmit FIFO watermark level
DMA.CTL*x*.DEST_MSIZE = SSI_TX_FIFO_DEPTH - SSI.DMATDLR = 2	DMA.CTL*x*.DEST_MSIZE is equal to the empty space in the transmit FIFO at the time the burst request is made.
SSI_TX_FIFO_DEPTH = 8	DW_apb_ssi transmit FIFO depth
DMA.CTL*x*.BLOCK_TS = 30	Block size

The number of burst transactions in the block are:

DMA.CTL*x*.BLOCK_TS/DMA.CTL*x*.DEST_MSIZE = 30/2 = 15

In this block transfer, there are fifteen destination burst transactions in a DMA block transfer, but the watermark level, SSI.DMATDLR, is high. Therefore, the probability of an SSI underflow is low because the DW_ahb_dmac has plenty of time to service the destination burst transaction request before the SSI transmit FIFO becomes empty.

Thus, the second case has a lower probability of underflow at the expense of more burst transactions per block. This potentially provides a greater amount of bursts per block and a worse bus utilization than the former case.

Therefore, the goal in choosing a watermark level is to minimize the number of transactions per block, while at the same time keeping the probability of an underflow condition to an acceptable level. In practice, this is a function of the following ratio:

$$\frac{\text{Rate of SSI data transmission}}{\text{Rate of DW_ahb_dmac response to destination burst requests}}$$

For example, promoting the channel to the highest-priority channel in the DW_ahb_dmac, and promoting the DW_ahb_dmac master interface to the highest-priority master in the AHB layer, increases the rate at

which the DW_ahb_dmac can respond to burst transaction requests. This in turn allows you to decrease the watermark level, which improves bus utilization without compromising the probability of an underflow occurring.

2.11.3 Selecting CTL*x*.DEST_MSIZE and Transmit FIFO Overflow

As can be seen from Figure 2-36, programming DMA.CTL*x*.DEST_MSIZE to a value greater than the watermark level that triggers the DW_ahb_dmac request may cause overflow when there is not enough space in the DW_apb_ssi transmit FIFO to service the destination burst request. Therefore, the following equation must be adhered to in order to avoid overflow:

DMA.CTL*x*.DEST_MSIZE <= SSI_TX_FIFO_DEPTH - SSI.DMATDLR(13)

In "Case 2: SSI.DMATDLR = 6" on page 92, the amount of space in the transmit FIFO at the time the burst request is made is equal to the destination burst length, DMA.CTL*x*.DEST_MSIZE. Thus, the transmit FIFO may be full, but not overflowed, upon completion of the burst transaction.

Therefore, for optimal operation, DMA.CTL*x*.DEST_MSIZE should be set at the FIFO level that triggers a transmit DW_ahb_dmac request; that is:

DMA.CTL*x*.DEST_MSIZE = SSI_TX_FIFO_DEPTH - SSI.DMATDLR(14)

This is the setting used in Figure 2-24.

Adhering to equation (14) reduces the number of DW_ahb_dmac bursts needed for a block transfer, and this in turn improves AHB bus utilization.

The DW_apb_ssi transmit FIFO is not full at the end of a DW_ahb_dmac burst transaction if the SSI has successfully transmitted one data item or more on the SSI serial transmit line before the end of the burst transaction.

2.11.4 Receive Watermark Level and Receive FIFO Overflow

During DW_apb_ssi serial transfers, DW_apb_ssi receive FIFO requests are made to the DW_ahb_dmac whenever the number of entries in the DW_apb_ssi receive FIFO is at or above the DW_ahb_dmac Receive Data Level Register; that is, SSI.DMARDLR+1. This is known as the watermark level. The DW_ahb_dmac responds by reading a burst of data from the receive FIFO buffer of length DMA.CTL*x*.SRC_MSIZE.

SSI.DMARDLR controls the level at which a source burst request is made by the receive logic. When the number of valid data entries in the DW_apb_ssi receive FIFO is equal to or greater than the watermark level (DMARDL+1), a source burst request is generated (active-edge of dma_req triggered).

Data should be fetched by the DW_ahb_dmac often enough for the DW_apb_ssi receive FIFO to accept serial transfers continuously; that is, when the DW_apb_ssi receive FIFO begins to fill, another burst transaction request should be triggered. Otherwise, the DW_apb_ssi receive FIFO fills with data (overflow). To prevent this condition, you must correctly set the watermark level.

2.11.5 Choosing the Receive Watermark level

Similar to choosing the transmit watermark level described earlier, the receive watermark level, SSI.DMARDLR+1, should be set to minimize the probability of overflow, as shown in Figure 2-37 on page 94. It is a trade-off between the number of burst transactions required per block versus the probability of an overflow occurring.

2.11.6 Selecting CTL*x*.SRC_MSIZE and Receive FIFO Underflow

As can be seen in Figure 2-37, programming a source burst transaction length greater than the watermark level may cause underflow when there is not enough data to service the source burst request. Therefore, equation (15) below must be adhered to in order to avoid underflow.

If the number of data items in the DW_apb_ssi receive FIFO is equal to the source burst length at the time the source burst request is made – DMA.CTL*x*.SRC_MSIZE – the DW_apb_ssi receive FIFO may be emptied, but not underflowed, at the completion of the source burst transaction. For optimal operation, DMA.CTL*x*.SRC_MSIZE should be set at the watermark level; that is:

DMA.CTL*x*.SRC_MSIZE = SSI.DMARDLR + 1 **(15)**

Adhering to equation (15) reduces the number of burst transactions in a block transfer, and this in turn can improve AHB bus utilization.

Note

The DW_apb_ssi receive FIFO is not empty at the end of the source burst transaction if the SSI has successfully received one data item or more on the SSI serial receive line before the end of the burst, as illustrated in Figure 2-37.

Figure 2-37 SSI Receive FIFO

2.12 Generating Requests for the AHB Master Bus Interface

Each channel has a source state machine and destination state machine running in parallel. These state machines generate the request inputs to the arbiter, which arbitrates for the master bus interface (one arbiter per master bus interface).

When the source/destination state machine is granted control of the master bus interface, and when the master bus interface is granted control of the external AHB bus, then AHB transfers between the peripheral and the DW_ahb_dmac (on behalf of the granted state machine) can take place.

AHB transfers from the source peripheral or to the destination peripheral cannot proceed until the channel FIFO is ready. For burst transaction requests and for transfers involving memory peripherals, the criterion for “FIFO readiness” is controlled by the FIFO_MODE field of the CFGx register.

The definition of FIFO readiness is the same for:

- Single transactions
- Burst transactions, where CFG*x*.FIFO_MODE = 0
- Transfers involving memory peripherals, where CFG*x*.FIFO_MODE = 0

The channel FIFO is deemed ready when the space/data available is sufficient to complete a single AHB transfer of the specified transfer width. FIFO readiness for source transfers occurs when the channel FIFO contains enough room to accept at least a single transfer of CTL*x*.SRC_TR_WIDTH width. FIFO readiness for destination transfers occurs when the channel FIFO contains data to form at least a single transfer of CTL*x*.DST_TR_WIDTH width.

Note An exception to FIFO readiness for destination transfers occurs in "FIFO flush mode." In this mode, FIFO readiness for destination transfers occurs when the channel FIFO contains data to form at least a single transfer of CTL*x*.SRC_TR_WIDTH width (and not CTL*x*.DST_TR_WIDTH width, as is the normal case). For an explanation of FIFO flush mode, refer to "Example 5" on page 63.

When CFG*x*.FIFO_MODE = 1, then the criteria for FIFO readiness for burst transaction requests and transfers involving memory peripherals are as follows:

- A FIFO is ready for a source burst transfer when the FIFO is less than half empty.
- A FIFO is ready for a destination burst transfer when the FIFO is greater than or equal to half full.

Exceptions to this "readiness" occur. During these exceptions, a value of CTLx.FIFO_MODE =0 is assumed. The following are the exceptions:

- Near the end of a burst transaction or block transfer – The channel source state machine does not wait for the channel FIFO to be less than half empty if the number of source data items left to complete the source burst transaction or source block transfer is less than DMAH_CH*x*_FIFO_DEPTH/2. Similarly, the channel destination state machine does not wait for the channel FIFO to be greater than or equal to half full, if the number of destination data items left to complete the destination burst transaction or destination block transfer is less than DMAH_CH*x*_FIFO_DEPTH/2.
- In FIFO flush mode – For an explanation of FIFO flush mode, refer to "Example 5" on page 63.
- When a channel is suspended – The destination state machine does not wait for the FIFO to become half empty to flush the FIFO, regardless of the value of the FIFO_MODEfield.
- After receipt of a split/retry response from a source or destination – The AMBA protocol requires that after an AHB master receives a split/retry response, it must re-issue the transfer that received the split/retry before attempting any other transfer. Therefore, a transfer is re-issued to the same address that returned the split/retry, regardless of FIFO_MODE, when the DW_ahb_dmac is next granted the AHB bus. This is repeated until an OKAY response is received on the AHB hresp bus.

When the source/destination peripheral is not memory, the source/destination state machine waits for a single/burst transaction request. Upon receipt of a transaction request and only if the channel FIFO is "ready" for source/destination AHB transfers, a request for the master bus interface is made by the source/destination state machine.

Note

There is one exception to this, which occurs when the destination peripheral is the flow controller and CFG*x*.FCMODE = 1 (data pre-fetching is disabled). Then the source state machine does not generate a request for the master bus interface (even if the FIFO is "ready" for source transfers and has received a source transaction request) until the destination requests new data. Refer to "Example 8" on page 76.

When the source/destination peripheral is memory, the source/destination state machine must wait until the channel FIFO is "ready." A request is then made for the master bus interface. There is no handshaking mechanism employed between a memory peripheral and the DW_ahb_dmac.

2.12.1 Locked DMA Transfers

It is possible to program the DW_ahb_dmac for:

- Bus locking – Asserts the AHB hlock signal.
- Channel locking – Locks the arbitration for the AHB master interface, which grants ownership of the master bus interface to one of the requesting channel state machines (source or destination).

Bus and channel locking can proceed for the duration of a DMA transfer, a block transfer, or a single or burst transaction.

2.12.1.1 Bus Locking

If the LOCK_Bbit in the channel configuration register (CFGx) is set, then the AHB hlock signal is asserted for the duration specified in the LOCK_B_L field.

2.12.1.2 Channel Locking

If the LOCK_CH field is set, then the arbitration for the master bus interface is exclusively reserved for the source and destination peripherals of that channel for the duration specified in the LOCK_CH_L field.

If bus locking is activated for a certain duration, then it follows that the channel is also automatically locked for that duration. Three cases arise:

- CFG*x*.LOCK_B = 0 – Programmed values of CFG*x*.LOCK_CH and CFG*x*.LOCK_CH_L are used.
- CFG*x*.LOCK_B = 1 and CFG*x*.LOCK_CH = 0 – DMA transfer proceeds as if CFG*x*.LOCK_CH = 1 and CFG*x*.LOCK_CH_L = CFG*x*.LOCK_B_L. The programmed values of CFG*x*.LOCK_CH and CFG*x*.LOCK_CH_L are ignored.
- CFG*x*.LOCK_B = 1 and CFG*x*.LOCK_CH = 1 – Two cases arise:
  - CFG*x*.LOCK_B_L <= CFG*x*.LOCK_CH_L – In this case, the DMA transfer proceeds as if CFG*x*.LOCK_CH_L = CFG*x*. LOCK_B_L and the programmed value of CFG*x*.LOCK_CH_L is ignored. Thus, if bus locking is enabled over the DMA transfer level, then channel locking is enabled over the DMA transfer level, regardless of the programmed value of CFG*x*.LOCK_CH_L
  - CFG*x*.LOCK_B_L > CFG*x*.LOCK_CH_L – The programmed value of CFG*x*.LOCK_CH_L is used. Thus, if bus locking is enabled over the DMA block transfer level and channel locking is enabled over the DMA transfer level, then channel locking is performed over the DMA transfer level.

2.12.1.3 Locking Levels

If locking is enabled for a channel, then locking of the AHB master bus interface at a programmed locking transfer level is activated when the channel is first granted the AHB master bus interface at the start of that locking transfer level. It continues until the locking transfer level has completed; that is, if channel 0 has enabled channel level locking at the block transfer level, then this channel locks the master bus interface when it is first granted the master bus interface at the start of the block transfer, and continues to lock the master bus interface until the block transfer has completed.

Source and destination block transfers occur successively in time, and a new source block cannot commence until the previous destination block has completed. When both source and destination are on the same AHB layer, then block level locking is terminated on completion of the block to the destination. If they are on separate layers, then block-level locking is terminated on completion of the block on that layer—when the source block on the source AHB layer completes, and when the destination block on the destination AHB layer completes. The same is true for DMA transfer-level locking.

Transaction-level locking is different due to the fact that source and destination transactions occur independently in time, and the number of source and destination transactions in a DMA block or DMA transfer do not have to match. When the source and destination are on the same AHB layer, then transaction-level locking is cleared at the end of a source or destination transaction only if the opposing peripheral is not currently in the middle of a transaction.

For example, if locking is enabled at the transaction level and an end-of-source transaction is signaled, then this disables locking only if one of the following is true:

- The destination is on a different AHB layer
- The destination is on the same AHB layer, but the channel is not currently in the middle of a transaction to the destination peripheral.

The same rules apply when an end-of-destination transaction is signalled.

If channel-level or bus-level locking is enabled for a channel at the transaction level, and either the source or destination of the channel is a memory device, then the locking is ignored and the channel proceeds as if locking (bus or channel) is disabled.

Note Since there is no notion of a transaction level for a memory peripheral, then transaction-level locking is not allowed when either source or destination is memory.

2.12.1.4 Channel Locking and Deadlock

Certain combinations of channel-level and bus-level locking may lead to deadlock, where multiple channels are concurrently enabled and no channel can proceed with the DMA transfer. This occurs only for configurations where DMAH_NUM_MASTER_INT > 1 and DMAH_NUM_CHANNELS > 1. The methods used to avoid deadlock are described in more detail in "Channel Locking and Deadlock" on page 409.

2.13 Arbitration for AHB Master Interface

Each DW_ahb_dmac channel has two request lines that request ownership of a particular master bus interface: channel source and channel destination request lines.

Source and destination arbitrate separately for the bus. Once a source/destination state machine gains ownership of the master bus interface and the master bus interface has ownership of the AHB bus, then AHB transfers can proceed between the peripheral and DW_ahb_dmac. Figure 2-38 illustrates the arbitration flow of the master bus interface.

An arbitration scheme decides which of the request lines (2 * DMAH_NUM_CHANNELS) is granted the particular master bus interface. Each channel has a programmable priority. A request for the master bus interface can be made at any time, but is granted only after the current AHB transfer (burst or single) has completed. Therefore, if the master interface is transferring data for a lower priority channel and a higher priority channel requests service, then the master interface will complete the current burst for the lower priority channel before switching to transfer data for the higher priority channel.

To prevent a channel from saturating the master bus interface, it can be given a maximum AMBA burst length (MAX_ABRST field in CFGx register) at channel setup time. This also prevents the master bus interface from saturating the AHB bus where the system arbiter cannot change the grant lines until the end of an undefined length burst.

The following is the interface arbitration scheme employed when no channel has locked (Channel Locking) the arbitration for the master bus interface:

- If only one request line is active at the highest priority level, then the request with the highest priority wins ownership of the AHB master bus interface; it is not necessary for the priority levels to be unique.

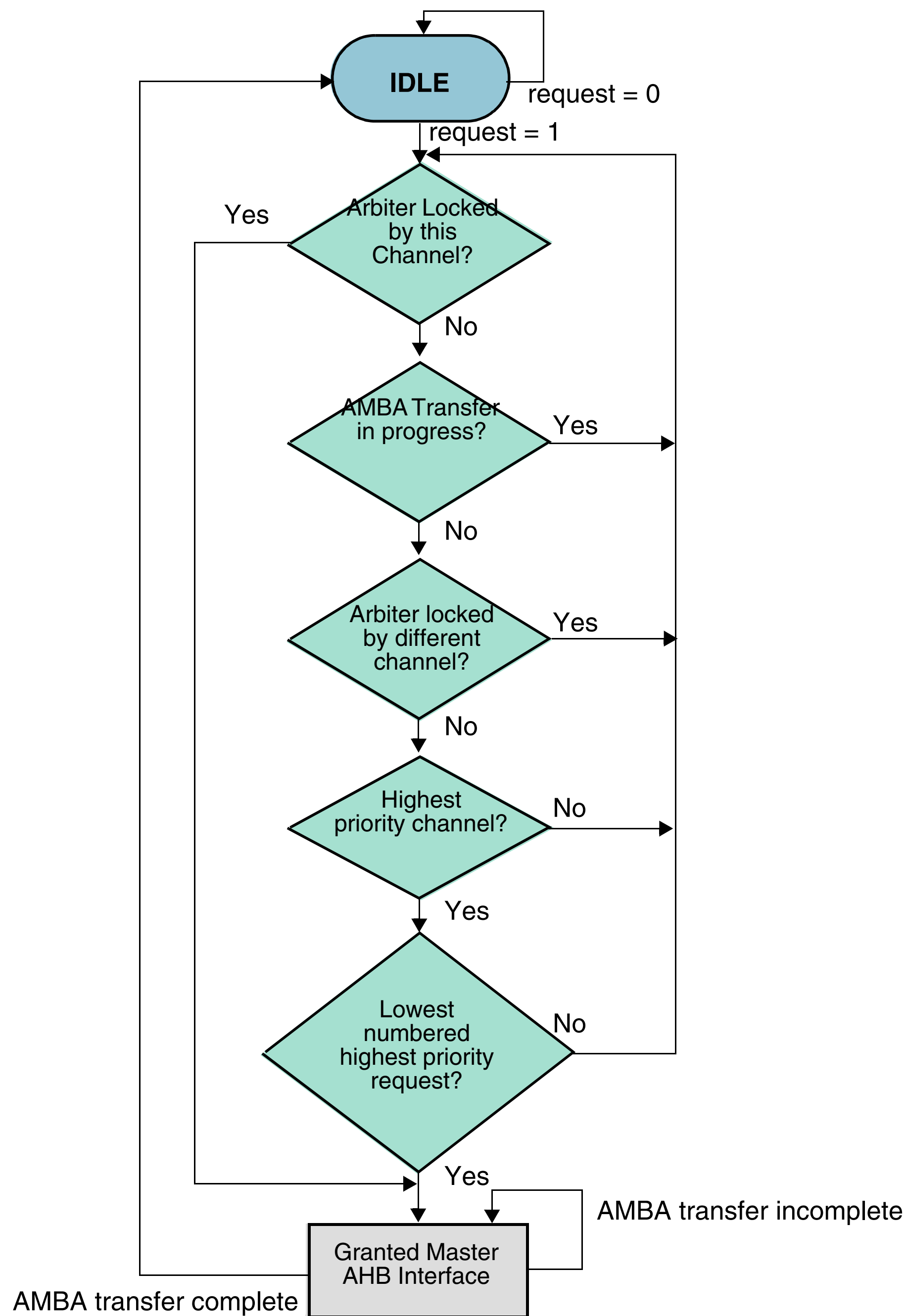
  If more than one request is active at the highest requesting priority, then these competing requests proceed to a second tier of arbitration.

- If equal priority requests occur, then the lower-numbered channel is granted.

  In other words, if a peripheral request attached to Channel 7 and a peripheral request attached to Channel 8 have the same priority, then the peripheral attached to Channel 7 is granted first.

A channel source is granted before the destination if both have their request lines asserted when a grant decision is made. A channel source and channel destination inherit their channel priority and therefore always have the same priority.

Figure 2-38 Arbitration Flow for Master Bus Interface



2.14 Latency

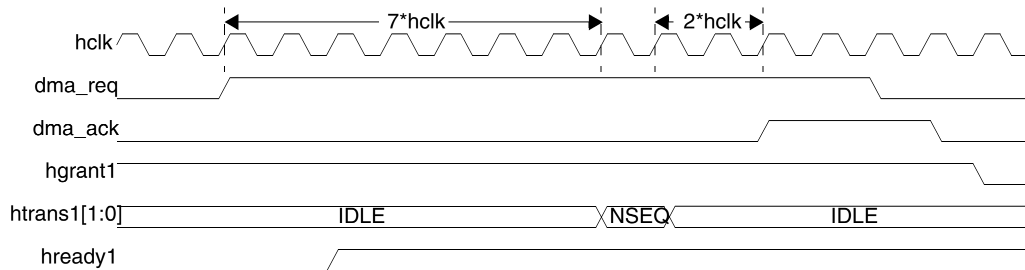
By default, the DW_ahb_dmac has:

- Seven cycles of latency from hardware handshaking request to NSEQ being issued from appropriate AHB master interface
- Two cycles of latency from completion of AHB burst to assertion of dma_ack

Note The following latency illustrations are best-case scenarios where the DW_ahb_dmac master interface already owns the AHB bus before issuing the transfer. AHB bus arbitration latency increases the observed latency.

Figure 2-39 shows the default latency from a hardware handshaking request to the AHB transfer, and the latency from the completion of that burst to the assertion of dma_ack.

Figure 2-39 Default DW_ahb_dmac Latency

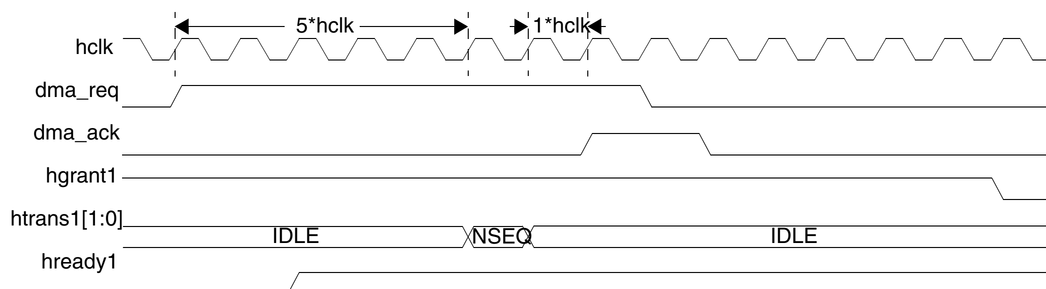


This default latency can be reduced by setting the parameter DMAH_REMOVE_PIPELINING to 1, which reduces the latency to:

- Five cycles from hardware handshaking request to NSEQ
- One cycle from AHB burst completion to dma_ack.

Figure 2-40 shows the latency from a hardware handshaking request to the AHB transfer, and the latency from the completion of that burst to the assertion of dma_ack; in this case, DMAH_REMOVE_PIPELINING is set to 1.

Figure 2-40 DW_ahb_dmac Latency with DMAH_REMOVE_PIPELINING Set to 1



Setting DMAH_REMOVE_PIPELINING to 1 also does the following:

- Results in longer logic paths in the DW_ahb_dmac, which can make it harder to reach the desired operating frequency.
- Reduces latency by one cycle when the hardware handshaking interfaces are not being used. One of the pipeline stages removed is in the master interface arbiter, which is used for all types of transfer.

2.15 Scatter/Gather

Scatter is relevant to a destination transfer. The destination address is incremented or decremented by a programmed amount – the scatter increment – when a scatter boundary is reached. Figure 2-41 shows an example destination scatter transfer. The destination address is incremented or decremented by the value stored in the destination scatter increment (DSR*x*.DSI) field (refer to "DSRx"), multiplied by the number of bytes in a single AHB transfer to the destination š (decoded value of CTLx.DST_TR_WIDTH)/8 – when a scatter boundary is reached. The number of destination transfers between successive scatter boundaries is programmed into the Destination Scatter Count (DSC) field of the DSR*x* register.

Scatter is enabled by writing a 1 to the CTL*x*.DST_SCATTER_EN field. The CTL*x*.DINC field determines if the address is incremented, decremented, or remains fixed when a scatter boundary is reached. If the CTL*x*.DINC field indicates a fixed-address control throughout a DMA transfer, then the CTL*x*.DST_SCATTER_EN field is ignored, and the scatter feature is automatically disabled.

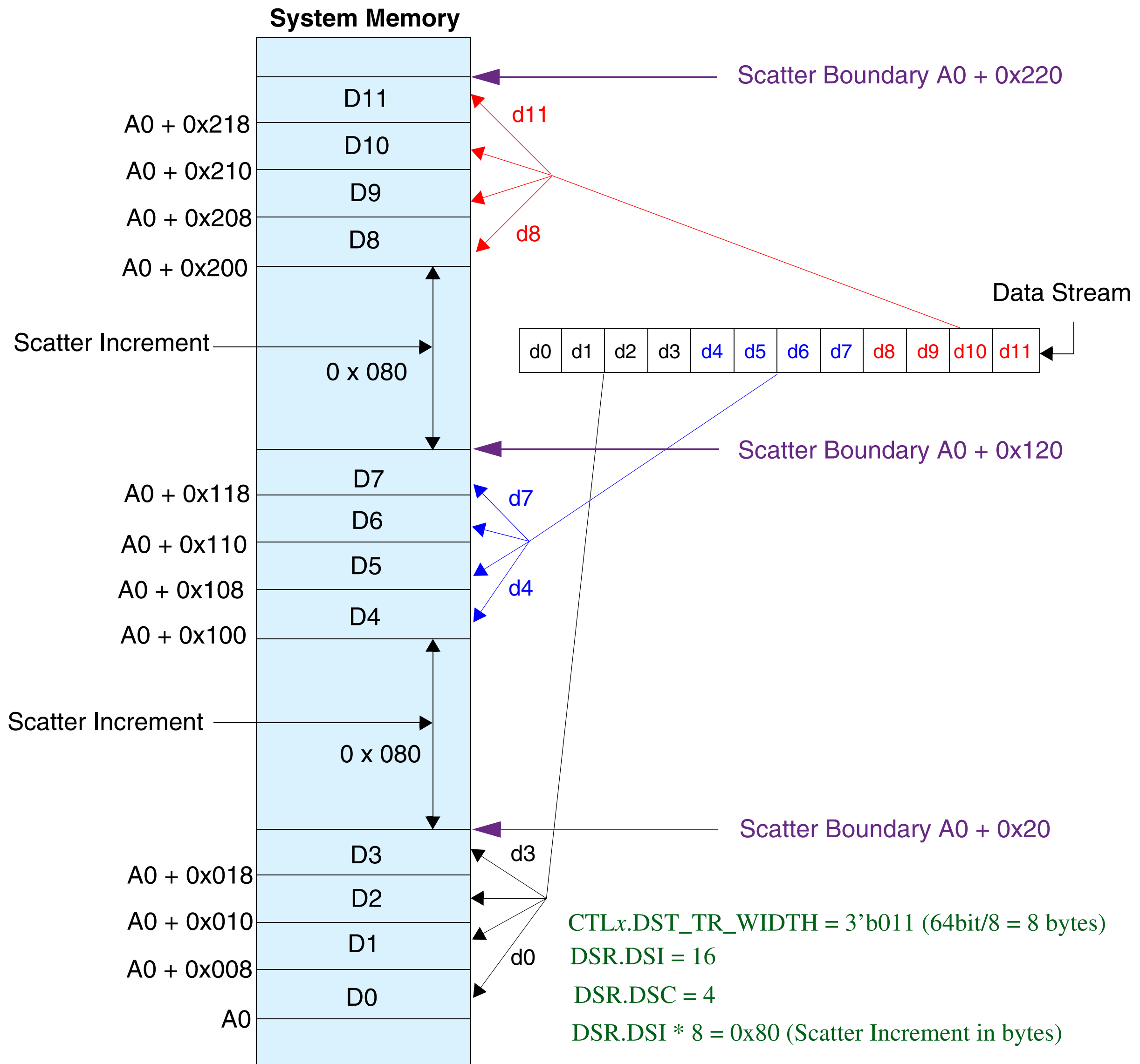
Gather is relevant to a source transfer. The source address is incremented or decremented by a programmed amount when a gather boundary is reached. The number of source transfers between successive gather boundaries is programmed into the Source Gather Count (SGR*x*.SGC) field. The source address is incremented or decremented by the value stored in the source gather increment (SGR*x*.SGI) field (refer to "SGRx"), multiplied by the number of bytes in a single AHB transfer from the source – (decoded value of CTLx.SRC_TR_WIDTH)/8 – when a gather boundary is reached.

Gather is enabled by writing a 1 to the CTLx.SRC_GATHER_EN field. The CTL*x*.SINC field determines if the address is incremented, decremented, or remains fixed when a gather boundary is reached. If the CTLx.SINC field indicates a fixed-address control throughout a DMA transfer, then the CTLx.SRC_GATHER_EN field is ignored, and the gather feature is automatically disabled.

Note For multi-block transfers, the counters that keep track of the number of transfers left to reach a gather/scatter boundary are re-initialized to the source gather count (SGR*x*.SGC) and destination scatter count (DSC), respectively, at the start of each block transfer.

Figure 2-41 shows an example of a destination scatter transfer:

Figure 2-41 Example of Destination Scatter Transfer



As an example of gather increment, consider the following:

SRC_TR_WIDTH = 3'b 010 (32 bit)

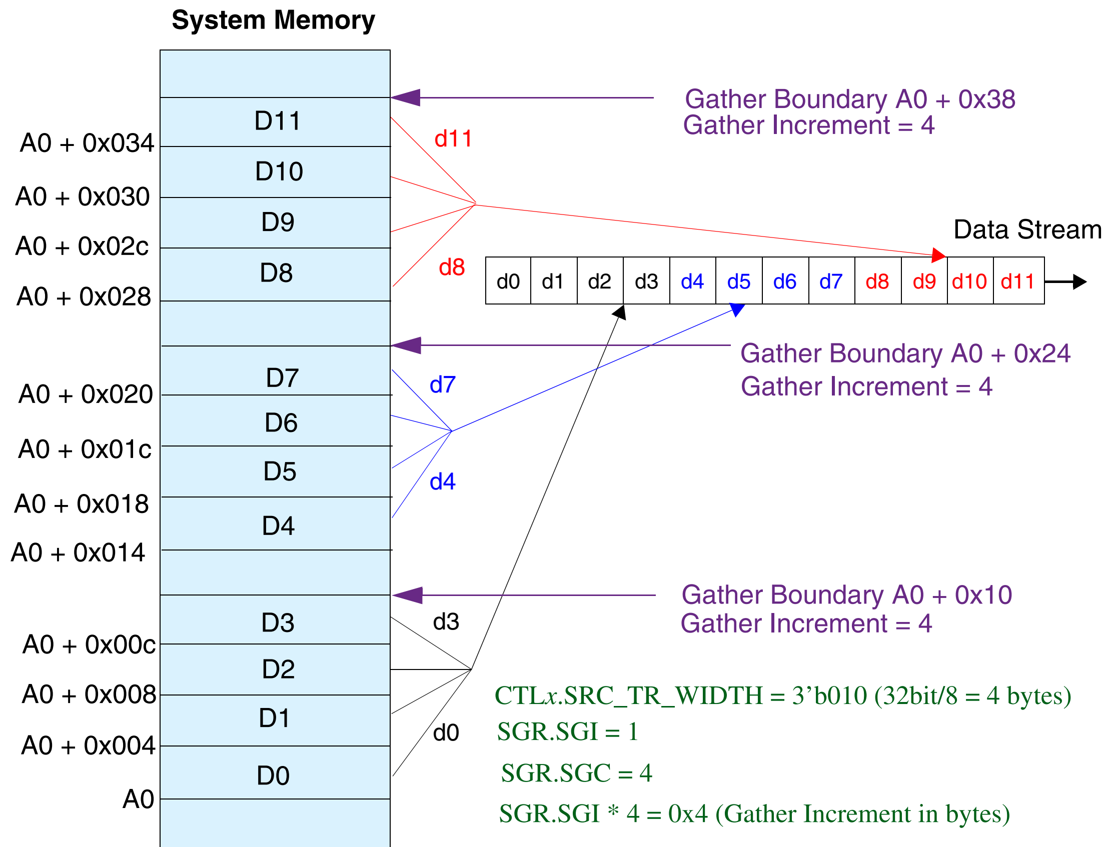
SGR.SGC = 0*x*04 (source gather count)

CTL*x*.SRC_GATHER_EN = 1 (source gather enabled)

SAR*x* = A0 (starting source address)

Figure 2-42 shows a source gather when SGR.SGI = 0x01.

Figure 2-42 Source Gather when SGR.SGI = 0x1



In general, if the starting address is A0 and CTL*x*.SINC = 2'b00 (increment source address control), then the transfer will be:

> A0, AO + TWB, A0 + 2*TWB ....(A0 + (SGR.SGC-1)*TWB)
> <-scatter_increment-> (A0 + (SGR.SGC*TWB) + (SGR.SGI *TWB))

where TWB is the transfer width in bytes, decoded value of CTL*x*.SRC_TR_WIDTH/8 = *src_single_size_bytes*.

2.16 Endianness

DW_ahb_dmac supports Little Endian, Address Invariant (AI) Big Endian scheme and BE-32 (Word Invariant) scheme of data access on AHB slave interface and each AHB master interface. Endian format can be either hard coded using coreConsultant configuration parameter or selected using a I/O pins (signal).

2.16.1 Big Endian-Little Endian Conversion Logic

The behavior of DW_ahb_dmac for Big Endian-Little Endian conversion using BE-32 (Word Invariant) method is as shown in the following figures.

Figure 2-43 BE-LE Conversion Using BE-32 (Word Invariant) Method for 32-Bit Data Bus

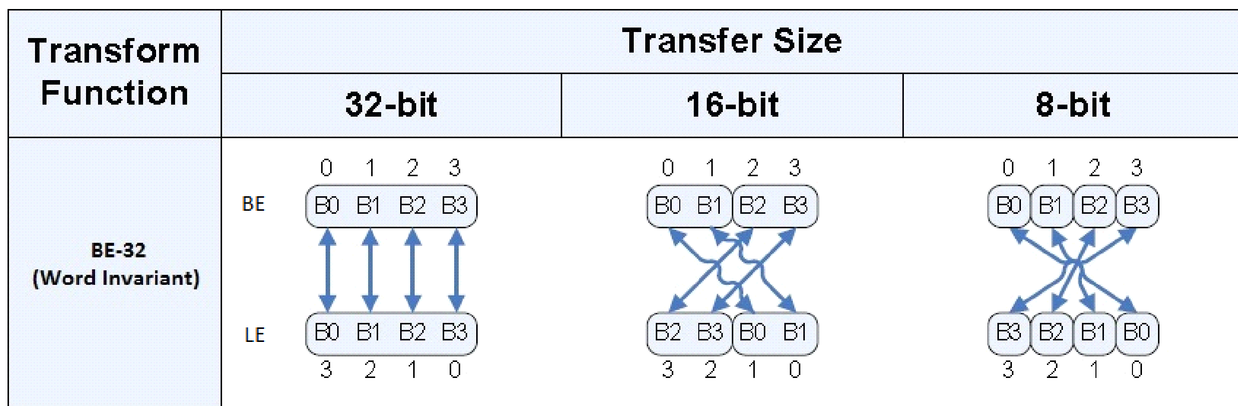


Figure 2-44 BE-LE Conversion Using BE-32 (Word Invariant) Method for 54-Bit Data Bus – Word Access

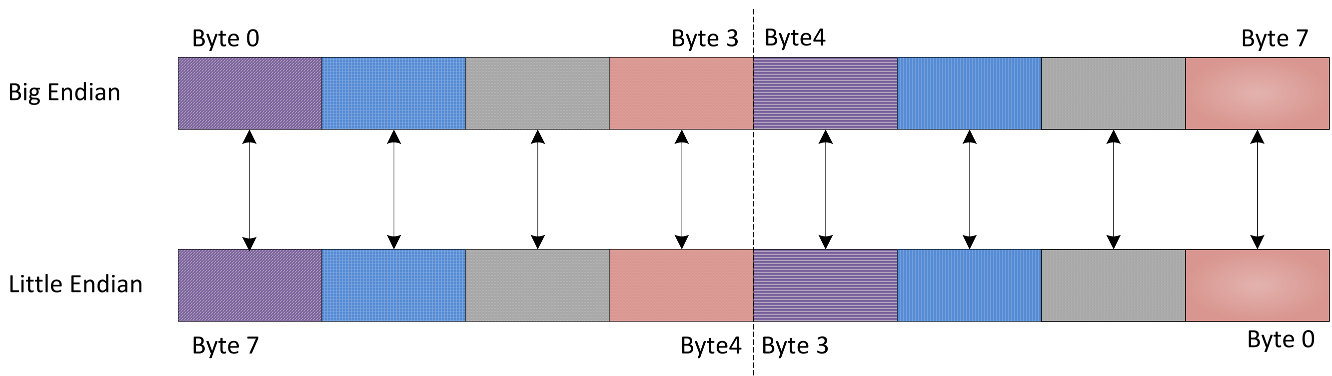


Figure 2-45 BE-LE Conversion Using BE-32 (Word Invariant) Method for 64-Bit Data Bus – Half Word Access

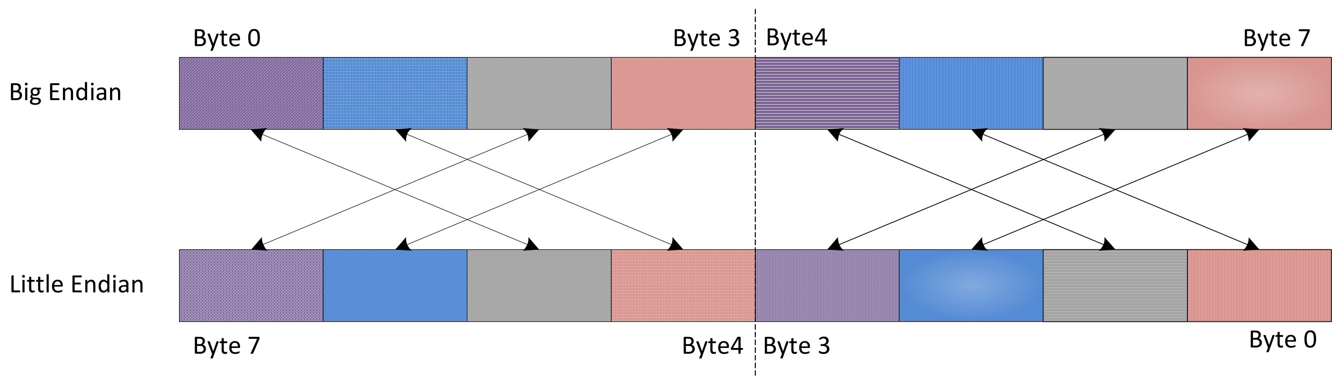
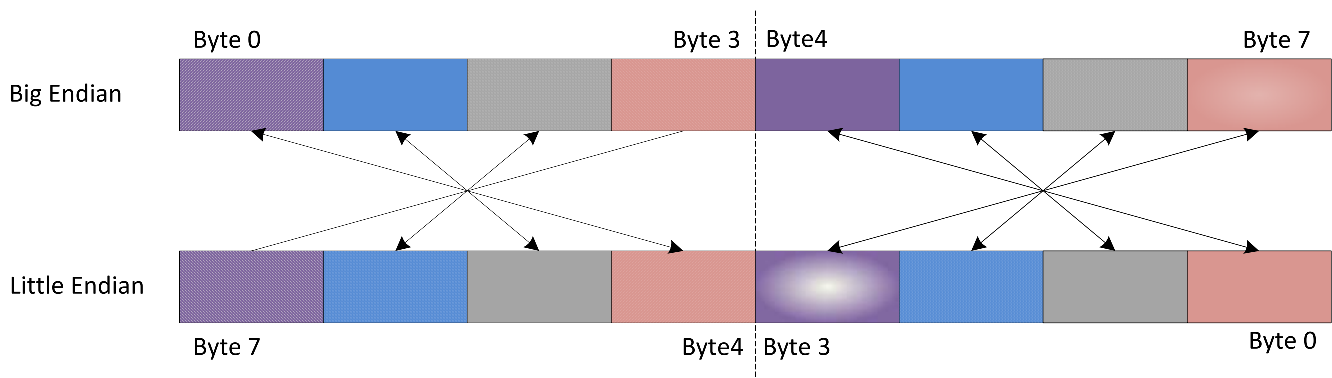


Figure 2-46 BE-LE Conversion Using BE-32 (Word Invariant) Method for 64-Bit Data Bus – Byte Access



2.16.2 LLI Fetch, and Status and Control Write-Back

The DW_ahb_dmac can be programmed to use the Little Endian scheme whenever it fetches a Linked List Item (LLI) or writes Status and Control register data back to the LLI location, regardless of the endian format selected for the source and destination data transfer on the particular AHB master interface.

2.16.3 Endian Selection

The endian scheme used for the register access, for source and destination data transfer on master interfaces and LLI fetch, and for Status and Control write-back access on master interfaces is decided by the values of the coreConsultant parameters and the I/O signals. All possible combinations of endian scheme are captured in Table 2-19.

Table 2-19 Endian Scheme Selection116

<table>
<tr><th>DMAH_STATIC_ENDIAN_SELECT</th><th>DMAH_BIG_ENDIAN</th><th>dma_big_endian_slv/m1/m2/m3/m4</th><th>DMAH_BE32_SELECTION_PIN_EN</th><th>dma_be_select_be32_slv/m1/m2/m3/m4</th><th>Endian Scheme Used for Register Access on Slave Interface and SRC/DST Data Access on M1/M2/M3/M4 Master Interface</th><th>DMAH_LLI_ENDIAN_SELECTION_PIN_EN</th><th>dma_le_select_lli_m1/m2/m3/m4</th><th>Endian Scheme Used for LLI Fetch/ Status and Control Write Back Access on M1/M2/M3/M4 Master Interface</th></tr>
<tr><td>1</td><td>0</td><td>X</td><td>X</td><td>X</td><td>Little Endian</td><td>X</td><td>X</td><td>Little Endian</td></tr>
<tr><td rowspan="3">1</td><td rowspan="3">1</td><td rowspan="3">X</td><td rowspan="3">0</td><td rowspan="3">X</td><td rowspan="3">Big Endian AI</td><td>0</td><td>X</td><td>Big Endian AI</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian AI</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
<tr><td rowspan="6">1</td><td rowspan="6">1</td><td rowspan="6">X</td><td rowspan="6">1</td><td rowspan="3">0</td><td rowspan="3">Big Endian AI</td><td>0</td><td>X</td><td>Big Endian AI</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian AI</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
<tr><td rowspan="3">1</td><td rowspan="3">Big Endian BE-32</td><td>0</td><td>X</td><td>Big Endian BE-32</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian BE-32</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
<tr><td>0</td><td>X</td><td>0</td><td>X</td><td>X</td><td>Little Endian</td><td>X</td><td>X</td><td>Little Endian</td></tr>
<tr><td rowspan="3">0</td><td rowspan="3">X</td><td rowspan="3">1</td><td rowspan="3">0</td><td rowspan="3">X</td><td rowspan="3">Big Endian AI</td><td>0</td><td>X</td><td>Big Endian AI</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian AI</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
<tr><td rowspan="6">0</td><td rowspan="6">X</td><td rowspan="6">1</td><td rowspan="6">1</td><td rowspan="3">0</td><td rowspan="3">Big Endian AI</td><td>0</td><td>X</td><td>Big Endian AI</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian AI</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
<tr><td rowspan="3">1</td><td rowspan="3">Big Endian BE-32</td><td>0</td><td>X</td><td>Big Endian BE-32</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>Big Endian BE-32</td></tr>
<tr><td>1</td><td>Little Endian</td></tr>
</table>

The endian scheme used for the Status and Control write-back is the same as defined by the DMAH_LLI_ENDIAN_SELECTION_PIN_EN parameter and dma_le_select_lli_m*N* input signal, where *N* = 1/2/3/4 for the corresponding master interface even when LLP_SRC_EN = LLP_DST_EN = 0 for the corresponding channel. This is because Status and Control write-back happens to the address location derived from the base address defined by LLP*x*.LOC value (base address of LLI).

2.16.4 Static Endian Configuration

You can statically configure endianness under the following conditions:

- Endianness of the AHB slave interface and all configured AHB master interfaces are the same
- Endianness is known at system configuration time

Under these circumstances, you do the following:

1. Set the DMAH_STATIC_ENDIAN_SELECT parameter to True.
2. Configure the DMAC_BIG_ENDIAN parameter to either big-endian or little-endian.
3. Configure the DMAH_BE32_SELECTION_PIN_EN parameter and drive the appropriate I/O to either big-endian AI or big-endian BE-32

2.16.5 Dynamic Endian Configuration

You should dynamically configure endianness if either of the following conditions exists:

- Endianness of the AHB slave interface and all configured master interfaces are not all the same.
- Endianness of the slave interface or any master interface is not known at system configuration time.

Under either of these circumstances, do the following:

1. Set the DMAH_STATIC_ENDIAN_SELECT parameter to False.
2. Drive the appropriate I/O pin to either big-endian (AI or BE-32) or little-endian.

For example, if you were to drive dmah_big_endian_m*N* to 1, where *N* is the master number, configure the DMAH_BE32_SELECTION_PIN_EN parameter to 1 and drive dma_be_select_be32_m*N* to 1, where *N* is the master number. The AMBA layer to which that master interface is attached is Big Endian BE-32. Conversely, if you were to drive dmah_big_endian_m*N* to 0, that master interface would be Little Endian.

The dma_big_endian_slv signal works in the same way for the slave interface.

2.17 AHB Transfer Error Handling

Upon occurrence of an error in an AHB transfer, the following occurs:

1. DMA transfer in progress stops immediately
2. Relevant channel is disabled,
3. An interrupt is issued (if not masked)

If multiple channels are enabled, only the one where the AHB error is detected is disabled.

The contents of the FIFO are not cleared, but they become inaccessible and are overwritten once the channel is re-enabled to start a new sequence.

There is no support for automatically resuming the transfer from the point where the error occurred, and the full block transfer has to be re-initiated in order to be successfully completed.

The DMA does not use the hardware handshaking interface to signal the error occurrence in any way, nor does it signal the end of a transfer. In practice, this means that if a request from a peripheral is active when the error occurs—dma_req is high if peripheral is the flow controller; dma_req or dma_single are high if peripheral is not the flow controller—the channel is disabled without the DMA ever asserting dma_ack (or dma_finish).

The hardware handshake interface on the peripheral side has to be re-initiated by the CPU upon detection of the error interrupt. The dma_req signal needs to be brought low before the channel is re-enabled and then brought high when the channel has been enabled.

2.18 Last Beat of DMA Burst Indication

DW_ahb_dmac provides an additional output signal (dma_wlast) on the AHB interface to indicate the last write data during burst transfers to destination peripherals. The dma_wlast signal is available only when the parameter DMAH_WLAST_EN is set.

The dma_wlast signal is an active high signal that is asserted on completion of the last address phase of every destination MSIZE data transfer, similar to the wlast signal in the AXI protocol. The signal remains asserted until the associated last dataphase is complete. This signal is available on all AHB interfaces when the parameter DMAH_WLAST_EN is set.

The following example explains the behavior of the dma_wlast signal in different cases.

2.18.1 Example 1

Scenario: Example block transfer when the DW_ahb_dmac is the flow controller. The table lists the DMA parameters for this example.

Table 2-20 Parameters Used in Transfer Operation - Example 1

Parameter	Description
CTLx.TT_FC = 3'b011	Peripheral to peripheral transfer with DW_ahb_dmac as flow controller
CTLx.BLOCK_TS = 16	-
CTLx.SRC_TR_WIDTH = 3'b010	32 bit
CTLx.DST_TR_WIDTH = 3'b010	32 bit
CTLx.SRC_MSIZE = 3'b 010	Source burst transaction length = 8
CTLx.DEST_MSIZE = 3'b 010	Destination burst transaction length = 8
CFGx.MAX_ABRST = 1'b 0	No limit on maximum AMBA burst length
DMAH_CHx_FIFO_DEPTH = 64 bytes	-

Using equation (5), a total of 64 bytes is transferred in the block; that is, blk_size_bytes_dma = 64. This block transfer consists of two destination MSIZE data transfers of length 8 beats. The first MSIZE data transfer consists of two INCR4 AHB bursts. The second MSIZE data transfer consists of a single INCR8 AHB burst. The dma_wlast signal is asserted at the end of every destination MSIZE data transfer as shown in Figure 2-47.

The dma_wlast signal indicates the last data beat of destination MSIZE data transfer only when the destination is peripheral. This signal is not applicable when the destination is memory.

Figure 2-47 Assertion of the dma_wlast Signal

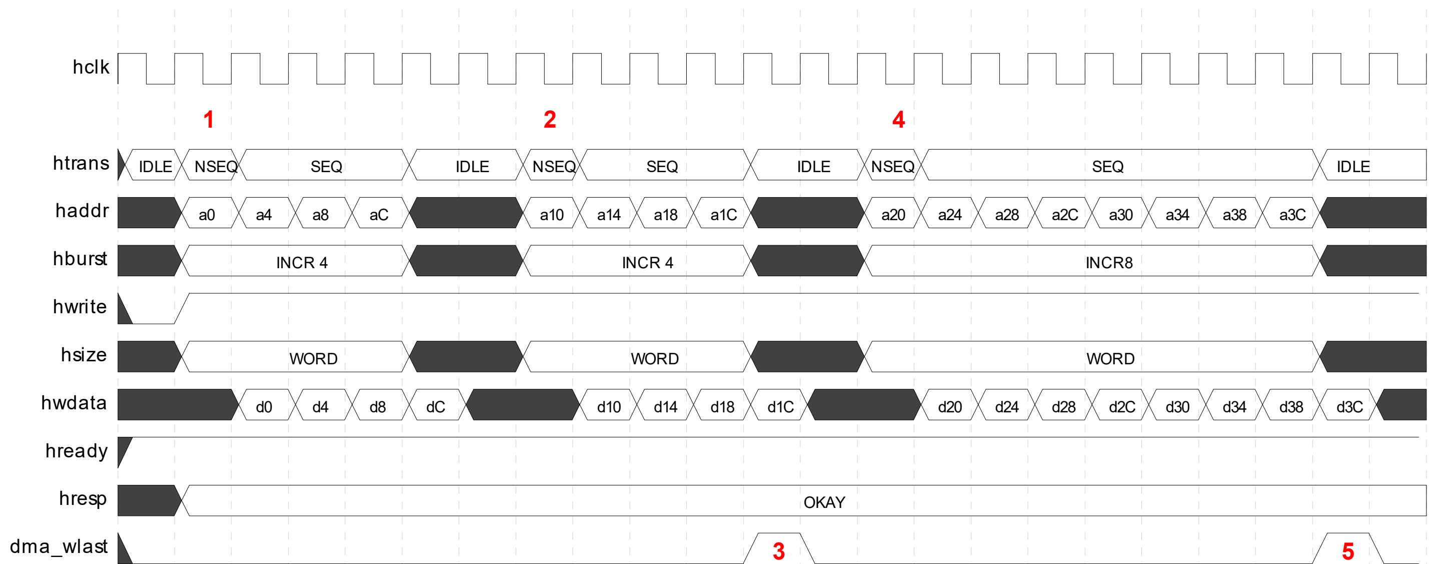


Figure 2-47 shows the following:

1. The first INCR4 AHB burst transfer of the first MSIZE data transfer.
2. The second INCR4 AHB burst transfer of the first MSIZE data transfer.
3. The signal dma_wlast asserted at the end of the first MSIZE data transfer.
4. A single INCR8 AHB burst ransfer of the second MSIZE data transfer.
5. The dma_wlast signal asserted at the end of the second MSIZE data transfer.

The dma_wlast signal is asserted regardless of HREADY and is therefore pulse extended if HREADY is low, as shown in Figure 2-48.

Figure 2-48 Behavior of the dma_wlast Signal During Wait State

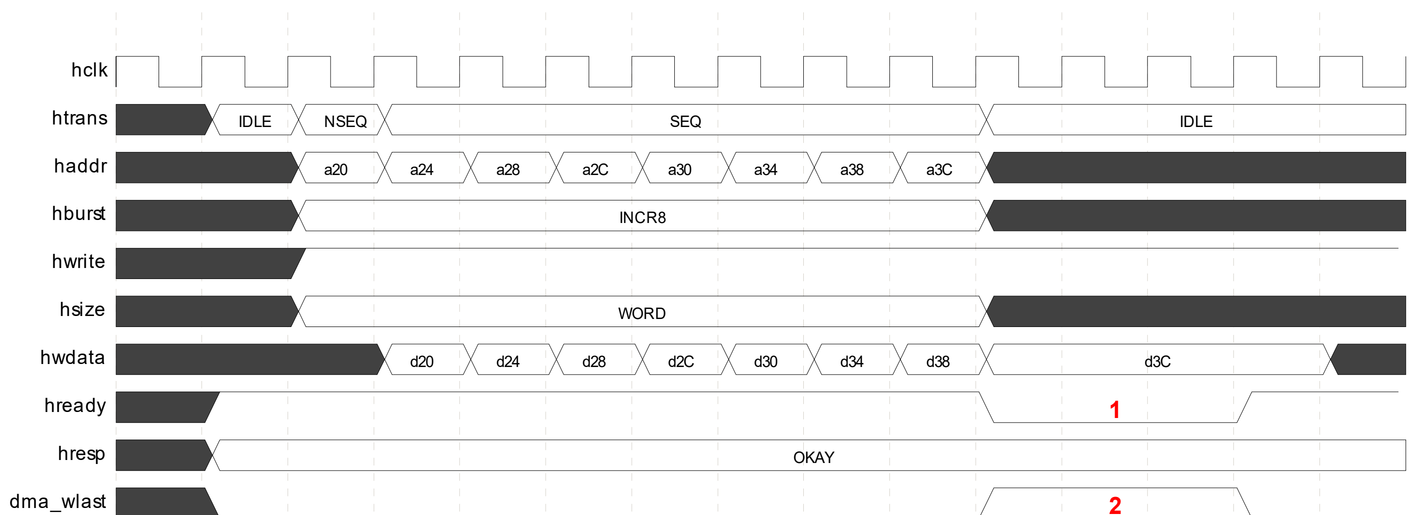


Figure 2-48 shows the following:

1. Wait states are inserted on the last data beat of a destination MSIZE transfer.
2. The dma_wlast signal remains asserted till the last data beat completes.

Figure 2-49 shows the behavior of the dma_wlast signal when SPLIT/RETRY response is issued by a slave on the last beat of destination MSIZE data transfer.

Figure 2-49 Behavior of the dma_wlast Signal During SPLIT Response

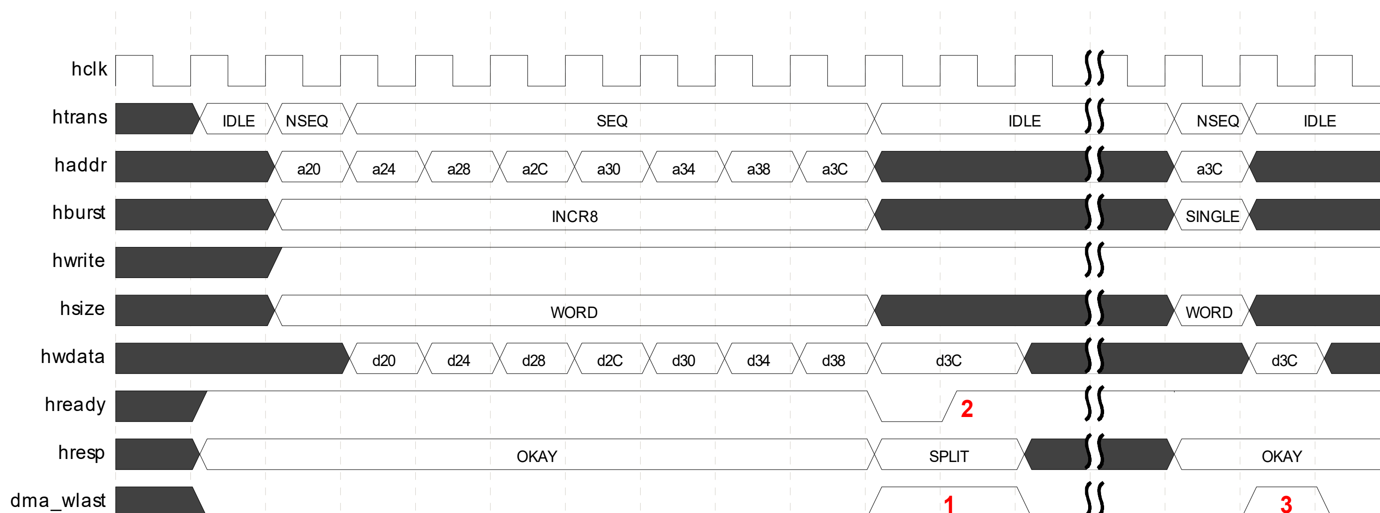


Figure 2-49 shows the following:

1. The dma_wlast asserted after the address phase of the last data beat of a destination MSIZE transfer.
2. The slave has issued a SPLIT response.
3. The dma_wlast again asserted while completing the last data beat of a destination MSIZE transfer.

2.19 Low Power Modes – Global and Channel Clock Gating

DW_ahb_dmac supports low power implementation, which is achieved by performing clock gating based on the idle time. This section discusses the following clock gating methodologies that DW_ahb_dmac incorporates to implement low power:

- "Global Clock Gating" on page 111
- "Channel Clock Gating" on page 113

2.19.1 Global Clock Gating

Global clock gating is a method to support low power mode in DW_ahb_dmac. When this mode is configured, DW_ahb_dmac enters the low power mode when DW_ahb_dmac is idle for a certain period of time.

The following conditions are used to determine whether DW_ahb_dmac is idle:

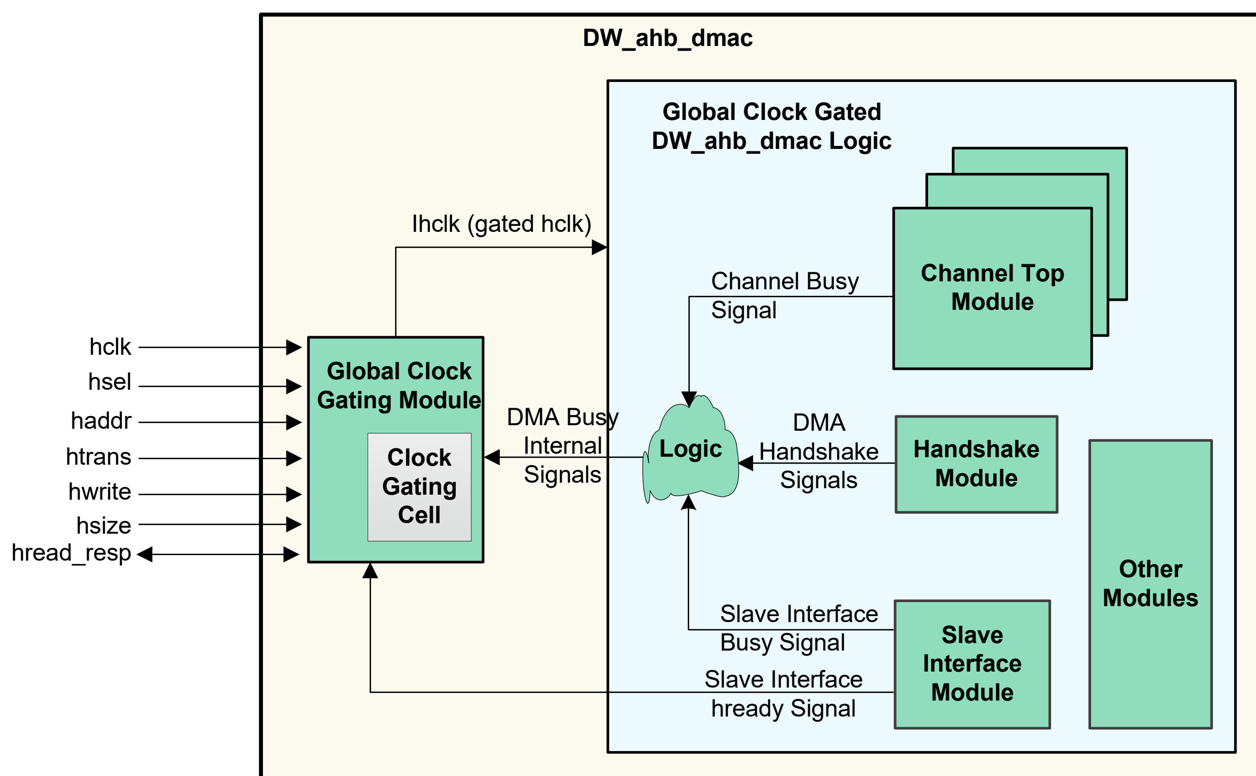
- All Channels are disabled and transactions are not happening in the slave interface.

- Transaction is not happening on the DW_ahb_dmac master interfaces and the active channels are in the IDLE state. This means that a channel is waiting for a request from a handshaking interface that takes considerable time to respond and active transfers are also not happening on the AHB Slave Interface.

Global clock gating is configured using the DMAH_LP_EN parameter and the inactive time period is programmed using the DMAH_LP_TIMEOUT register. The timeout period can also be hardcoded using the DMAH_HC_LP_TIMEOUT_VALUE parameter.

DW_ahb_dmac includes an internal clock gating cell to enable clock gating. This cell must be replaced by a user-specific clock gating module during synthesis. Figure 2-50 describes the block diagram of DW_ahb_dmac when DMAH_LP_EN is set to 1, that is when global clock gating is enabled.

Figure 2-50 Global Clock Gating in DW_ahb_dmac



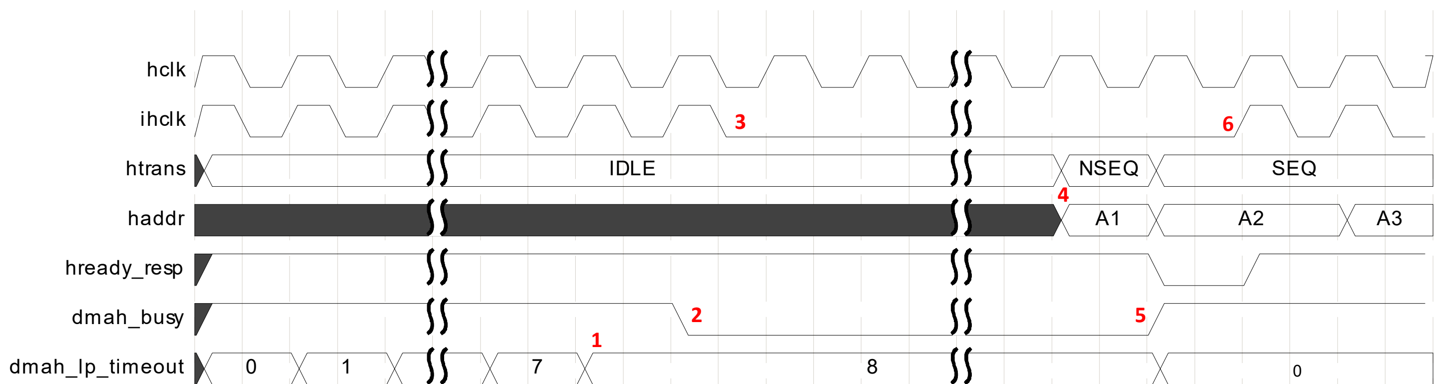
After entering low power mode, DW_ahb_dmac can wake up under any of the following conditions:

- Register read/write on the AHB Slave interface.
- Handshake request initiated by the peripheral.

As soon as DW_ahb_dmac is inactive, the DW_ahb_dmac low power counter starts incrementing, and when this counter times out, DW_ahb_dmac enters the low power mode (as shown in Figure 2-51). To exit the low power mode, DW_ahb_dmac takes one hclk cycle. As all other blocks are clock gated during this period of time, all read/write are delayed by one cycle.

Figure 2-51 describes the wake up sequence of DW_ahb_dmac when it exits low power mode after noticing a read/write transaction on the slave interface.

Figure 2-51 Global Clock Gating Sequence in DW_ahb_dmac – Wakeup Due to a Read/Write Transaction



In Figure 2-51:

- The dmah_lp_timeout signal denotes the low power counter that expires as the counter reaches the timeout value (that is, 8 in Figure 2-51).
- The dmah_busy signal denotes the internal signal indicating that DW_ahb_dmac is busy (not idle) and is de-asserted. This indicates that DW_ahb_dmac has to enter into the low power mode.
- The internal low power mode gated hclk (ihclk) is switched off.
- A register read/write transaction is received on the AHB Slave interface.
- The internal dmah_busy signal is asserted and low power counter is reset, indicating that DW_ahb_dmac must wake up from the low power mode.
- The internal ihclk is switched on, to exit the low power mode.

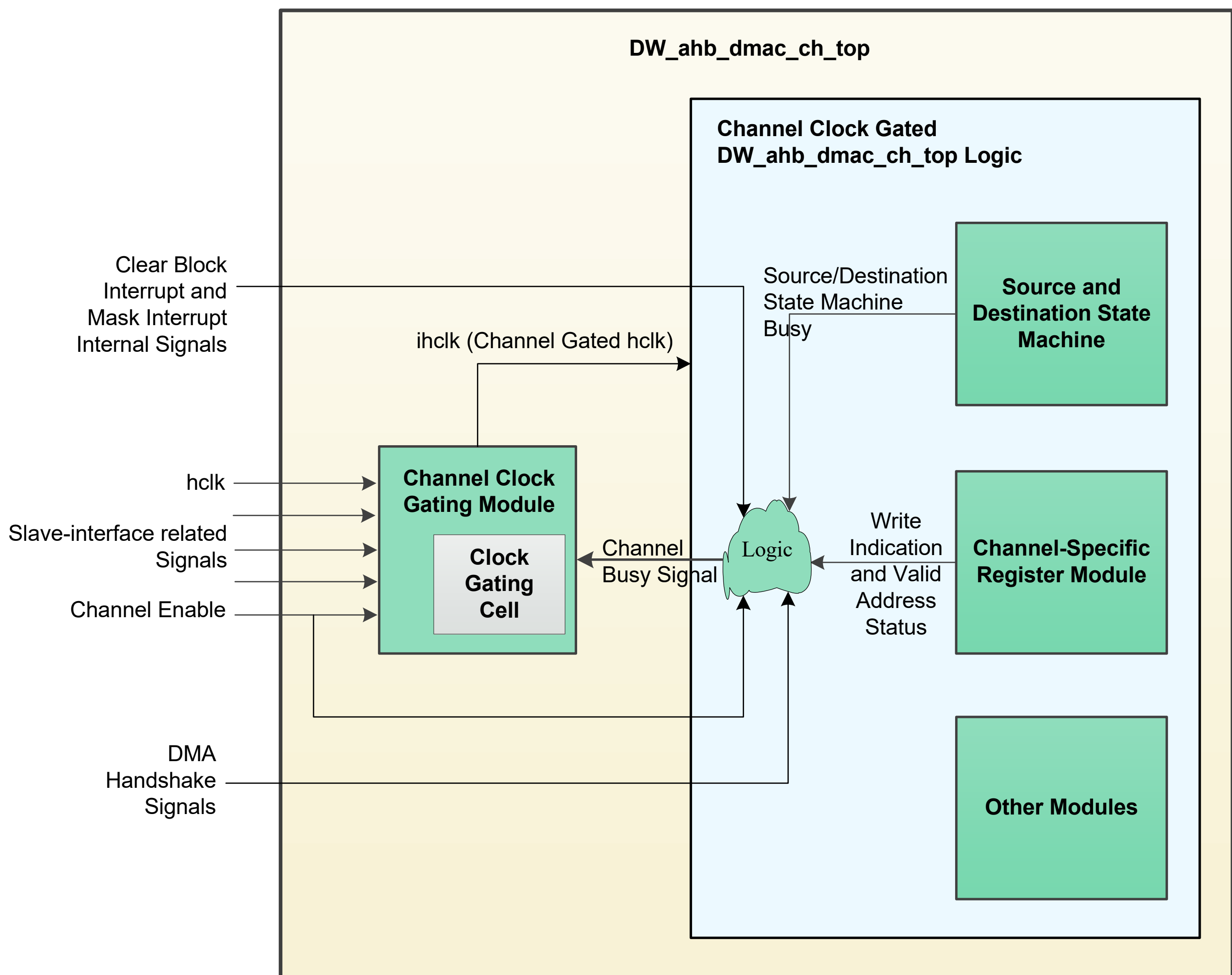
2.19.2 Channel Clock Gating

Channel Clock Gating is another method to enable low power mode, in addition to the Global Clock Gating method (see "Global Clock Gating" on page 111). When this mode is configured, DW_ahb_dmac turns off the clock for a particular channel when the channel is idle. Only a small portion of the channel that is responsible to turn on the clock is constantly supplied with a clock.

This mode is configured using the DMAH_CH_LP_EN parameter and the inactive time period is programmed using the DMAH_LP_TIMEOUT register.

Figure 2-52 describes the block diagram of DW_ahb_dmac when DMAH_CH_LP_EN is set to 1, that is, when Channel Clock Gating is enabled.

Figure 2-52 Channel Clock Gating in DW_ahb_dmac



The Channel Clock Gating sequence is similar to that of Global Clock Gating.

The conditions for a channel to enter low power mode in Channel Clock Gating are as follows:

- DW_ahb_dmac comes out of reset and none of the channel registers are being written.
- While a channel is enabled, activity does not happen on the handshaking interface and active AHB transfer does not happen (related to the specific channel) on the Master interface.
- When a channel gets disabled and remains inactive for the duration programmed or hardcoded in DMAH_LP_TIMEOUT register.

If the channel is clock gated due to inactivity on the handshaking interface/AHB Master interface when a channel is enabled or disabled, then the channel exits from clock low power mode due any to the following conditions:

- Any new request is detected on the hardware/software handshaking interface.
- Write transaction occurs on any channel-specific registers.

- Block interrupt corresponding to the channel cleared through the ClearBlock register.
- Block mask corresponding to the channel asserted through the MaskBlock register.
- Channel is disabled due to:
  - De-assertion of CH_EN bit (that corresponds to the channel of interest) of the ChEnReg register.
  - De-assertion of DMA_EN bit of the DmaCfgReg register.
  - Channel disabling due to the completion of transfer or error response received on AHB Master interface for source and destination.

The Channel Clock Gating can be enabled only if the Global Clock Gating is enabled.

2.20 Interrupt Registers

The following sections describe the registers pertaining to interrupts, their status, and how to clear them. For each channel, there are five types of interrupt sources:

- IntBlock – Block Transfer Complete Interrupt

  This interrupt is generated on DMA block transfer completion to the destination peripheral.
- IntDstTran – Destination Transaction Complete Interrupt

  This interrupt is generated after completion of the last AHB transfer of the requested single/burst transaction from the handshaking interface (either the hardware or software handshaking interface) on the destination side.

Note If the destination for a channel is memory, then that channel will never generate the IntDstTran interrupt. Because of this, the corresponding bit in this field will not be set.

- IntErr – Error Interrupt

  This interrupt is generated when an ERROR response is received from an AHB slave on the HRESP bus during a DMA transfer. In addition, the DMA transfer is cancelled and the channel is disabled.
- IntSrcTran – Source Transaction Complete Interrupt

  This interrupt is generated after completion of the last AHB transfer of the requested single/burst transaction from the handshaking interface (either the hardware or software handshaking interface) on the source side.

Note If the source or destination is memory, then IntSrcTran/IntDstTran interrupts should be ignored, as there is no concept of a "DMA transaction level" for memory.

- IntTfr – DMA Transfer Complete Interrupt

  This interrupt is generated on DMA transfer completion to the destination peripheral.

There are several groups of interrupt-related registers:

- RawBlock, RawDstTran, RawErr, RawSrcTran, RawTfr

- StatusBlock, StatusDstTran, StatusErr, StatusSrcTran, StatusTfr
- MaskBlock, MaskDstTran, MaskErr, MaskSrcTran, MaskTfr
- ClearBlock, ClearDstTran, ClearErr, ClearSrcTran, ClearTfr
- StatusInt

When a channel has been enabled to generate interrupts, the following is true:

- Interrupt events are stored in the Raw Status registers.
- The contents of the Raw Status registers are masked with the contents of the Mask registers.
- The masked interrupts are stored in the Status registers.
- The contents of the Status registers are used to drive the int_* port signals.
- Writing to the appropriate bit in the Clear registers clears an interrupt in the Raw Status registers and the Status registers on the same clock cycle.

The contents of each of the five Status registers is ORed to produce a single bit for each interrupt type in the Combined Status register; that is, StatusInt.

Note For interrupts to propagate past the raw* interrupt register stage, CTLx.INT_EN must be set to 1'b1, and the relevant interrupt must be unmasked in the mask* interrupt register.

3 Parameter Descriptions

This chapter details all the configuration parameters. **You can use the coreConsultant GUI configuration reports to determine the actual configured state of the controller.** Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

The parameter descriptions in this chapter include the **Enabled:** attribute which indicates the values required to be set on other parameters before you can change the value of this parameter.

These tables define all of the user configuration options for this component.

- DMA Source Code Configuration on page 118
- Global DMA Configuration on page 119
- Configuration of AMBA layers on page 125
- Channel x configuration on page 129

3.1 DMA Source Code Configuration Parameters

Table 3-1 DMA Source Code Configuration Parameters

Label	Description
DMA Source Code Configuration	
Use DesignWare Foundation Synthesis Library	The component code utilizes DesignWare Foundation parts for optimal Synthesis QoR. Customers with only a DesignWare license must use Foundation parts. Customers with only a Source license, CANNOT use Foundation parts. Customers with both Source and DesignWare licenses have the option of using DesignWare Foundation parts. **Values:** ■ false (0) ■ true (1) **Default Value:** True if DesignWare License is available; False if no DesignWare License is available. **Enabled:** Parameter is enabled if customer has both Source and DesignWare licenses. **Parameter Name:** USE_FOUNDATION

3.2 Global DMA Configuration Parameters

Table 3-2 Global DMA Configuration Parameters

Label	Description
Top Level Configurations	
Number of AHB master interfaces	Creates the specified number of AHB master interfaces. A channel source or destination device can be programmed to be on any of the configured AHB layers attached to the AHB Master interface. This setting determines if a master interface signal set is present on the I/O or not. AHB master interface 1 signals are always present. **Values:** 1, 2, 3, 4 **Default Value:** 1 **Enabled:** Always **Parameter Name:** DMAH_NUM_MASTER_INT
Number of DMA channels	Creates the specified number of DW_ahb_dmac channels. Each channel is uni-directional and transfers data from the channel source to the channel destination. The channel source and destination AHB layer, system address and handshaking interface are under software control. **Values:** 1, 2, 3, 4, 5, 6, 7, 8 **Default Value:** 1 **Enabled:** Always **Parameter Name:** DMAH_NUM_CHANNELS
Number of handshaking interfaces	Creates the specified number of handshaking interfaces. You can program the DW_ahb_dmac to assign a handshaking interface for each channel source and destination. If 0 is selected, then no hardware handshaking signals are present on the I/O. **Values:** 0, ..., 16 **Default Value:** 2 **Enabled:** Always **Parameter Name:** DMAH_NUM_HS_INT
Use undefined length bursts only?	When set to 1, the DW_ahb_dmac selects undefined length INCR bursts only. Setting this option to 0 allows the DW_ahb_dmac to select the largest valid defined length burst (SINGLE, INCR4, INCR8, INCR16) for the transfer. **Note**: It is not valid to send defined-length bursts to non-changing or decrementing addresses. If this option is set to 0, the DW_ahb_dmac selects undefined-length INCR bursts when CTLx.SINC or CTLx.DINC are set to Decrement or No Change. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** Always **Parameter Name:** DMAH_INCR_BURSTS

Table 3-2 Global DMA Configuration Parameters (Continued)

Label	Description
Allow the slave interface to return an error response when an illegal access is attempted?	Setting this option to 1 allows the slave interface to return an error response on the hresp bus when an illegal access is attempted over the AHB slave interface. For a list of illegal accesses, refer to \"Illegal Register Access\" chapter. Setting this option to 0 results in an OKAY response being returned on the hresp bus for all of the illegal accesses. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** Always **Parameter Name:** DMAH_RETURN_ERR_RESP
Interrupts are active high?	Interrupt active polarity for all of the configured interrupt ports. When this box is selected the interrupt polarity for all interrupt pins on the I/O interface is active high. When this box is unselected the interrupt polarity for all interrupt pins on the I/O interface is active low. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** DMAH_INTR_POL

Table 3-2 Global DMA Configuration Parameters (Continued)

Label	Description
Interrupt pins to appear as outputs	Selects which interrupt related pins appear as outputs of the design. ■ **All**:Contents of all five Interrupts Status Registers appear as output pins on int(_n) output bus. 5 * DMAH_NUM_CHANNELS bits wide. ■ **Type**:All internal interrupts are combined by type and appear on int_flag(_n) output 5 bit bus. - bit[4] = Error interrupt - bit[3] = Destination transaction complete interrupt - bit[2] = Source transaction complete interrupt - bit[1] = Block complete interrupt - bit[0] = Transfer complete interrupt Also, the bitwise OR of all bits of int_flag(_n) bus is driven onto the int_combined(_n) single bit output port. ■ **Combined** - Bitwise OR of all bits of int_flag(_n) bus is driven onto the int_combined(_n) single bit output port. **Values:** ■ ALL (0) ■ TYPE (1) ■ COMBINED (2) **Default Value:** COMBINED **Enabled:** Always **Parameter Name:** DMAH_INTR_IO
ID number	This 32-bit value is hardwired and read back by a read to the DMA Controller ID Register (DmaldReg). **Values:** 0x0, ..., 0xffffffff **Default Value:** 0x0 **Enabled:** Always **Parameter Name:** DMAH_ID_NUM
Allow the AMBA burst length to be limited to a programmable maximum value?	When set to 1, you can limit the maximum AMBA burst length to a value under software control by writing to the channel configuration register; this is a global parameter for all of the configured master interfaces. Setting this option to 0 allows for some logic optimization. **Note**:If this option is set to 0, then the maximum AMBA burst length is a function of the channel FIFO depths. The DW_ahb_dmac may try to fill or empty a channel FIFO in a single burst. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** DMAH_MABRST

Table 3-2 Global DMA Configuration Parameters (Continued)

Label	Description
Remove Internal Pipeline Stages ?	Setting this parameter to 1 removes 3 pipeline stages internal to DW_ahb_dmac. ■ Input register on hardware handshaking interface input signals. ■ Output register on master interface arbiter. ■ Register on path from the destination state machine to dma_ack*. Removing these pipeline stages improves latency, but can result in reduced operating frequency. Setting this parameter to 0 keeps the 3 pipeline stages. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** DMAH_REMOVE_PIPELINING
Reverse order of LLI Status WriteBack	When the DMAC writes back the status information to the LLI memory location, the following order is used: 1. DSTATx 2. SSTATx 3. CTRLx. When this parameter is set, the write back order will be changed to 1. CTRLx 2. SSTATx 3. DSTATx **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** At least one of DMAH_CHx_CTL_WB_EN parameters should be equal to 1 **Parameter Name:** DMAH_REVERSE_WB_OVERRIDE
Add encoded parameters	Adding the encoded parameters gives firmware an easy and quick way of identifying the DesignWare component within an I/O memory map. Some critical design-time options determine how a driver should interact with the peripheral. There is a minimal area overhead when you include these parameters. Additionally, this option allows a self-configurable single driver to be developed for each component. **Values:** ■ false (0x0) ■ true (0x1) **Default Value:** true **Enabled:** Always **Parameter Name:** DMAH_ADD_ENCODED_PARAMS

Table 3-2 Global DMA Configuration Parameters (Continued)

<table>
<tr><th>Label</th><th>Description</th></tr>
<tr><td>Include logic to enable dma_wlast[N] Signal?</td><td>Enables the additional handshaking signal dma_wlast on all selected AHB interfaces. The dma_wlast signal is asserted on completion of the last address phase of every destination transaction and remains asserted until the associated last dataphase completes.
Values:
■ false (0)
■ true (1)
Default Value: false
Enabled: DMAH_NUM_HS_INT != 0
Parameter Name: DMAH_WLAST_EN</td></tr>
<tr><td colspan="2">Low Power Options</td></tr>
<tr><td>Enable DW_ahb_dmac Global Low Power logic?</td><td>This parameter is used to enable Global Clock Gating logic
Values:
■ false (0)
■ true (1)
Default Value: True if DesignWare License is available; False if no DesignWare License is available.
Enabled: Parameter is enabled if customer has both Source and DesignWare licenses.
Parameter Name: DMAH_LP_EN</td></tr>
<tr><td>Enable DW_ahb_dmac Channel Low Power logic?</td><td>This parameter is used to enable Channel Global Clock Gating logic
Values:
■ false (0)
■ true (1)
Default Value: false
Enabled: DMAH_LP_EN==1
Parameter Name: DMAH_CH_LP_EN</td></tr>
<tr><td>Width of Low Power Counter register</td><td>Defines width of Low Power Counter. Setting the values from 4 through 32 for the parameter, enables the user to configure the timeout period value from 4 to 2^32-1 pclk cycles.
Values: 4, ..., 32
Default Value: 4
Enabled: DMAH_LP_EN || DMAH_CH_LP_EN
Parameter Name: DMAH_LP_TIMEOUT_WIDTH</td></tr>
</table>

Table 3-2 Global DMA Configuration Parameters (Continued)

Label	Description
Hardcode time-out counter value	Enabling this parameter makes Low Power Counter register a read-only register. The register can be programmed by user if the hardcode option is turned off. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** (DMAH_LP_EN || DMAH_CH_LP_EN) && DMAH_LP_TIMEOUT_WIDTH!=1 **Parameter Name:** DMAH_HC_LP_TIMEOUT_VALUE
Default Value of inactive cycles after which DW_ahb_dmac goes into Low Power mode	Defines the reset value of Low Power Counter register. This value can be over - ridden by programming the Low Power Counter register before enabling the DW_ahb_dmac. **Values:** 4, ..., 4294967295 **Default Value:** 8 **Enabled:** (DMAH_LP_EN || DMAH_CH_LP_EN) && DMAH_LP_TIMEOUT_WIDTH>2 **Parameter Name:** DMAH_LP_TIMEOUT_VALUE

3.3 Configuration of AMBA layers Parameters

Table 3-3 Configuration of AMBA layers Parameters

Label	Description
	Configuration of AMBA layers
Statically configure endianness through coreConsultant?	The endian scheme of the DW_ahb_dmac can be configured statically through coreConsultant or dynamically via pins on the I/O. For the static case, there is a single coreConsultant parameter that selects between Little Endian/Big Endian for all AHB master interfaces and the AHB slave interface. For the dynamic case, there is an individual pin for each of the AHB master interfaces and one for the AHB slave interface. Selection between AI and BE-32 method for Big Endian access is controlled by additional parameters defined for slave interface and each master interface. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** Always **Parameter Name:** DMAH_STATIC_ENDIAN_SELECT
System is big endian?	The AHB master and slave interfaces can be configured to exist in either big or little endian system. When set to 1, all master interfaces and the slave interface become big endian. Otherwise, they are set to little endian. Selection between AI and BE-32 methods for big endian access is controlled by additional parameters and inputs defined for slave interface and each master interface. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_STATIC_ENDIAN_SELECT==1 **Parameter Name:** DMAH_BIG_ENDIAN

Table 3-3 Configuration of AMBA layers Parameters (Continued)

Label	Description
Add AI/BE-32 Big Endian scheme selection pin on AHB Slave and Master Interfaces ?	If this parameter is enabled, additional inputs are enabled to control the scheme used for big endian access (individual pin is added for AHB Slave interface and each of the AHB Master interfaces). AHB Slave interface and each AHB Master interfaces can be independently configured to support Address Invariant (AI) or Word Invariant (BE-32) Big Endian scheme if this interface is configured for Big Endian access. ■ 0: Address Invariant(AI) method is used for Big Endian access for Slave and Master 1/2/3/4 Interfaces (current scheme) ■ 1: Address Invariant(AI) or Word Invariant(BE-32) method is used for Big Endian access for Slave and Master 1/2/3/4 Interfaces depending on the value of dma_be_select_be32_slv/mN, input where N = 1/2/3/4 **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** (DMAH_STATIC_ENDIAN_SELECT==0) || ((DMAH_STATIC_ENDIAN_SELECT==1) && (DMAH_BIG_ENDIAN==1)) **Parameter Name:** DMAH_BE32_SELECTION_PIN_EN
Add Little Endian scheme selection pin for LLI access on AHB Master Interfaces ?	If this parameter is enabled, additional inputs are enabled to control the endian scheme used for LLI fetch and Status and Control writeback. (Individual pin is added for each of the AHB Master interfaces.) LLI fetch and Status and Control write-back access on each AHB Master interfaces can be independently configured to support Big Endian scheme (AI or BE-32 based on the endian scheme selected for that particular Master interface) or Little Endian scheme (irrespective of the endian scheme selected for that particular Master interface). ■ 0: Endian scheme used for LLI fetch and Status & Control write-back is same as that used for data access for M1/M2/M3/M4 interfaces (current scheme) ■ 1: Endian scheme used for LLI fetch and Status & Control write-back access is same as that used for data access for M1/M2/M3/M4 interfaces (OR) Little Endian method is used for LLI fetch and Status and Control writeback access irrespective of the endian scheme used for data access depending on the value of dma_le_select_lli_mN input, where N = 1/2/3/4. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** (DMAH_STATIC_ENDIAN_SELECT==0) || ((DMAH_STATIC_ENDIAN_SELECT==1) && (DMAH_BIG_ENDIAN==1)) **Parameter Name:** DMAH_LLI_ENDIAN_SELECTION_PIN_EN

Table 3-3 Configuration of AMBA layers Parameters (Continued)

Label	Description
Slave interface data bus width	Specifies the data bus width for the AHB slave interface. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** Always **Parameter Name:** DMAH_S_HDATA_WIDTH
Is Layer 1 AMBA-Lite?	Select this box if master interface 1 is the only master on this AMBA layer. If this is the case then this is an AMBA-Lite layer. De-select this box if there is more than one master on this AMBA layer. Even though the DMAH_M1_AHB_LITE parameter is associated with master behavior, this parameter configures the DW_ahb_dmac in AHB Lite mode for slave functions as well. When the DMAH_M1_AHB_LITE parameter is configured for AHB Lite mode, the SPLIT and RETRY functions are disabled for both master and slave functions. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_MASTER_INT>=1 **Parameter Name:** DMAH_M1_AHB_LITE
Master 1 interface data bus width	AHB Master 1 interface data bus width. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** DMAH_NUM_MASTER_INT>=1 **Parameter Name:** DMAH_M1_HDATA_WIDTH
Is Layer 2 AMBA-Lite?	Select this box if master interface 2 is the only master on this AMBA layer. If this is the case then this is an AMBA-Lite layer. De-select this box if there is more than one master on this AMBA layer. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_MASTER_INT>=2 **Parameter Name:** DMAH_M2_AHB_LITE
Master 2 interface data bus width	AHB Master 2 interface data bus width. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** DMAH_NUM_MASTER_INT>=2 **Parameter Name:** DMAH_M2_HDATA_WIDTH

Table 3-3 Configuration of AMBA layers Parameters (Continued)

Label	Description
Is Layer 3 AMBA-Lite?	Select this box if master interface 3 is the only master on this AMBA layer. If this is the case then this is an AMBA-Lite layer. De-select this box if there is more than one master on this AMBA layer. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_MASTER_INT>=3 **Parameter Name:** DMAH_M3_AHB_LITE
Master 3 interface data bus width	AHB Master 3 interface data bus width. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** DMAH_NUM_MASTER_INT>=3 **Parameter Name:** DMAH_M3_HDATA_WIDTH
Is Layer 4 AMBA-Lite?	Select this box if master interface 4 is the only master on this AMBA layer. If this is the case then this is an AMBA-Lite layer. De-select this box if there is more than one master on this AMBA layer. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_MASTER_INT>=4 **Parameter Name:** DMAH_M4_AHB_LITE
Master 4 interface data bus width	AHB Master 4 interface data bus width. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** DMAH_NUM_MASTER_INT>=4 **Parameter Name:** DMAH_M4_HDATA_WIDTH

3.4 Channel x configuration Parameters

Table 3-4 Channel x configuration Parameters

Label	Description
Channel x configuration	
Channel x FIFO depth in bytes (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Channel x FIFO depth in bytes. **Values:** 8, 16, 32, 64, 128, 256 **Default Value:** 16 **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_FIFO_DEPTH
Maximum value of burst transaction size (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Maximum value of burst transaction size that can be programmed for channel x (CTLx.SRC_MSIZE and CTLx.DEST_MSIZE). Limiting DMAH_CHx_MAX_MULT_SIZE to a maximum value will allow for some logic optimization of the implementation. **Values:** 4, 8, 16, 32, 64, 128, 256 **Default Value:** 8 **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_MAX_MULT_SIZE
Maximum block size in source transfer widths (for x = 0; x <= DMAH_NUM_CHANNELS-1)	The description of this parameter is dependent on what is assigned as the flow controller. ■ **DW_ahb_dmac flow controller**: Maximum block size, in multiples of source transfer width, that can be programmed for channel x. A programmed value greater than this will result in erroneous behavior. ■ **Source/destination assigned as flow controller**: In this case, the blocks can be greater than DMAH_CHx_MAX_BLK_SIZE in size, but the logic that keeps track of the size of a block saturates at DMAH_CHx_MAX_BLK_SIZE. This does not result in erroneous behavior, but a readback by software of the block size is incorrect when the block size exceeds the saturated value. This parameter is also used to limit the gather and scatter count registers to a maximum value of DMAH_CHx_MAX_BLK_SIZE. Limiting DMAH_CHx_MAX_BLK_SIZE to a maximum value allows some logic optimization of the implementation. **Values:** 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095 **Default Value:** 31 **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_MAX_BLK_SIZE

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Hardcode channel x's flow control device to allow for logic optimization (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcodes the flow control peripheral for the channel. If ANY_FC is selected, then the flow control device is not hardcoded, and software selects the flow control device for a DMA transfer. Hardcoding the flow control device allows some logic optimization of the implementation. **Values:** ■ DMA_FC_ONLY (0) ■ SRC_FC_ONLY (1) ■ DST_FC_ONLY (2) ■ ANY_FC (3) **Default Value:** DMA_FC_ONLY **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_FC
Include logic to enable channel or bus locking on channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to enable channel or bus level locking on channel x. When set to 1, then software can program the DMA controller to assert hlock over the DMA transfer, DMA block transfer or DMA transaction. When enabled, then software can also program the DMA controller to lock the arbitration for the master bus interface unit over the DMA transfer, DMA block transfer or DMA transaction. Disabling this option allows for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_LOCK_EN
Hardcode the master interface attached to the source of channel x (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcode the Master interface attached to the source of channel x. If this is not hardcoded, then software can program the source of channel x to be attached to any of the configured layers. Hardcoding this value will allow for some logic optimization of the implementation. **Values:** ■ MASTER_1 (0) ■ MASTER_2 (1) ■ MASTER_3 (2) ■ MASTER_4 (3) ■ NO_HARDCODE (4) **Default Value:** MASTER_1(0) if DMAH_NUM_MASTER_INT = 1; NO_HARDCODE otherwise **Enabled:** DMAH_NUM_CHANELS >= x **Parameter Name:** DMAH_CHx_SMS

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Hardcode the master interface attached to the destination of channel x (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcode the Master interface attached to the destination of channel x. If this is not hardcoded then software can program the destination of channel x to be attached to any of the configured layers. Hardcoding this value will allow for some logic optimization of the implementation. **Values:** ■ MASTER_1 (0) ■ MASTER_2 (1) ■ MASTER_3 (2) ■ MASTER_4 (3) ■ NO_HARDCODE (4) **Default Value:** MASTER_1(0) if DMAH_NUM_MASTER_INT = 1; NO_HARDCODE otherwise **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_DMS
Hardcode channel x's source transfer width (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcode the source transfer width for transfers from the source of channel x. If this is not hardcoded then software can program the source transfer width for channel x. Hardcoding this value will allow for some logic optimization of the implementation. **Values:** ■ NO_HARDCODE (0) ■ BYTE (8) ■ HALFWORD (16) ■ WORD (32) ■ TWO_WORD (64) ■ FOUR_WORD (128) ■ EIGHT_WORD (256) **Default Value:** WORD **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_STW

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Hardcode channel x's destination transfer width (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcode the destination transfer width for transfers from the destination of channel x. If this is not hardcoded then software can program the destination transfer width for channel x. Hardcoding this value will allow for some logic optimization of the implementation. **Values:** ■ NO_HARDCODE (0) ■ BYTE (8) ■ HALFWORD (16) ■ WORD (32) ■ TWO_WORD (64) ■ FOUR_WORD (128) ■ EIGHT_WORD (256) **Default Value:** WORD **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_DTW
Can the source of channel x return a non-ok response on hresp? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Set this parameter to 1 if the source peripheral attached to channel x can return a non-OK response on hresp, such as a SPLIT, RETRY, or ERROR response. If set to 0, then hardware assumes that only OK responses are returned from the source peripheral attached to the channel. Setting this parameter to 0 allows some logic optimization of the implementation **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_SRC_NON_OK
Can the destination of channel x return a non-ok response on hresp? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Set this parameter to 1 if the destination peripheral attached to channel x can return a non-OKAY response on hresp, such as a SPLIT, RETRY, or ERROR response. If set to 0, then hardware assumes that only OKAY responses are returned from the destination peripheral attached to the channel. Setting this parameter to 0 allows some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_DST_NON_OK

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Include logic to enable multi-block DMA transfers on channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Includes or excludes logic to enable multi-block DMA transfers on channel x. If this option is de-selected then hardware hardwires channel x to do single block transfers only. De-selecting this box will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_MULTI_BLK_EN

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Choose type of multi-blocks to be supported? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	This parameter allows the user to hardcode the type of multi-block transfers that DW_ahb_dmac can perform. This results in some logic optimization of the implementation. **Legal Values**: ■ NO_HARDCODE: Allow all types of multi-block support ■ CONT_RELOAD: Allow only multi-block transfers where SARx is contiguous; DARx and CTLx are reloaded from their initial values. ■ RELOAD_RELOAD: Allow only multi-block transfers where SARx,DARx and CTLx are reloaded from their initial values. ■ CONT_LLP: Allow only multi-block transfers where SARx is contiguous; DARx,CTLx and LLPx are loaded from the next Linked List Item. ■ RELOAD_LLP: Allow only multi-block transfers where SARx is reloaded from it's initial value; DARx,CTLx and LLPx are loaded from the next Linked List Item. ■ LLP_CONT: Allow only multi-block transfers where SARx,CTLx and LLPx are loaded from the next Linked List Item; DARx is contiguous. ■ LLP_RELOAD: Allow only multi-block transfers where SARx,CTLx and LLPx are loaded from the next Linked List Item; DARx is reloaded from it's initial values. ■ LLP_LLP: Allow only multi-block transfers where SARx,DARx,CTLx and LLPx are loaded from the next Linked List Item. **Values:** ■ NO_HARDCODE (0) ■ CONT_RELOAD (1) ■ RELOAD_CONT (2) ■ RELOAD_RELOAD (3) ■ CONT_LLP (4) ■ RELOAD_LLP (5) ■ LLP_CONT (6) ■ LLP_RELOAD (7) ■ LLP_LLP (8) **Default Value:** NO_HARDCODE **Enabled:** DMAH_CHx_MULTI_BLK_EN = 1 **Parameter Name:** DMAH_CHx_MULTI_BLK_TYPE

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Hardcode Channel x LLP register to 0? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	If set to 1, hardcodes channel x Linked List Pointer register to 0 (LLPx.LOC == 0), which disables the following features: ■ Multi-block DMA transfers using block chaining ■ Source and destination status fetch ■ Control and source/destination status writeback Multi-block DMA transfers may still be enabled on channel x through auto-reloading of channel registers. Selecting this box will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** true **Enabled:** DMAH_CHx_MULTI_BLK_TYPE == 0 or DMAH_CHx_MULTI_BLK_TYPE >= 4 **Parameter Name:** DMAH_CHx_HC_LLP
Fetch status from source of channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to fetch a status register from the source peripheral of channel x and write this status information to system memory at the end of each block transfer. The location from where the source status register is fetched from is under software control as is the writeback location. Disabling this parameter will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_CHx_HC_LLP==0 **Parameter Name:** DMAH_CHx_STAT_SRC
Fetch status from destination of channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to fetch a status register from the destination peripheral of channel x and write this status information to system memory at the end of each block transfer. The location from where the destination status register is fetched from is under software control as is the writeback location. Disabling this parameter will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_CHx_HC_LLP==0 **Parameter Name:** DMAH_CHx_STAT_DST

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Hardcode the master interface attached to the LLP peripheral of channel 0 (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Hardcode the AHB master interface attached to the peripheral that stores the LLI information (Linked List Item) for channel x. If this is not hardcoded, then software can program the peripheral that stores the LLI information of channel x to be attached to any of the configured layers. Hardcoding this value allows some logic optimization of the implementation. LLI accesses are always 32-bit accesses (Hsize = 2) aligned to 32-bit boundaries and cannot be changed or programmed to anything other than 32-bit, even if the AHB master interface of the LLI supports more than a 32-bit data width. **Values:** ■ MASTER_1 (0) ■ MASTER_2 (1) ■ MASTER_3 (2) ■ MASTER_4 (3) ■ NO_HARDCODE (4) **Default Value:** MASTER_1(0) if DMAH_NUM_MASTER_INT = 1; NO_HARDCODE otherwise **Enabled:** DMAH_CHx_HC_LLP == 0 and DMAH_NUM_MASTER_INT > 1 and (DMAH_CHx_MULTI_BLK_TYPE == 0 or DMAH_CHx_MULTI_BLK_TYPE >= 4) **Parameter Name:** DMAH_CHx_LMS
Can the LLP peripheral of channel 0 return a non-ok response on hresp? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Set this parameter to 1 if the LLP peripheral attached to channel x can return a non-OK response on hresp, such as a SPLIT, RETRY, or ERROR response. If set to 0, then hardware assumes that only OK responses are returned from the LLP peripheral attached to the channel. Setting this parameter to 0 allows some logic optimization of the implementation **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_CHx_HC_LLP == 0 and (DMAH_CHx_MULTI_BLK_TYPE == 0 or DMAH_CHx_MULTI_BLK_TYPE >= 4) **Parameter Name:** DMAH_CHx_LLP_NON_OK
Include logic to enable the 'gather' feature on channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to enable the 'gather' feature on channel x. De-selecting this box will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_SRC_GAT_EN

Table 3-4 Channel x configuration Parameters (Continued)

Label	Description
Include logic to enable 'scatter' feature on channel x? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to enable the 'scatter' feature on channel x. De-selecting this box will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** DMAH_NUM_CHANNELS >= x **Parameter Name:** DMAH_CHx_DST_SCA_EN
Include logic to enable control register writeback after each block transfer? (for x = 0; x <= DMAH_NUM_CHANNELS-1)	Include or exclude logic to enable writeback of the CTLx, SSTATx and DSTATx registers at the end of every block transfer. De-selecting this box will allow for some logic optimization of the implementation. **Values:** ■ false (0) ■ true (1) **Default Value:** False (0) **Enabled:** DMAH_CHx_HC_LLP == 0 and DMAH_CHx_STAT_SRC == 0 and DMAH_CHx_STAT_DST == 0 **Parameter Name:** DMAH_CHx_CTL_WB_EN

4
Signal Descriptions

This chapter details all possible I/O signals in the core. For configurable IP titles, your actual configuration might not contain all of these signals.

Inputs are on the left of the signal diagrams; outputs are on the right.

Attention: For configurable IP titles, do not use this document to determine the exact I/O footprint of the core. It is for reference purposes only.

When you configure the core in coreConsultant, you must access the I/O signals for your actual configuration at workspace/report/IO.html or workspace/report/IO.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the I/O signals that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the widths might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the core in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

In addition to describing the function of each signal, the signal descriptions in this chapter include the following information:

Active State: Indicates whether the signal is active high or active low. When a signal is not intended to be used in a particular application, then this signal needs to be tied or driven to the in-active state (opposite of the active state).

Registered: Indicates whether or not the signal is registered directly inside the IP boundary without intervening logic (excluding simple buffers). A value of No does not imply that the signal is not synchronous, only that there is some combinatorial logic between the signal's origin or destination register and the boundary of the controller. A value of N/A indicates that this information is not provided for this IP title.

Synchronous to: Indicates which clock(s) in the IP sample this input (drive for an output) when considering all possible configurations. A particular configuration might not have all of the clocks listed. This clock might not be the same as the clock that your application logic should use to clock (sample/drive) this pin. For more details, consult the clock section in the databook.

Exists: Name of configuration parameter(s) that populates this signal in your configuration.

Validated by: Assertion or de-assertion of signal(s) that validates the signal being described.

Attributes used with Synchronous To

- Clock name - The name of the clock that samples an input or drive and output.
- None - This attribute may be used for clock inputs, hard-coded outputs, feed-through (direct or combinatorial), dangling inputs, unused inputs and asynchronous outputs.
- Asynchronous - This attribute is used for asynchronous inputs and asynchronous resets.

The I/O signals are grouped as follows:

- Slave Interface on page 141
- Master N Interface on page 144
- Test Interface on page 148
- Peripheral Handshaking Interface on page 149
- Interrupt Interface on page 151
- Debug Bus Interface on page 154

4.1 Slave Interface Signals

Table 4-1 Slave Interface Signals

Port Name	I/O	Description
hclk	I	Bus clock. Common clock for all layers **Exists:** Always **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
dma_big_endian_slv	I	Slave endianness. When active high, this signal indicates big endianness on the AMBA layer or the slave interface. When active low, this signal indicates little endianness. **Exists:** DMAH_STATIC_ENDIAN_SELECT==0 **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** 1 for big endian; 0 for little endian
dma_be_select_be32_slv	I	Slave interface Big Endian format selection pin. ■ 0: Big Endian AI format is used for data transfer on slave interface. ■ 1: Big Endian BE-32format is used for data transfer on slave interface **Exists:** DMAH_BE32_SELECTION_PIN_EN==1 **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-1 Slave Interface Signals (Continued)

Port Name	I/O	Description
hresetn	I	Bus reset. Common reset for all layer. Asynchronously asserts and synchronously removed. **Exists:** Always **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low
haddr[(DMAH_HADDR_WIDTH-1):0]	I	Slave port address **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A.
hwdata[(DMAH_S_HDATA_WIDTH-1):0]	I	Slave port write data **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hwrite	I	Slave port transfer direction. When active high, this signal indicates a write transfer. When active low, this signal indicates a read transfer. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** 1 for Write; 0 for Read
hready	I	Current transfer is complete **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
hsize[2:0]	I	Slave port transfer size **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-1 Slave Interface Signals (Continued)

Port Name	I/O	Description
hsel	I	Slave select. Asserted when the current transfer on the AHB is intended for the DW_ahb_dmac **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
htrans[1:0]	I	Slave port transfer control **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hrdata[(DMAH_S_HDATA_WIDTH-1):0]	O	Slave port read data **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hready_resp	O	Transfer complete **Exists:** Always **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
hresp[1:0]	O	Slave port response type from slave. When DW_ahb_dmac is configured for AHB Lite mode and instantiated in an AHB Lite system, the HRESP0 signal is connected to the HRESP signal in the AHB Bus Fabric; the HRESP1 signal is left unconnected. **Exists:** Always **Synchronous To:** hclk **Registered:** DMAH_RETURN_ERR_RESP==1 ? Yes : No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

4.2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals

dma_big_endian_mN -
dma_be_select_be32_mN -
dma_le_select_lli_mN -
hrdataN -
hgrantN -
hrespN -
hreadyN -

- haddrN
- hwdataN
- hwriteN
- hbusreqN
- hlockN
- htransN
- hburstN
- hsizeN
- hprotN

Table 4-2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals

Port Name	I/O	Description
dma_big_endian_mN	I	Master endianness. When active high, this signal indicates big endianness on the AMBA layer or the master interface. When active low, this signal indicates little endianness. **Exists:** DMAH_STATIC_ENDIAN_SELECT==0 **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** 1 for big endian; 0 for little endian
dma_be_select_be32_mN	I	Master interface Big Endian format selection pin. ■ 0: Big Endian AI format is used for data transfer on master interface. ■ 1: Big Endian BE-32format is used for data transfer on master interface **Exists:** DMAH_BE32_SELECTION_PIN_EN==1 **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals (Continued)

Port Name	I/O	Description
dma_le_select_lli_mN	I	Big Endian format selection pin for LLI fetch and status and control write-back for the Master N Interface. 0 : Endian scheme used for LLI fetch and status & control write-back access on master interface is same as that used for data access for master interface. 1 : Little Endian scheme used for LLI fetch and status & control write-back access on master interface irrespective of the endian scheme used for data access on master interface. **Exists:** DMAH_LLI_ENDIAN_SELECTION_PIN_EN==1 **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
hrdataN[(DMAH_M1_HDATA_WIDTH-1):0]	I	Master port read data **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hgrantN	I	Bus grant **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
hrespN[1:0]	I	Master port response type from slave. When DW_ahb_dmac is configured for AHB Lite mode and instantiated in an AHB Lite system, the HRESP0 signal is connected to the HRESP signal in the AHB Bus Fabric; the HRESP1 signal is left unconnected. **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hreadyN	I	Indicates that the current transfer is completed. **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals (Continued)

Port Name	I/O	Description
haddrN[(DMAH_HADDR_WIDTH-1):0]	O	Master port address **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hwdataN[(DMAH_M1_HDATA_WIDTH-1):0]	O	Master port write data **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hwriteN	O	Master port transfer direction. When active high, this signal indicates a write transfer. When active low, this signal indicates a read transfer. **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** 1 for Write; 0 for Read
hbusreqN	O	Bus request **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
hlockN	O	AHB bus lock qualifier. Indicates to the arbiter that the master is performing a number of indivisible transfers. **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
htransN[1:0]	O	Transfer control. Indicates type of current transfer. **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-2 Master N Interface (for N = 1; N <= DMAH_NUM_MASTER_INT) Signals (Continued)

Port Name	I/O	Description
hburstN[2:0]	O	Burst type and length control **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** DMAH_INCR_BURSTS==0 ? Yes : No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hsizeN[2:0]	O	Master port transfer size **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hprotN[3:0]	O	Protection control **Exists:** DMAH_NUM_MASTER_INT >= N **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

4.3 Test Interface Signals

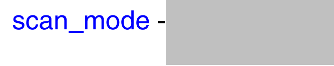


Table 4-3 Test Interface Signals

Port Name	I/O	Description
scan_mode	I	Scan Mode. This signal should be asserted - that is, driven to logic 1 during scan testing and should be deasserted - tied to logic 0 - at all other times. **Exists:** (DMAH_LP_EN==1) || (DMAH_CH_LP_EN==1) **Synchronous To:** Asynchronous **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High.

4.4 Peripheral Handshaking Interface Signals

Table 4-4 Peripheral Handshaking Interface Signals

Port Name	I/O	Description
dma_req[(DMAH_NUM_HS_INT_NZ-1):0]	I	Burst transaction request from peripheral **Exists:** (DMAH_NUM_HS_INT != 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
dma_single[(DMAH_NUM_HS_INT_NZ-1):0]	I	Single transfer status **Exists:** (DMAH_NUM_HS_INT != 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
dma_last[(DMAH_NUM_HS_INT_NZ-1):0]	I	Last transaction in block indicator. **Exists:** (DMAH_NUM_HS_INT != 0) **Synchronous To:** hclk **Registered:** DMAH_REG_HS_IF==1 ? Yes : No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
dma_ack[(DMAH_NUM_HS_INT-1):0]	O	Transaction complete acknowledge signal. **Exists:** (DMAH_NUM_HS_INT != 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-4 Peripheral Handshaking Interface Signals (Continued)

Port Name	I/O	Description
dma_wlast[(DMAH_NUM_MASTER_INT-1):0]	O	Indicates the completion of the last address phase of every destination transaction. It remains asserted until the associated last data phase completes. **Exists:** (DMAH_NUM_HS_INT != 0) && (DMAH_WLAST_EN == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
dma_finish[(DMAH_NUM_HS_INT-1):0]	O	DMA block complete signal. **Exists:** (DMAH_NUM_HS_INT != 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

4.5 Interrupt Interface Signals

Table 4-5 Interrupt Interface Signals

Port Name	I/O	Description
intr[((5*DMAH_NUM_CHANNELS)-1):0]	O	Contents of all five Interrupt Status Registers. x is the number of configured DW_ahb_dmac channels (1 to 8). Bits[x-1:0]- contents of StatTfr Bits[2x-1:x]- contents of StatBlock Bits[3x-1:2x]- contents of StatSrcTrans Bits[4x-1:3x]- contents of StatDstTrans Bits[5x-1:4x]- contents of StatErr **Exists:** (DMAH_INTR_IO == 0) && (DMAH_INTR_POL == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
int_flag[4:0]	O	All internal interrupts for each channel are combined by type for maximum flexibility in supporting driver SoC interrupt schemes. Bit 0: logical OR of all interrupts of type IntTfr Bit 1: logical OR of all interrupts of type IntBlock Bit 2: logical OR of all interrupts of type IntSrcTran Bit 3: logical OR of all interrupts of type IntDstTran Bit 4: logical OR of all interrupts of type IntErr. **Exists:** (DMAH_INTR_IO == 1) && (DMAH_INTR_POL == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-5 Interrupt Interface Signals (Continued)

Port Name	I/O	Description
int_combined	O	Logical OR of all individual interrupts. **Exists:** ((DMAH_INTR_IO == 2) || (DMAH_INTR_IO == 1)) && (DMAH_INTR_POL == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
intr_n[((5*DMAH_NUM_CHANNELS)-1):0]	O	Contents of all five Interrupt Status Registers. x is the number of configured DW_ahb_dmac channels (1 to 8). ■ Bits[x-1:0]- contents of StatTfr ■ Bits[2x-1:x]- contents of StatBlock ■ Bits[3x-1:2x]- contents of StatSrcTrans ■ Bits[4x-1:3x]- contents of StatDstTrans ■ Bits[5x-1:4x]- contents of StatErr **Exists:** (DMAH_INTR_IO == 0) && (DMAH_INTR_POL == 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low
int_flag_n[4:0]	O	All internal interrupts for each channel are combined by type for maximum flexibility in supporting driver SoC interrupt schemes. ■ Bit 0: logical OR of all interrupts of type IntTfr ■ Bit 1: logical OR of all interrupts of type IntBlock ■ Bit 2: logical OR of all interrupts of type IntSrcTran ■ Bit 3: logical OR of all interrupts of type IntDstTran ■ Bit 4: logical OR of all interrupts of type IntErr. **Exists:** (DMAH_INTR_IO == 1) && (DMAH_INTR_POL == 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low
int_combined_n	O	logical OR of all individual interrupts. **Exists:** ((DMAH_INTR_IO == 2) || (DMAH_INTR_IO == 1)) && (DMAH_INTR_POL == 0) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** Low

4.6 Debug Bus Interface Signals

- debug_dmah_busy
- debug_ch_busy
- debug_prevent_deadlock_rfe
- debug_grant_index_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)
- debug_granted_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)
- debug_fifo_ready_src
- debug_fifo_ready_dst
- debug_fifo_half_full
- debug_fifo_empty
- debug_tfr_req_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)
- debug_length_m_i
- debug_dma_data_req
- debug_rd_rawtfr
- debug_rd_rawblock
- debug_rd_rawsrctran
- debug_rd_rawdsttran
- debug_rd_rawerr
- debug_rd_int_en
- debug_rd_masktfr
- debug_rd_maskblock
- debug_rd_masksrctran
- debug_rd_maskdsttran
- debug_rd_maskerr
- debug_statusint_dmacore
- debug_req_miN (for N = 1; N <= DMAH_NUM_MASTER_INT)
- debug_mask_lck_ch_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)
- debug_ch_enable
- debug_ch_enable_reg
- debug_dum_req_dst_region
- debug_dum_req_src_region
- debug_en_src_hs_sgl
- debug_en_dst_hs_sgl
- debug_dma_ctl_en

Table 4-6 Debug Bus Interface Signals

Port Name	I/O	Description
debug_dmah_busy	O	Debug bus, indicates the Low Power State of the DMA. Included on the I/O for verification purposes. These signals should be left disconnected. **Exists:** (DMAH_DEBUG_BUS == 1) && (DMAH_LP_EN == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
debug_ch_busy[(DMAH_NUM_CHANNELS-1):0]	O	Debug bus, indicates the Low Power State of the Channel. Included on the I/O for verification purposes. These signals should be left disconnected. **Exists:** (DMAH_DEBUG_BUS == 1) && (DMAH_CH_LP_EN == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_prevent_deadlock_rfe[(DMAH_NUM_CHANNELS-1):0]	O	Debug signal for deadlock **Exists:** (DMAH_DEBUG_BUS == 1) && ((DMAH_LP_EN == 1) || (DMAH_CH_LP_EN == 1)) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_grant_index_mN[(LOG2_DMAH_NUM_PER-1):0] (for N = 1; N <= DMAH_NUM_MASTER_INT)	O	Debug bus included on the I/O for verification purposes. These signals should be left disconnected **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** DMAH_REMOVE_PIPELINING==1 ? No :Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_granted_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)	O	Debug bus granted to master N **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** DMAH_REMOVE_PIPELINING==1 ? No :Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-6 Debug Bus Interface Signals (Continued)

Port Name	I/O	Description
debug_fifo_ready_src[(DMAH_NUM_CHANNELS-1):0]	O	Source debug fifo ready **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_fifo_ready_dst[(DMAH_NUM_CHANNELS-1):0]	O	Destination debug fifo ready **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_fifo_half_full[(DMAH_NUM_CHANNELS-1):0]	O	Debug fifo half full **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_fifo_empty[(DMAH_NUM_CHANNELS-1):0]	O	Debug fifo empty **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_tfr_req_mN (for N = 1; N <= DMAH_NUM_MASTER_INT)	O	Debug transfer request to master N **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
debug_length_m_i[((DMAH_NUM_MASTER_INT*(MAX_LOG2_FIFO_DEPTH_BYTES+1))-1):0]	O	Debug transfer length **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-6 Debug Bus Interface Signals (Continued)

Port Name	I/O	Description
debug_dma_data_req[(DMAH_NUM_PER-1):0]	O	Debug DMA data request **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_rawtfr[(DMAH_NUM_CHANNELS-1):0]	O	Raw interrupt read output from Tfr **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_rawblock[(DMAH_NUM_CHANNELS-1):0]	O	Raw interrupt read output from Block **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_rawsrctran[(DMAH_NUM_CHANNELS-1):0]	O	Raw interrupt read output from SrcTran **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_rawdsttran[(DMAH_NUM_CHANNELS-1):0]	O	Raw interrupt read output from DstTran **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_rawerr[(DMAH_NUM_CHANNELS-1):0]	O	Raw interrupt read output from Err **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-6 Debug Bus Interface Signals (Continued)

Port Name	I/O	Description
debug_rd_int_en[(DMAH_NUM_CHANNELS-1):0]	O	Interrupt read enable **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_masktfr[(DMAH_NUM_CHANNELS-1):0]	O	Mask interrupt read output from Tfr **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_maskblock[(DMAH_NUM_CHANNELS-1):0]	O	Mask interrupt read output from Block **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_masksrctran[(DMAH_NUM_CHANNELS-1):0]	O	Mask interrupt read output from SrcTran **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_maskdsttran[(DMAH_NUM_CHANNELS-1):0]	O	Mask interrupt read output from DstTran **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_rd_maskerr[(DMAH_NUM_CHANNELS-1):0]	O	Mask interrupt read output from Err **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-6 Debug Bus Interface Signals (Continued)

Port Name	I/O	Description
debug_statusint_dmacore[4:0]	O	Debug status interrupt from DMA core **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_req_miN[(DMAH_NUM_PER-1):0] (for N = 1; N <= DMAH_NUM_MASTER_INT)	O	Debug request from master N **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_mask_lck_ch_mN[(DMAH_NUM_PER-1):0] (for N = 1; N <= DMAH_NUM_MASTER_INT)	O	Lock the transfer channel for master N **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_ch_enable[(DMAH_NUM_CHANNELS-1):0]	O	Debug channel enable **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_ch_enable_reg[(DMAH_NUM_CHANNELS-1):0]	O	Debug channel enable register **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_dum_req_dst_region[(DMAH_NUM_CHANNELS-1):0]	O	Destination debug bus dummy request **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-6 Debug Bus Interface Signals (Continued)

Port Name	I/O	Description
debug_dum_req_src_region[(DMAH_NUM_CHANNELS-1):0]	O	Source debug bus dummy request **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_en_src_hs_sgl[(DMAH_NUM_CHANNELS-1):0]	O	Debug enable source **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_en_dst_hs_sgl[(DMAH_NUM_CHANNELS-1):0]	O	Debug enable destination **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
debug_dma_ctl_en	O	Debug DMA control enable **Exists:** (DMAH_DEBUG_BUS == 1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

5
Register Descriptions

This chapter details all possible registers in the controller. They are arranged hierarchically into maps and blocks (banks). For configurable IP titles, your actual configuration might not contain all of these registers.

Attention: For configurable IP titles, do not use this document to determine the exact attributes of your register map. It is for reference purposes only.

When you configure the controller in coreConsultant, you must access the register attributes for your actual configuration at workspace/report/ComponentRegisters.html or workspace/report/ComponentRegisters.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the registers that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the Offset and Memory Access values might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

Exists Expressions

These expressions indicate the combination of configuration parameters required for a register, field, or block to exist in the memory map. The expression is only valid in the local context and does not indicate the conditions for existence of the parent. For example, the expression for a bit field in a register assumes that the register exists and does not include the conditions for existence of the register.

Offset

The term *Offset* is synonymous with *Address.*

Memory Access Attributes

The Memory Access attribute is defined as <ReadBehavior>/<WriteBehavior> which are defined in the following table.

Table 5-1 Possible Read and Write Behaviors

Read (or Write) Behavior	Description
RC	A read clears this register field.
RS	A read sets this register field.
RM	A read modifies the contents of this register field.
Wo	You can only write to this register once field.
W1C	A write of 1 clears this register field.
W1S	A write of 1 sets this register field.
W1T	A write of 1 toggles this register field.
W0C	A write of 0 clears this register field.
W0S	A write of 0 sets this register field.
W0T	A write of 0 toggles this register field.
WC	Any write clears this register field.
WS	Any write sets this register field.
WM	Any write toggles this register field.
no Read Behavior attribute	You cannot read this register. It is Write-Only.
no Write Behavior attribute	You cannot write to this register. It is Read-Only.

Table 5-2 Memory Access Examples

Memory Access	Description
R	Read-only register field.
W	Write-only register field.
R/W	Read/write register field.
R/W1C	You can read this register field. Writing 1 clears it.
RC/W1C	Reading this register field clears it. Writing 1 clears it.
R/Wo	You can read this register field. You can only write to it once.

Special Optional Attributes

Some register fields might use the following optional attributes.

Table 5-3 Optional Attributes

Attribute	Description
Volatile	As defined by the IP-XACT specification. If true, indicates in the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee as to what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which this field can acquire new values other than by reads/writes/resets and other access methods known to IP-XACT. For example, when the core updates the register field contents.
Testable	As defined by the IP-XACT specification. Possible values are unconstrained, untestable, readOnly, writeAsRead, restore. Untestable means that this field is untestable by a simple automated register test. For example, the read-write access of the register is controlled by a pin or another register. readOnly means that you should not write to this register; only read from it. This might apply for a register that modifies the contents of another register.
Reset Mask	As defined by the IP-XACT specification. Indicates that this register field has an unknown reset value. For example, the reset value is set by another register or an input pin; or the register is implemented using RAM.
* Varies	Indicates that the memory access (or reset) attribute (read, write behavior) is not fixed. For example, the read-write access of the register is controlled by a pin or another register. Or when the access depends on some configuration parameter; in this case the post-configuration report in coreConsultant gives the actual access value.

Component Banks/Blocks

The following table shows the address blocks for each memory map. Follow the link for an address block to see a table of its registers.

Table 5-4 Address Banks/Blocks for Memory Map: DMAC

Address Block	Description
DW_ahb_dmac Channel x register address block (for x = 1; x <= DMAH_NUM_CHANNELS-1) on page 165	DW_ahb_dmac Channel x register address block **Exists:** DMAH_NUM_CHANNELS > 0
Interrupt_Registers on page 203	Interrupt registers **Exists:** Always
Software_Handshake_Registers on page 233	Software Handshaking Registers **Exists:** Always

Table 5-4 Address Banks/Blocks for Memory Map: DMAC (Continued)

Address Block	Description
Miscellaneous_Registers on page 246	Miscellaneous Registers **Exists:** Always

5.1 DMAC/Channel_x_Registers (for x = 1; x <= DMAH_NUM_CHANNELS-1) Registers

Channel registers. Follow the link for the register to see a detailed description of the register.

Table 5-5 Registers for Address Block: DMAC/Channel_x_Registers (for x = 1; x <= DMAH_NUM_CHANNELS-1)

Register	Offset	Description
SARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 167	0x0 + x*0x58	The starting source address is programmed by software before the DMA channel is enabled, or by an...
DARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 168	0x8 + x*0x58	The starting destination address is programmed by software before the DMA channel is enabled, or...
LLPx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 169	0x10 + x*0x58	This register does not exist if the DMAH_CHx_HC_LLP configuration parameter is set to True The LLP...
CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 171	0x18 + x*0x58	This register contains fields that control the DMA transfer. The CTLx register is part of the block...
SSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 182	0x20 + x*0x58	After each block transfer completes, hardware can retrieve the source status information from the...
DSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 184	0x28 + x*0x58	After each block transfer completes, hardware can retrieve the destination status information from...
SSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 186	0x30 + x*0x58	After completion of each block transfer, hardware can retrieve the source status information from...
DSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 188	0x38 + x*0x58	After completion of each block transfer, hardware can retrieve the destination status information...
CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 190	0x40 + x*0x58	This register contains fields that configure the DMA transfer. The channel configuration register...
SGRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 199	0x48 + x*0x58	The Source Gather register contains two fields: - Source gather count field (SGRx.SGC) Specifies...

Table 5-5 Registers for Address Block: DMAC/Channel_x_Registers (for x = 1; x <= DMAH_NUM_CHANNELS-1) (Continued)

Register	Offset	Description
DSRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) on page 201	0x50 + x*0x58	The Destination Scatter register contains two fields: - Destination scatter count field (DSRx.DSC)...

5.1.1 SARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Source Address for Channel x
- **Description:** The starting source address is programmed by software before the DMA channel is enabled, or by an LLI update before the start of the DMA transfer. While the DMA transfer is in progress, this register is updated to reflect the source address of the current AHB transfer.

  Note:You must program the SAR address to be aligned to CTLx.SRC_TR_WIDTH.

  For information on how the DARx is updated at the start of each DMA block for multi-block transfers, refer "Programming Examples".
- **Size:** 64 bits
- **Offset:** 0x0 + x*0x58
- **Exists:** DMAH_NUM_CHANNELS > 0

63:32	31:0
Rsvd_SAR	SAR

Table 5-6 Fields for Register: SARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_SAR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
31:0	SAR	R/W	**Current Source Address of DMA transfer.** Updated after each source transfer. The SINC field in the CTLx register determines whether the address increments, decrements, or is left unchanged on every source transfer through the block transfer. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.1.2 DARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Destination Address Register for Channel x
- **Description:** The starting destination address is programmed by software before the DMA channel is enabled, or by an LLI update before the start of the DMA transfer. While the DMA transfer is in progress, this register is updated to reflect the destination address of the current AHB transfer.

  Note: You must program the DAR to be aligned to CTLx.DST_TR_WIDTH.

  For information on how the DARx is updated at the start of each DMA block for multi-block transfers, refer "Programming Examples".
- **Size:** 64 bits
- **Offset:** 0x8 + x*0x58
- **Exists:** DMAH_NUM_CHANNELS > 0

63:32	31:0
Rsvd_DAR	DAR

Table 5-7 Fields for Register: DARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_DAR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
31:0	DAR	R/W	**Current Destination address of DMA transfer.** Updated after each destination transfer. The DINC field in the CTLx register determines whether the address increments, decrements, or is left unchanged on every destination transfer throughout the block transfer. **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.1.3 LLPx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Linked List Pointer Register for Channel x
- **Description:** This register does not exist if the DMAH_CHx_HC_LLP configuration parameter is set to True

  The LLP register has two functions:

  - The logical result of the equation LLP.LOC !=0 is used to set up the type of DMA transfer - single or multi-block. The Table "Programming of Transfer Types and Channel Register Update Method" shows how the method of updating the channel registers is a function of LLP.LOC != 0. If LLP.LOC is set to 0x0, then transfers using linked lists are not enabled. This register must be programmed prior to enabling the channel in order to set up the transfer type.

  - LLP.LOC != 0 contains the pointer to the next LLI for block chaining using linked lists; refer to "Block Chaining Using Linked Lists". The LLPx register can also point to the address where write-back of the control and source/destination status information occur after block completion.

  Note:You need to program this register to point to the first Linked List Item (LLI) in memory prior to enabling the channel if block chaining is enabled. If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x10 + x*0x58
- **Exists:** !DMAH_CHx_HC_LLP && DMAH_NUM_CHANNELS > x

63:32	31:2	1:0
Rsvd_LLP	LOC	LMS

Table 5-8 Fields for Register: LLPx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_LLP	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-8 Fields for Register: LLPx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:2	LOC	R/W	**Starting Address In Memory** of next LLI if block chaining is enabled. Note that the two LSBs of the starting address are not stored because the address is assumed to be aligned to a 32-bit boundary. LLI accesses are always 32-bit accesses (Hsize=2) aligned to 32-bit boundaries and cannot be changed or programmed to anything other than 32-bit. **Value After Reset:** 0x0 **Exists:** !DMAH_CHx_HC_LLP **Volatile:** true
1:0	LMS	R/W	**List Master Select.** Identifies the AHB layer/interface where the memory device that stores the next linked list item resides. This field does not exist if the configuration parameter is not set to NO_HARDCODE. In this case, the read-back value is always the hardcoded value. The maximum value of this field that can be read back is DMAH_NUM_MASTER_INT-1. **Values:** ■ 0x0 (LIST_MASTER_SELECT_1): The memory device stores the next linked list item on AHB master 1 ■ 0x1 (LIST_MASTER_SELECT_2): The memory device stores the next linked list item on AHB master 2 ■ 0x2 (LIST_MASTER_SELECT_3): The memory device stores the next linked list item on AHB master 3 ■ 0x3 (LIST_MASTER_SELECT_4): The memory device stores the next linked list item on AHB master 4 **Value After Reset:** 0x0 **Exists:** !DMAH_CHx_HC_LLP && DMAH_CHx_LMS == 4 **Volatile:** true

5.1.4 CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Control Register for Channel x
- **Description:** This register contains fields that control the DMA transfer.

  The CTLx register is part of the block descriptor (linked list item - LLI) when block chaining is enabled. It can be varied on a block-by-block basis within a DMA transfer when block chaining is enabled. For information about the behavior of this register between blocks, refer to "Multi-Block Transfers".

  If status write-back is enabled, the upper word of the control register, CTLx[63:32], is written to the control register location of the LLI in system memory at the end of the block transfer.

  Note:You need to program this register prior to enabling the channel.
- **Size:** 64 bits
- **Offset:** 0x18 + x*0x58
- **Exists:** DMAH_NUM_CHANNELS > x

63:45	44	43:y	x:32	31:29	28	27	26:25	24:23	22:20	19	18	17	16:14	13:11	10:9	8:7	6:4	3:1	0
Rsvd_3_CTL	DONE	Rsvd_2_CTL	BLOCK_TS	Rsvd_1_CTL	LLP_SRC_EN	LLP_DST_EN	SMS	DMS	TT_FC	Rsvd_CTL	DST_SCATTER_EN	SRC_GATHER_EN	SRC_MSIZE	DEST_MSIZE	SINC	DINC	SRC_TR_WIDTH	DST_TR_WIDTH	INT_EN

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:45	Rsvd_3_CTL	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
44	DONE	R/W	**Done bit.** If status write-back is enabled, the upper word of the control register, CTLx[63:32], is written to the control register location of the Linked List Item (LLI) in system memory at the end of the block transfer with the done bit set. Software can poll the LLI CTLx.DONE bit to see when a block transfer is complete. The LLI CTLx.DONE bit should be cleared when the linked lists are set up in memory prior to enabling the channel. LLI accesses are always 32-bit accesses (Hsize=2) aligned to 32-bit boundaries and cannot be changed or programmed to anything other than 32-bit. For more information, refer to "Multi-Block Transfers". **Values:** ■ 0x0 (DISABLED): DONE bit is deasserted the enb of block transfer ■ 0x1 (ENABLED): SET the DONE bit at the end of block transfer **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
43:y	Rsvd_2_CTL	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_CH0_MAX_BLK_SIZE_INT + 32

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
x:32	BLOCK_TS	R/W	**Block Transfer Size.** When the DW_ahb_dmac is the flow controller, the user writes this field before the channel is enabled in order to indicate the block size. The number programmed into BLOCK_TS indicates the total number of single transactions to perform for every block transfer; a single transaction is mapped to a single AMBA beat. **Width:** The width of single transaction is determined by CTLx.SRC_TR_WIDTH. For further information on setting this field, refer to "Transfer Operation". Once the transfer starts, the read-back value is the total number of data items already read from the source peripheral, regardless of what is the flow controller. When the source or destination peripheral is assigned as the flow controller, then the maximum block size that can be read back saturates at DMAH_CHx_MAX_BLK_SIZE, but the actual block size can be greater. **Value After Reset:** 0x2 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_CH0_MAX_BLK_SIZE_INT + 31
31:29	Rsvd_1_CTL	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
28	LLP_SRC_EN	R/W	Block chaining is enabled on the source side only if the LLP_SRC_EN field is high and LLPx.LOC is non-zero. For more information, see "Block Chaining using Linked Lists". **Values:** ■ 0x0 (LLP_SRC_DISABLE): Block chaining using Linked List is disabled on the Source side ■ 0x1 (LLP_SRC_ENABLE): Block chaining using Linked List is enabled on the Source side **Value After Reset:** 0x0 **Exists:** !DMAH_CHx_HC_LLP && DMAH_CHx_MULTI_BLK_EN **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
27	LLP_DST_EN	R/W	Block chaining is enabled on the destination side only if LLP_DST_EN field is high and LLPx.LOC is non-zero. For more information, see "Block Chaining using Linked Lists". **Values:** ■ 0x0 (LLP_DST_DISABLE): Block chaining using Linked List is disabled on the Destination side ■ 0x1 (LLP_DST_ENABLE): Block chaining using Linked List is enabled on the Destination side **Value After Reset:** 0x0 **Exists:** !DMAH_CHx_HC_LLP && DMAH_CHx_MULTI_BLK_EN **Volatile:** true
26:25	SMS	R/W	**Source Master Select.** Identifies the Master Interface layer where the source device (peripheral or memory) resides. The maximum value of this field that can be read back is DMAH_NUM_MASTER_INT-1. **Values:** ■ 0x0 (SMS_0): Source device (peripheral or memory) is accessed from AHB master 1 ■ 0x1 (SMS_1): Source device (peripheral or memory) is accessed from AHB master 2 ■ 0x2 (SMS_2): Source device (peripheral or memory) is accessed from AHB master 3 ■ 0x3 (SMS_3): Source device (peripheral or memory) is accessed from AHB master 4 **Value After Reset:** {(DMAH_CTLx_SMS_RST)} **Exists:** DMAH_CHx_SMS == 4 **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
24:23	DMS	R/W	**Destination Master Select.** Identifies the Master Interface layer where the destination device (peripheral or memory) resides. The maximum value of this field that can be read back is DMAH_NUM_MASTER_INT-1. **Values:** ■ 0x0 (DMS_0): Destination device (peripheral or memory) is accessed from AHB master 1 ■ 0x1 (DMS_1): Destination device (peripheral or memory) is accessed from AHB master 2 ■ 0x2 (DMS_2): Destination device (peripheral or memory) is accessed from AHB master 3 ■ 0x3 (DMS_3): Destination device (peripheral or memory) is accessed from AHB master 4 **Value After Reset:** {(DMAH_CTLx_DMS_RST)} **Exists:** DMAH_CHx_DMS == 4 **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
22:20	TT_FC	R/W	**Transfer Type and Flow Control.** Flow control can be assigned to the DW_ahb_dmac, the source peripheral, or the destination peripheral. For more information on transfer types and flow control, refer to "Setup/Operation of the DW_ahb_dmac Transfers". **Dependencies:** If the configuration parameter DMAH_CHx_FC is set to DMA_FC_ONLY, then TT_FC[2] does not exist and TT_FC[2] always reads back 0. If DMAH_CHx_FC is set to SRC_FC_ONLY, then TT_FC[2:1] does not exist and TT_FC[2:1] always reads back 2'b10. If DMAH_CHx_FC is set to DST_FC_ONLY, then TT_FC[2:1] does not exist and TT_FC[2:1] always reads back 2'b11. For multi-block transfers using linked list operation, TT_FC must be constant for all blocks of this multi-block transfer. **Values:** ■ 0x0 (TT_FC_0): Transfer type is Memory to Memory and Flow Controller is DW_ahb_dmac ■ 0x1 (TT_FC_1): Transfer type is Memory to Peripheral and Flow Controller is DW_ahb_dmac ■ 0x2 (TT_FC_2): Transfer type is Peripheral to Memory and Flow Controller is DW_ahb_dmac ■ 0x3 (TT_FC_3): Transfer type is Peripheral to Peripheral and Flow Controller is DW_ahb_dmac ■ 0x4 (TT_FC_4): Transfer type is Peripheral to Memory and Flow Controller is Peripheral ■ 0x5 (TT_FC_5): Transfer type is Peripheral to Peripheral and Flow Controller is Source Peripheral ■ 0x6 (TT_FC_6): Transfer type is Memory to Peripheral and Flow Controller is Peripheral ■ 0x7 (TT_FC_7): Transfer type is Peripheral to Peripheral and Flow Controller is Destination Peripheral **Value After Reset:** {(DMAH_CTLx_TT_FC_RST)} **Exists:** Always **Volatile:** true
19	Rsvd_CTL	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
18	DST_SCATTER_EN	R/W	**Destination scatter enable.** Scatter on the destination side is applicable only when the CTLx.DINC bit indicates an incrementing or decrementing address control. **Values:** ■ 0x0 (DST_SCATTER_DISABLE): Destination Scatter is disabled ■ 0x1 (DST_SCATTER_ENABLE): Destination Scatter is enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_DST_SCA_EN **Volatile:** true
17	SRC_GATHER_EN	R/W	**Source gather enable.** Gather on the source side is applicable only when the CTLx.SINC bit indicates an incrementing or decrementing address control. **Values:** ■ 0x0 (SRC_GATHER_DISABLE): Source gather is disabled ■ 0x1 (SRC_GATHER_ENABLE): Source gather is enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_SRC_GAT_EN **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
16:14	SRC_MSIZE	R/W	**Source Burst Transaction Length.** Number of data items, each of width CTLx.SRC_TR_WIDTH, to be read from the source every time a burst transferred request is made from either the corresponding hardware or software handshaking interface. **NOTE:** This value is not related to the AHB bus master HBURST bus. For information on the decoding for this field, see the "Setting Up Transfers" section and for more inforamtion about this field, see the "Choosing the Receive Watermark level" section in the DW_ahb_dmac Databook. **Values:** ■ 0x0 (SRC_MSIZE_0): Number of data items to be transferred is 1 ■ 0x1 (SRC_MSIZE_1): Number of data items to be transferred is 4 ■ 0x2 (SRC_MSIZE_2): Number of data items to be transferred is 8 ■ 0x3 (SRC_MSIZE_3): Number of data items to be transferred is 16 ■ 0x4 (SRC_MSIZE_4): Number of data items to be transferred is 32 ■ 0x5 (SRC_MSIZE_5): Number of data items to be transferred is 64 ■ 0x6 (SRC_MSIZE_6): Number of data items to be transferred is 128 ■ 0x7 (SRC_MSIZE_7): Number of data items to be transferred is 256 **Value After Reset:** 0x1 **Exists:** Always **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
13:11	DEST_MSIZE	R/W	**Destination Burst Transaction Length.** Number of data items, each of width CTLx.DST_TR_WIDTH, to be written to the destination every time a destination burst transaction request is made from either the corresponding hardware or software handshaking interface. **NOTE:** This value is not related to the AHB bus master HBURST bus. **Values:** ■ 0x0 (DEST_MSIZE_0): Number of data items to be transferred is 1 ■ 0x1 (DEST_MSIZE_1): Number of data items to be transferred is 4 ■ 0x2 (DEST_MSIZE_2): Number of data items to be transferred is 8 ■ 0x3 (DEST_MSIZE_3): Number of data items to be transferred is 16 ■ 0x4 (DEST_MSIZE_4): Number of data items to be transferred is 32 ■ 0x5 (DEST_MSIZE_5): Number of data items to be transferred is 64 ■ 0x6 (DEST_MSIZE_6): Number of data items to be transferred is 128 ■ 0x7 (DEST_MSIZE_7): Number of data items to be transferred is 256 **Value After Reset:** 0x1 **Exists:** Always **Volatile:** true
10:9	SINC	R/W	**Source Address Increment.** Indicates whether to increment or decrement the source address on every source transfer. If the device is fetching data from a source peripheral FIFO with a fixed address, then set this field to "No change". **Values:** ■ 0x0 (SINC_0): Increments the source address ■ 0x1 (SINC_1): Decrements the source address ■ 0x2 (SINC_2): No change in the source address ■ 0x3 (SINC_3): No change in the source address **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
8:7	DINC	R/W	**Destination Address Increment.** Indicates whether to increment or decrement the destination address on every destination transfer. If your device is writing data to a destination peripheral FIFO with a fixed address, then set this field to \"No Change\". **Values:** ■ 0x0 (DINC_0): Increments the destination address ■ 0x1 (DINC_1): Decrements the destination address ■ 0x2 (DINC_2): No change in the destination address ■ 0x3 (DINC_3): No change in the destination address **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
6:4	SRC_TR_WIDTH	R/W	**Source Transfer Width.** Mapped to AHB bus hsize. For a non-memory peripheral, typically the peripheral (source) FIFO width. This value must be less than or equl to DMAH_Mk_HDATA_WIDTH, where k is the AHB layer 1 to 4 where the source resides. **Values:** ■ 0x0 (SRC_TR_WIDTH_0): Source transfer width is 8 bits ■ 0x1 (SRC_TR_WIDTH_1): Source transfer width is 16 bits ■ 0x2 (SRC_TR_WIDTH_2): Source transfer width is 32 bits ■ 0x3 (SRC_TR_WIDTH_3): Source transfer width is 64 bits ■ 0x4 (SRC_TR_WIDTH_4): Source transfer width is 128 bits ■ 0x5 (SRC_TR_WIDTH_5): Source transfer width is 256 bits ■ 0x6 (SRC_TR_WIDTH_6): Source transfer width is 256 bits ■ 0x7 (SRC_TR_WIDTH_7): Source transfer width is 256 bits **Value After Reset:** {(DMAH_CTLx_SRC_TR_RST)} **Exists:** !DMAH_CHx_STW **Volatile:** true

Table 5-9 Fields for Register: CTLx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
3:1	DST_TR_WIDTH	R/W	**Destination Transfer Width.** Mapped to AHB bus hsize. For a non-memory peripheral, typically the peripheral (destination) FIFO width. This value must be less than or equal to DMAH_Mk_HDATA_WIDTH where k is the AHB layer 1 to 4 where the destination resides. For the decoding of this field, see the "Setting Up Transfers" section in the DW_ahb_dmac Databook. **Values:** ■ 0x0 (DST_TR_WIDTH_0): Destination transfer width is 8 bits ■ 0x1 (DST_TR_WIDTH_1): Destination transfer width is 16 bits ■ 0x2 (DST_TR_WIDTH_2): Destination transfer width is 32 bits ■ 0x3 (DST_TR_WIDTH_3): Destination transfer width is 64 bits ■ 0x4 (DST_TR_WIDTH_4): Destination transfer width is 128 bits ■ 0x5 (DST_TR_WIDTH_5): Destination transfer width is 256 bits ■ 0x6 (DST_TR_WIDTH_6): Destination transfer width is 256 bits ■ 0x7 (DST_TR_WIDTH_7): Destination transfer width is 256 bits **Value After Reset:** {(DMAH_CTLx_DST_TR_RST)} **Exists:** !DMAH_CHx_DTW **Volatile:** true
0	INT_EN	R/W	**Interrupt Enable Bit.** If set, then all interrupt-generating sources are enabled. Functions as a global mask bit for all interrupts for the channel; raw* interrupt registers still assert if CTLx.INT_EN=0. **Values:** ■ 0x0 (INTERRUPT_DISABLE): Interrupt is disabled ■ 0x1 (INTERRUPT_ENABLE): Interrupt is enabled **Value After Reset:** 0x1 **Exists:** Always **Volatile:** true

5.1.5 SSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Source Status Register for Channel x
- **Description:** After each block transfer completes, hardware can retrieve the source status information from the address pointed to by the contents of the SSTATARx register. This status information is then stored in the SSTATx register and written out to the SSTATx register location of the LLI before the start of the next block. For conditions under which the source status information is fetched, refer to "Multi-Block Transfers". This register does not exist if DMAH_CHx_STAT_SRC is set to False; in this case, the read-back value is always 0.

  Note: This register is a temporary placeholder for the source status information on its way to the SSTATx register location of the LLI. The source status information should be retrieved by software from the SSTATx register location of the LLI, and not by a read of this register over the DW_ahb_dmac slave interface.

  If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x20 + x*0x58
- **Exists:** DMAH_CHx_STAT_SRC && DMAH_NUM_CHANNELS > x

63:32	31:0
Rsvd_1_SSTAT	SSTAT

Table 5-10 Fields for Register: SSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_SSTAT	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-10 Fields for Register: SSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:0	SSTAT	R/W	Source status information retrieved by hardware from the address pointed to by the contents of the STATARx register **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_SRC **Volatile:** true

5.1.6 DSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Destination Status Register for Channel x
- **Description:** After each block transfer completes, hardware can retrieve the destination status information from the address pointed to by the contents of the DSTATARx register. This status information is then stored in the DSTATx register and written out to the DSTATx register location of the LLI before the start of the next block. For conditions under which the destination status information is fetched, refer to "Multi-Block Transfers". This register does not exist if DMAH_CHx_STAT_DST is set to False; in this case, the read-back value is always 0.

  Note:This register is a temporary placeholder for the destination status information on its way to the DSTATx register location of the LLI. The destination status information should be retrieved by software from the DSTATx register location of the LLI, and not by a read of this register over the DW_ahb_dmac slave interface.

  If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x28 + x*0x58
- **Exists:** DMAH_CHx_STAT_DST && DMAH_NUM_CHANNELS > x

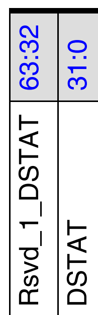


Table 5-11 Fields for Register: DSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_DSTAT	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-11 Fields for Register: DSTATx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:0	DSTAT	R/W	Destination status information retrieved by hardware from the address pointed to by the contents of DSTATARx register. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_DST **Volatile:** true

5.1.7 SSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Source Status Address Register for Channel x
- **Description:** After completion of each block transfer, hardware can retrieve the source status information from the user-defined address to which the contents of the SSTATARx register point. The user can select any location in system memory that would provide a 32-bit value to indicate the status of the source transfer. For example, if the DW_apb_ssi is the source peripheral for the DMA transfer, the user can use one of the SSI registers to indicate the status of the transfer. Thus the address programmed in SSTATARx could be the address of the SSI.CTRL register or the SSI.ISR register, or it could be the address of the SSI.RXFLR register.

  Note: If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".

  This register does not exist if the configuration parameter DMAH_CHx_STAT_SRC is set to False; in this case, the read-back value is always 0.
- **Size:** 64 bits
- **Offset:** 0x30 + x*0x58
- **Exists:** DMAH_CHx_STAT_SRC && DMAH_NUM_CHANNELS > x

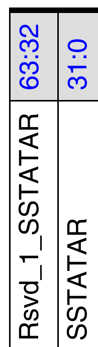


Table 5-12 Fields for Register: SSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_SSTATAR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-12 Fields for Register: SSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:0	SSTATAR	R/W	Pointer from where hardware can fetch the source status information, which is registered in the SSTATx register and written out to the SSTATx register location of the LLI before the start of the next block. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_SRC

5.1.8 DSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Destination Status Address Register for Channel x
- **Description:** After completion of each block transfer, hardware can retrieve the destination status information from the user-defined address to which the contents of the DSTATARx register point. The user can select any location in system memory that would provide a 32-bit value to indicate the status of the destination transfer. For example, if the DW_apb_ssi is the destination peripheral for the DMA transfer, the user can use one of the SSI registers to indicate the status of the transfer. Thus the address programmed in DSTATARx could be the address of the SSI.CTRL register or the SSI.ISR register, or it could be the address of the SSI.TXFLR register.

  Note: If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".

  This register does not exist if the configuration parameter DMAH_CHx-STAT_DST is set to False; in this case, the read-back value is always 0.
- **Size:** 64 bits
- **Offset:** 0x38 + x*0x58
- **Exists:** DMAH_CHx_STAT_DST && DMAH_NUM_CHANNELS > x

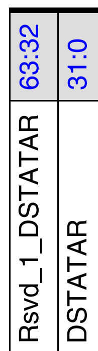


Table 5-13 Fields for Register: DSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_DSTATAR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-13 Fields for Register: DSTATARx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:0	DSTATAR	R/W	Pointer from where hardware can fetch the destination status information, which is registered in the DSTATx register and written out to the DSTATx register location of the LLI before the start of the next block. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_DST

5.1.9 CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Configuration Register for Channel x
- **Description:** This register contains fields that configure the DMA transfer. The channel configuration register remains fixed for all blocks of a multi-block transfer.

  Note: You need to program this register prior to enabling the channel.
- **Size:** 64 bits
- **Offset:** 0x40 + x*0x58
- **Exists:** DMAH_NUM_CHANNELS > x

63:47	46:y	x:43	42:y	x:39	38	37	36:34	33	32	31	30	29:20	19	18	17	16	15:14	13:12	11	10	9	8	7:5	4:0
Rsvd_3_CFG	Rsvd_2_CFG	DEST_PER	Rsvd_1_CFG	SRC_PER	SS_UPD_EN	DS_UPD_EN	PROTCTL	FIFO_MODE	FCMODE	RELOAD_DST	RELOAD_SRC	MAX_ABRST	SRC_HS_POL	DST_HS_POL	LOCK_B	LOCK_CH	LOCK_B_L	LOCK_CH_L	HS_SEL_SRC	HS_SEL_DST	FIFO_EMPTY	CH_SUSP	CH_PRIOR	Rsvd_CFG

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:47	Rsvd_3_CFG	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always
46:y	Rsvd_2_CFG	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** LOG2_DMAH_NUM_HS_INT + 43

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
x:43	DEST_PER	R/W	**Destination hardware interface.** Assigns a hardware handshaking interface (0 : DMAH_NUM_HS_INT-1) to the destination of channel x if the CFGx.HS_SEL_DST field is 0; otherwise, this field is ignored. The channel can then communicate with the destination peripheral connected to that interface through the assigned hardware handshaking interface. **NOTE1:** For correct DW_ahb_dmac operation, only one peripheral (source or destination) should be assigned to the same handshaking interface. **NOTE2:** This field does not exist if the configuration parameter DMAH_NUM_HS_INT is set to 0. **Value After Reset:** 0x0 **Exists:** DMAH_NUM_HS_INT > 0 **Range Variable[x]:** LOG2_DMAH_NUM_HS_INT + 42
42:y	Rsvd_1_CFG	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** LOG2_DMAH_NUM_HS_INT + 39
x:39	SRC_PER	R/W	**Source Hardware Interface.**. Assigns a hardware handshaking interface (0 : DMAH_NUM_HS_INT-1) to the source of channel x if the CFGx.HS_SEL_SRC field is 0; otherwise, this field is ignored. The channel can then communicate with the source peripheral connected to that interface through the assigned hardware handshaking interface. **NOTE1:** For correct DW_ahb_dmac operation, only one peripheral (source or destination) should be assigned to the same handshaking interface. **NOTE2:** This field does not exist if the configuration parameter DMAH_NUM_HS_INT is set to 0. **Value After Reset:** 0x0 **Exists:** DMAH_NUM_HS_INT > 0 **Range Variable[x]:** LOG2_DMAH_NUM_HS_INT + 38

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
38	SS_UPD_EN	R/W	**Source Status Update Enable.** Source status information is fetched only from the location pointed to by the SSTATARx register, stored in the SSTATx register and written out to the SSTATx location of the LLI, if SS_UPD_EN is high. **NOTE:** This enalbe is applicable only if DMAH_CHx_STAT_SRC is set to True. This field does not exist if the configuration parameter DMAH_CHx_STAT_SRC is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLED): Source Status Update is disabled. ■ 0x1 (ENABLED): Source Status Update is enabled. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_SRC
37	DS_UPD_EN	R/W	**Destination Status Update Enable.** Destination status information is fetched only from the location pointed to by the DSTATARx register, stored in the DSTATx register and written out to the DSTATx location of the LLI, if DS_UPD_EN is high. This field does not exist if the configuration parameter DMAH_CHx_SATAT_DST is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLED): Destination Status Update is disabled ■ 0x1 (ENABLED): Destination Status Update is enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_STAT_DST
36:34	PROTCTL	R/W	**Protection Control** bits used to drive the AHB HPROT[3:1] bus. The AMBA Specification recommends that the default of HPROT indicates a non-cached, non-buffered, privileged data access. The reset value is used to indicate such an access. HPROT[0] is tied high because all transfers are data accesses, as there are no opcode fetches. There is a one-to-one mapping of these register bits to the HPROT[3:1] master interface signals. Mapping of HPROT bus is as follows: ■ 1'b1 to HPROT[0] ■ CFGx.PROTCTL[1] to HPROT[1] ■ CFGx.PROTCTL[2] to HPROT[2] ■ CFGx.PROTCTL[3] to HPROT[3] **Value After Reset:** 0x1 **Exists:** Always

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
33	FIFO_MODE	R/W	**FIFO Mode Select.** Determines how much space or data needs to be available in the FIFO before a burst transaction request is serviced. **Values:** ■ 0x0 (FIFO_MODE_0): Space/data available for single AHB transfer of the specified transfer width ■ 0x1 (FIFO_MODE_1): Data available is greater than or equal to half the FIFO depth for destination transfers and space available is greater than half the fifo depth for source transfers. The exceptions are at the end of a burst transaction request or at the end of a block transfer. **Value After Reset:** 0x0 **Exists:** Always
32	FCMODE	R/W	**Flow Control Mode.** Determines when source transaction requests are serviced when the Destination Peripheral is the flow controller. **Values:** ■ 0x0 (FCMODE_0): Source transaction requests are serviced when they occur. Data pre-fetching is enabled ■ 0x1 (FCMODE_1): Source transaction requests are not serviced until a destination transaction request occurs. In this mode, the amount of data transferred from the source is limited so that it is guaranteed to be transferred to the destination prior to block termination by the destination. Data pre-fetching is disabled. **Value After Reset:** 0x0 **Exists:** Always
31	RELOAD_DST	* Varies	**Automatic Destination Reload.** The DARx register can be automatically reloaded from its initial value at the end of every block for multi-block transfers. A new block transfer is then initiated. This register does not exist if the configuration parameter DMAH_CHx_MULTI_BLK_EN is not selected; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLE): Destination Reload Disabled. ■ 0x1 (ENABLE): Destination Reload Enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_MULTI_BLK_EN **Memory Access:** {(DMAH_CH0_RELOAD_DST_HC == 0) ? "read-write" : "read-only"}

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
30	RELOAD_SRC	* Varies	**Automatic Source Reload.** The SARx register can be automatically reloaded from its initial value at the end of every block for multi-block transfers. A new block transfer is then initiated. This field does not exist if the configuration parameter DMAH_CHx_MULTI_BLK_EN is not selected; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLE): Source Reload Disabled ■ 0x1 (ENABLE): Source Reload Enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_MULTI_BLK_EN **Memory Access:** {(DMAH_CH0_RELOAD_SRC_HC == 0) ? "read-write" : "read-only"}
29:20	MAX_ABRST	R/W	**Maximum AMBA Burst Length.** Maximum AMBA burst length that is used for DMA transfers on this channel. A value of 0 indicates that software is not limiting the maximum AMBA burst length for DMA transfers on this channel. This field does not exist if the configuration parameter DMAH_MABRST is not selected; in this case, the read-back value is always 0, and the maximum AMBA burst length cannot be limited by software. **Value After Reset:** 0x0 **Exists:** DMAH_MABRST
19	SRC_HS_POL	R/W	**Source Handshaking Interface Polarity.** **Values:** ■ 0x0 (ACTIVE_HIGH): Source Handshaking Interface Polarity is Active high ■ 0x1 (ACTIVE_LOW): Source Handshaking Interface Polarity is Active low **Value After Reset:** 0x0 **Exists:** Always
18	DST_HS_POL	R/W	**Destination Handshaking Interface Polarity.** **Values:** ■ 0x0 (ACTIVE_HIGH): Destination Handshaking Interface Polarity is Active high ■ 0x1 (ACTIVE_LOW): Destination Handshaking Interface Polarity is Active low **Value After Reset:** 0x0 **Exists:** Always

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
17	LOCK_B	R/W	**Bus Lock Bit.** When active, the AHB bus master signal hlock is asserted for the duration specified in CFGx.LOCK_B_L. For more information, refer to "Locked DMA Transfers". This field does not exist if the configuration parameter DMAH_CHx_LOCK_EN is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLED): Bus lock bit is not enabled ■ 0x1 (ENABLED): Bus lock bit is enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_LOCK_EN
16	LOCK_CH	R/W	**Channel Lock Bit.** When the channel is granted control of the master bus interface and if the CFGx.LOCK_CH bit is asserted, then no other channels are granted control of the master bus interface for the duration specified in CFGx.LOCK_CH_L. Indicates to the master bus interface arbiter that this channel wants exclusive access to the master bus interface for the duration specified in CFGx.LOCK_CH_L. This field does not exist if the configuration parameter DMAH_CHx_LOCK_EN is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (DISABLED): Channel lock bit is not enabled ■ 0x1 (ENABLED): Channel lock bit is enabled **Value After Reset:** 0x0 **Exists:** DMAH_CHx_LOCK_EN
15:14	LOCK_B_L	R/W	**Bus lock level.** Indicates the duration over which CFGx.LOCK_B bit applies. This field does not exist if the parameter DMAH_CHx_LOCK_EN is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (LOCK_B_L_0): Over complete DMA transfer ■ 0x1 (LOCK_B_L_1): Over complete DMA block transfer ■ 0x2 (LOCK_B_L_2): Over complete DMA transaction ■ 0x3 (LOCK_B_L_3): Over complete DMA transaction **Value After Reset:** 0x0 **Exists:** DMAH_CHx_LOCK_EN

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
13:12	LOCK_CH_L	R/W	**Channel Local Level.** Indicates the duration over which CFGx.LOCK_CH applies. This field does not exist if the configuration parameter DMAH_CHx_LOCK_EN is set to False; in this case, the read-back value is always 0. **Values:** ■ 0x0 (LOCK_CH_L_0): Over complete DMA transfer ■ 0x1 (LOCK_CH_L_1): Over complete DMA block transfer ■ 0x2 (LOCK_CH_L_2): Over complete DMA transaction ■ 0x3 (LOCK_CH_L_3): Over complete DMA transaction **Value After Reset:** 0x0 **Exists:** DMAH_CHx_LOCK_EN
11	HS_SEL_SRC	R/W	**Source Software or Hardware Handshaking Select.** This register selects which of the handshaking interfaces - hardware or software - is active for source requests on this channel. If the source peripheral is memory, then this bit is ignored. **Values:** ■ 0x0 (HARDWARE_HS): Hardware handshaking interface. Software initiated transaction requests are ignored. ■ 0x1 (SOFTWARE_HS): Software handshaking interface. Hardware initiated transaction requests are ignored. **Value After Reset:** 0x1 **Exists:** Always
10	HS_SEL_DST	R/W	**Destination Software or Hardware Handshaking Select.** This register selects which of the handshaking interfaces - hardware or software - is active for destination requests on this channel. If the destination peripheral is memory, then this bit is ignored. **Values:** ■ 0x0 (HARDWARE_HS): Hardware handshaking interface. Software initiated transaction requests are ignored. ■ 0x1 (SOFTWARE_HS): Software handshaking interface. Hardware initiated transaction requests are ignored. **Value After Reset:** 0x1 **Exists:** Always

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
9	FIFO_EMPTY	R	**Channel FIFO status.** Indicates if there is data left in the channel FIFO. Can be used in conjunction with CFGx.CH_SUSP to cleanly disable a channel. For more information, refer to "Disabling a Channel Prior to Transfer Completion". **Values:** ■ 0x0 (NOT_EMPTY): Channel FIFO is not empty ■ 0x1 (EMPTY): Channel FIFO is empty **Value After Reset:** 0x1 **Exists:** Always
8	CH_SUSP	R/W	**Channel Suspend.** Suspends all DMA data transfers from the source until this bit is cleared. There is no guarantee that the current transaction will complete. Can also be used in conjunction with CFGx.FIFO_EMPTY to cleanly disable a channel without losing any data. For more information, refer to "Disabling a Channel Prior to Transfer Completion". **Values:** ■ 0x0 (NOT_SUSPENDED): DMA transfer from the source is not suspended ■ 0x1 (SUSPENDED): Suspend DMA transfer from the source **Value After Reset:** 0x0 **Exists:** Always
7:5	CH_PRIOR	R/W	**Channel Priority.** A priority of 7 is the highest priority, and 0 is the lowest. This field must be programmed within the range 0 to DMAH_NUM_CHANNELS-1. A programmed value outside this range will cause erroneous behavior. **Values:** ■ 0x0 (CH_PRIOR_0): Channel priority is 0 ■ 0x1 (CH_PRIOR_1): Channel priority is 1 ■ 0x2 (CH_PRIOR_2): Channel priority is 2 ■ 0x3 (CH_PRIOR_3): Channel priority is 3 ■ 0x4 (CH_PRIOR_4): Channel priority is 4 ■ 0x5 (CH_PRIOR_5): Channel priority is 5 ■ 0x6 (CH_PRIOR_6): Channel priority is 6 ■ 0x7 (CH_PRIOR_7): Channel priority is 7 **Value After Reset:** x **Exists:** Always

Table 5-14 Fields for Register: CFGx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
4:0	Rsvd_CFG	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always

5.1.10 SGRx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Source Gather Register for Channel x
- **Description:** The Source Gather register contains two fields:
  - Source gather count field (SGRx.SGC) Specifies the number of contiguous source transfers of CTLx.SRC_TR_WIDTH between successive gather intervals. This is defined as a gather boundary.
  - Source gather interval field (SGRx.SGI) Specifies the source address increment/decrement in multiples of CTLx.SRC_TR_WIDTH on a gather boundary when gather mode is enabled for the source transfer.

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".

  The CTLx.SINC field controls whether the address increments or decrements. When the CTLx.SINC field indicates a fixed-address control, then the address remains constant throughout the transfer and the SGRx register is ignored. This register does not exist if the configuration parameter DMAH_CHx_SRC_GAT_EN is set to False. For more information, see "Scatter/Gather".
- **Size:** 64 bits
- **Offset:** 0x48 + x*0x58
- **Exists:** DMAH_CHx_SRC_GAT_EN && DMAH_NUM_CHANNELS > x

63:32	31:y	x:20	19:0
Rsvd_1_SGR	Rsvd_SGR	SGC	SGI

Table 5-15 Fields for Register: SGRx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_SGR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-15 Fields for Register: SGRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
31:y	Rsvd_SGR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_CH0_MAX_BLK_SIZE_INT + 20
x:20	SGC	R/W	Source Gather Count. Source contiguous tranfer count between successive gather boundaries. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_SRC_GAT_EN **Range Variable[x]:** DMAH_CH0_MAX_BLK_SIZE_INT + 19
19:0	SGI	R/W	Source Gather Interval. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_SRC_GAT_EN

5.1.11 DSRx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

- **Name:** Destination Scatter Register for Channel x
- **Description:** The Destination Scatter register contains two fields:
  - Destination scatter count field (DSRx.DSC) Specifies the number of contiguous destination transfers of CTLx.DST_TR_WIDTH between successive scatter boundaries.
  - Destination scatter interval field (DSRx.DSI) Specifies the destination address increment/decrement in multiples of CTLx.DST_TR_WIDTH on a scatter boundary when scatter mode is enabled for the destination transfer.

  Note: If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".

  The CTLx.DINC field controls whether the address increments or decrements. When the CTLx.DINC field indicates a fixed address control, then the address remains constant throughout the transfer and the DSRx register is ignored. This register does not exist if the configuration parameter DMAH_CHx_DST_SCA_EN is set to False. For more information, see "Scatter/Gather".
- **Size:** 64 bits
- **Offset:** 0x50 + x*0x58
- **Exists:** DMAH_CHx_DST_SCA_EN && DMAH_NUM_CHANNELS > x

63:32	31:y	x:20	19:0
Rsvd_1_DSR	Rsvd_DSR	DSC	DSI

Table 5-16 Fields for Register: DSRx (for x = 0; x <= DMAH_NUM_CHANNELS-1)

Bits	Name	Memory Access	Description
63:32	Rsvd_1_DSR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always
31:y	Rsvd_DSR	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_CH0_MAX_BLK_SIZE_INT + 20

Table 5-16 Fields for Register: DSRx (for x = 0; x <= DMAH_NUM_CHANNELS-1) (Continued)

Bits	Name	Memory Access	Description
x:20	DSC	R/W	Destination Scatter Count. Destination contiguous transfer count between successive scatter boundaries. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_DST_SCA_EN **Range Variable[x]:** DMAH_CH0_MAX_BLK_SIZE_INT + 19
19:0	DSI	R/W	Destination Scatter Interval. **Value After Reset:** 0x0 **Exists:** DMAH_CHx_DST_SCA_EN

5.2 DMAC/Interrupt Registers

Interrupt registers. Follow the link for the register to see a detailed description of the register.

Table 5-17 Registers for Address Block: DMAC/Interrupt_Registers

Register	Offset	Description
RawTfr on page 205	0x2c0	Interrupt events are stored in this Raw Interrupt Status register before masking. This register...
RawBlock on page 206	0x2c8	Interrupt events are stored in this Raw Interrupt Status register before masking. This register...
RawSrcTran on page 207	0x2d0	Interrupt events are stored in this Raw Interrupt Status register before masking. This register...
RawDstTran on page 209	0x2d8	Interrupt events are stored in this Raw Interrupt Status register before masking. This register...
RawErr on page 210	0x2e0	Interrupt events are stored in this Raw Interrupt Status register before masking. This register...
StatusTfr on page 211	0x2e8	Channel DMA Transfer complete interrupt event from all channels is stored in this Interrupt Status...
StatusBlock on page 212	0x2f0	Channel Block complete interrupt event from all channels is stored in this Interrupt Status register...
StatusSrcTran on page 213	0x2f8	Channel Source Transaction complete interrupt event from all channels is stored in this Interrupt...
StatusDstTran on page 214	0x300	Channel destination transaction complete interrupt event from all channels is stored in this Interrupt...
StatusErr on page 215	0x308	Channel Error interrupt event from all channels is stored in this Interrupt Status register after...
MaskTfr on page 216	0x310	The contents of the Raw Status register RawTfr is masked with the contents of the Mask register...
MaskBlock on page 218	0x318	The contents of the Raw Status register RawBlock is masked with the contents of the Mask register...
MaskSrcTran on page 220	0x320	The contents of the Raw Status register RawSrcTran is masked with the contents of the Mask register...
MaskDstTran on page 222	0x328	The contents of the Raw Status register RawDstTran is masked with the contents of the Mask register...
MaskErr on page 224	0x330	The contents of the Raw Status register RawErr is masked with the contents of the Mask register...
ClearTfr on page 226	0x338	Each bit in the RawTfr and StatusTfr is cleared on the same cycle by writing a 1 to the corresponding...

Table 5-17 Registers for Address Block: DMAC/Interrupt_Registers (Continued)

Register	Offset	Description
ClearBlock on page 227	0x340	Each bit in the RawBlock and StatusBlock is cleared on the same cycle by writing a 1 to the corresponding...
ClearSrcTran on page 228	0x348	Each bit in the RawSrcTran and StatusSrcTran is cleared on the same cycle by writing a 1 to the...
ClearDstTran on page 229	0x350	Each bit in the RawDstTran and StatusDstTran is cleared on the same cycle by writing a 1 to the...
ClearErr on page 230	0x358	Each bit in the RawErr and StatusErr is cleared on the same cycle by writing a 1 to the corresponding...
StatusInt on page 231	0x360	The contents of each of the five Status registers StatusTfr, StatusBlock, StatusSrcTran, StatusDstTran,...

5.2.1 RawTfr

- **Name:** Raw Status for IntTfr Interrupt
- **Description:** Interrupt events are stored in this Raw Interrupt Status register before masking. This register has a bit allocated per channel; for example, RawTfr[2] is the Channel 2 raw transfer complete interrupt.

  Each bit in this register is cleared by writing a 1 to the corresponding location in the ClearTfr register.

  Note: Write access is available to this register or software testing purposes only. Under normal operation, writes to this register are not recommended.
- **Size:** 64 bits
- **Offset:** 0x2c0
- **Exists:** Always

63:y	x:0
Rsvd_RawTfr	RAW

Table 5-18 Fields for Register: RawTfr

Bits	Name	Memory Access	Description
63:y	Rsvd_RawTfr	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	RAW	R/W	Raw Status for IntTfr Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Raw Interrupt Status ■ 0x1 (ACTIVE): Active Raw Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.2 RawBlock

- **Name:** Raw Status for IntBlock Interrupt
- **Description:** Interrupt events are stored in this Raw Interrupt Status register before masking. This register has a bit allocated per channel; for example, RawBlock[2] is the Channel 2 raw block complete interrupt.

  Each bit in this register is cleared by writing a 1 to the corresponding location in the ClearBlock register.

  Note:Write access is available to this register or software testing purposes only. Under normal operation, writes to this register are not recommended.
- **Size:** 64 bits
- **Offset:** 0x2c8
- **Exists:** Always

63:y	x:0
Rsvd_RawBlock	RAW

Table 5-19 Fields for Register: RawBlock

Bits	Name	Memory Access	Description
63:y	Rsvd_RawBlock	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	RAW	R/W	Raw Status for IntBlock Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Raw Interrupt Status ■ 0x1 (ACTIVE): Active Raw Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.3 RawSrcTran

- **Name:** Raw Status for IntSrcTran Interrupt
- **Description:** Interrupt events are stored in this Raw Interrupt Status register before masking. This register has a bit allocated per channel; for example, RawSrcTran[2] is the Channel 2 raw source transaction complete interrupt.

  Each bit in this register is cleared by writing a 1 to the corresponding location in the ClearSrcTran register.

  Note:Write access is available to this register or software testing purposes only. Under normal operation, writes to this register are not recommended.
- **Size:** 64 bits
- **Offset:** 0x2d0
- **Exists:** Always

63:y	x:0
Rsvd_RawSrcTran	RAW

Table 5-20 Fields for Register: RawSrcTran

Bits	Name	Memory Access	Description
63:y	Rsvd_RawSrcTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS

Table 5-20 Fields for Register: RawSrcTran (Continued)

Bits	Name	Memory Access	Description
x:0	RAW	R/W	Raw Status for IntSrcTran Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Raw Interrupt Status ■ 0x1 (ACTIVE): Active Raw Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.4 RawDstTran

- **Name:** Raw Status for IntDstTran Interrupt
- **Description:** Interrupt events are stored in this Raw Interrupt Status register before masking. This register has a bit allocated per channel; for example, RawDstTran[2] is the Channel 2 raw destination transaction complete interrupt.

  Each bit in this register is cleared by writing a 1 to the corresponding location in the ClearDstTran register.

  Note: Write access is available to this register or software testing purposes only. Under normal operation, writes to this register are not recommended.
- **Size:** 64 bits
- **Offset:** 0x2d8
- **Exists:** Always

63:y	x:0
Rsvd_RawDstTran	RAW

Table 5-21 Fields for Register: RawDstTran

Bits	Name	Memory Access	Description
63:y	Rsvd_RawDstTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	RAW	R/W	Raw Status for IntDstTran Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Raw Interrupt Status ■ 0x1 (ACTIVE): Active Raw Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.5 RawErr

- **Name:** Raw Status for IntErr Interrupt
- **Description:** Interrupt events are stored in this Raw Interrupt Status register before masking. This register has a bit allocated per channel; for example, RawErr[2] is the Channel 2 raw error interrupt.

  Each bit in this register is cleared by writing a 1 to the corresponding location in the ClearErr register.

  Note: Write access is available to this register or software testing purposes only. Under normal operation, writes to this register are not recommended.
- **Size:** 64 bits
- **Offset:** 0x2e0
- **Exists:** Always

63:y	x:0
Rsvd_RawErr	RAW

Table 5-22 Fields for Register: RawErr

Bits	Name	Memory Access	Description
63:y	Rsvd_RawErr	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	RAW	R/W	Raw Status for IntErr Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Raw Interrupt Status ■ 0x1 (ACTIVE): Active Raw Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.6 StatusTfr

- **Name:** Status for IntTfr Interrupt
- **Description:** Channel DMA Transfer complete interrupt event from all channels is stored in this Interrupt Status register after masking. This register has a bit allocated per channel; for example, StatusTfr[2] is the Channel 2 source DMA transfer complete interrupt. The contents of this register are used to generate the interrupt signals (int or int_n bus, depending on interrupt polarity) leaving the DW_ahb_dmac.
- **Size:** 64 bits
- **Offset:** 0x2e8
- **Exists:** Always

63:y	x:0
Rsvd_StatusTfr	STATUS

Table 5-23 Fields for Register: StatusTfr

Bits	Name	Memory Access	Description
63:y	Rsvd_StatusTfr	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	STATUS	R	Status for IntTfr Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Interrupt Status ■ 0x1 (ACTIVE): Active Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.7 StatusBlock

- **Name:** Status for IntBlock Interrupt
- **Description:** Channel Block complete interrupt event from all channels is stored in this Interrupt Status register after masking. This register has a bit allocated per channel; for example, StatusBlock[2] is the Channel 2 block complete interrupt. The contents of this register are used to generate the interrupt signals (int or int_n bus, depending on interrupt polarity) leaving the DW_ahb_dmac.
- **Size:** 64 bits
- **Offset:** 0x2f0
- **Exists:** Always

63:y	x:0
Rsvd_StatusBlock	STATUS

Table 5-24 Fields for Register: StatusBlock

Bits	Name	Memory Access	Description
63:y	Rsvd_StatusBlock	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	STATUS	R	Status for IntBlock Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Interrupt Status ■ 0x1 (ACTIVE): Active Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.8 StatusSrcTran

- **Name:** Status for IntSrcTran Interrupt
- **Description:** Channel Source Transaction complete interrupt event from all channels is stored in this Interrupt Status register after masking. This register has a bit allocated per channel; for example, StatusSrcTran[2] is the Channel 2 source transaction complete interrupt. The contents of this register are used to generate the interrupt signals (int or int_n bus, depending on interrupt polarity) leaving the DW_ahb_dmac.
- **Size:** 64 bits
- **Offset:** 0x2f8
- **Exists:** Always

63:y	x:0
Rsvd_StatusSrcTran	STATUS

Table 5-25 Fields for Register: StatusSrcTran

Bits	Name	Memory Access	Description
63:y	Rsvd_StatusSrcTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	STATUS	R	Status for IntSrcTran Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Interrupt Status ■ 0x1 (ACTIVE): Active Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.9 StatusDstTran

- **Name:** Status for IntDstTran Interrupt
- **Description:** Channel destination transaction complete interrupt event from all channels is stored in this Interrupt Status register after masking. This register has a bit allocated per channel; for example, StatusDstTran[2] is the Channel 2 status destination transaction complete interrupt. The contents of this register are used to generate the interrupt signals (int or int_n bus, depending on interrupt polarity) leaving the DW_ahb_dmac.
- **Size:** 64 bits
- **Offset:** 0x300
- **Exists:** Always

63:y	x:0
Rsvd_StatusDstTran	STATUS

Table 5-26 Fields for Register: StatusDstTran

Bits	Name	Memory Access	Description
63:y	Rsvd_StatusDstTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	STATUS	R	Status for IntDstTran Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Interrupt Status ■ 0x1 (ACTIVE): Active Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.10 StatusErr

- **Name:** Status for IntErr Interrupt
- **Description:** Channel Error interrupt event from all channels is stored in this Interrupt Status register after masking. This register has a bit allocated per channel; for example, StatusErr[2] is the Channel 2 status Error interrupt. The contents of this register are used to generate the interrupt signals (int or int_n bus, depending on interrupt polarity) leaving the DW_ahb_dmac.
- **Size:** 64 bits
- **Offset:** 0x308
- **Exists:** Always

63:y	x:0
Rsvd_StatusErr	STATUS

Table 5-27 Fields for Register: StatusErr

Bits	Name	Memory Access	Description
63:y	Rsvd_StatusErr	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	STATUS	R	Status for IntErr Interrupt **Values:** ■ 0x0 (INACTIVE): Inactive Interrupt Status ■ 0x1 (ACTIVE): Active Interrupt Status **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.11 MaskTfr

- **Name:** Status for IntTfr Interrupt
- **Description:** The contents of the Raw Status register RawTfr is masked with the contents of the Mask register MaskTfr. Each bit of register is allocated per channel; for example, MaskTfr[2] is the mask bit for the Channel 2 transfer complete interrupt.

  A channel INT_MASK bit will be written only if the corresponding mask write enable bit in the INT_MASK_WE field is asserted on the same AHB write transfer. This allows software to set a mask bit without performing a read-modified write operation. For example, writing hex 01x1 to the MaskTfr register writes a 1 into MaskTfr[0], while MaskTfr[7:1] remains unchanged. Writing hex 00xx leaves MaskTfr[7:0] unchanged.

  Writing a 1 to any bit in this register unmasks the corresponding interrupt, thus allowing the DW_ahb_dmac to set the appropriate bit in the Status registers and int_* port signals.
- **Size:** 64 bits
- **Offset:** 0x310
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_MaskTfr	INT_MASK_WE	Rsvd_MaskTfr	INT_MASK

Table 5-28 Fields for Register: MaskTfr

Bits	Name	Memory Access	Description
63:y	Rsvd_1_MaskTfr	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-28 Fields for Register: MaskTfr (Continued)

Bits	Name	Memory Access	Description
x:8	INT_MASK_WE	W	Interrupt Mask Write Enable **Values:** ■ 0x0 (DISABLED): Interrupt mask write disable ■ 0x1 (ENABLED): Interrupt mask write enable **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_MaskTfr	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	INT_MASK	R/W	Mask for IntTfr Interrupt **Values:** ■ 0x0 (MASK): Mask the interrupts ■ 0x1 (UNMASK): Unmask the interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.12 MaskBlock

- **Name:** Mask for IntBlock Interrupt
- **Description:** The contents of the Raw Status register RawBlock is masked with the contents of the Mask register MaskBlock. Each bit of register is allocated per channel; for example, MaskBlock[2] is the mask bit for the Channel 2 block complete interrupt.

  A channel INT_MASK bit will be written only if the corresponding mask write enable bit in the INT_MASK_WE field is asserted on the same AHB write transfer. This allows software to set a mask bit without performing a read-modified write operation. For example, writing hex 01x1 to the MaskBlock register writes a 1 into MaskBlock[0], while MaskBlock[7:1] remains unchanged. Writing hex 00xx leaves MaskBlock[7:0] unchanged.

  Writing a 1 to any bit in this register unmasks the corresponding interrupt, thus allowing the DW_ahb_dmac to set the appropriate bit in the Status registers and int_* port signals.
- **Size:** 64 bits
- **Offset:** 0x318
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_MaskBlock	INT_MASK_WE	Rsvd_MaskBlock	INT_MASK

Table 5-29 Fields for Register: MaskBlock

Bits	Name	Memory Access	Description
63:y	Rsvd_1_MaskBlock	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-29 Fields for Register: MaskBlock (Continued)

Bits	Name	Memory Access	Description
x:8	INT_MASK_WE	W	Interrupt Mask Write Enable **Values:** ■ 0x0 (DISABLED): Interrupt mask write disable ■ 0x1 (ENABLED): Interrupt mask write enable **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_MaskBlock	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	INT_MASK	R/W	Mask for IntBlock Interrupt **Values:** ■ 0x0 (MASK): Mask the interrupts ■ 0x1 (UNMASK): Unmask the interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.13 MaskSrcTran

- **Name:** Status for IntSrcTran Interrupt
- **Description:** The contents of the Raw Status register RawSrcTran is masked with the contents of the Mask register MaskSrcTran. Each bit of register is allocated per channel; for example, MaskSrcTran[2] is the mask bit for the Channel 2 source transaction complete interrupt.

  When the source peripheral of DMA channel i is memory, then the source transaction complete interrupt, MaskSrcTran[i], must be masked to prevent an erroneous triggering of an interrupt on the int_combined signal.

  A channel INT_MASK bit will be written only if the corresponding mask write enable bit in the INT_MASK_WE field is asserted on the same AHB write transfer. This allows software to set a mask bit without performing a read-modified write operation. For example, writing hex 01x1 to the MaskSrcTran register writes a 1 into MaskSrcTran[0], while MaskSrcTran[7:1] remains unchanged. Writing hex 00xx leaves MaskSrcTran[7:0] unchanged.

  Writing a 1 to any bit in this register unmasks the corresponding interrupt, thus allowing the DW_ahb_dmac to set the appropriate bit in the Status registers and int_* port signals.
- **Size:** 64 bits
- **Offset:** 0x320
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_MaskSrcTran	INT_MASK_WE	Rsvd_MaskSrcTran	INT_MASK

Table 5-30 Fields for Register: MaskSrcTran

Bits	Name	Memory Access	Description
63:y	Rsvd_1_MaskSrcTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-30 Fields for Register: MaskSrcTran (Continued)

Bits	Name	Memory Access	Description
x:8	INT_MASK_WE	W	Interrupt Mask Write Enable **Values:** ■ 0x0 (DISABLED): Interrupt mask write disable ■ 0x1 (ENABLED): Interrupt mask write enable **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_MaskSrcTran	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	INT_MASK	R/W	Mask for IntSrcTran Interrupt **Values:** ■ 0x0 (MASK): Mask the interrupts ■ 0x1 (UNMASK): Unmask the interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.14 MaskDstTran

- **Name:** Mask for IntDstTran Interrupt
- **Description:** The contents of the Raw Status register RawDstTran is masked with the contents of the Mask register MaskDstTran. Each bit of register is allocated per channel; for example, MaskDstTran[2] is the mask bit for the Channel 2 destination transaction complete interrupt.

  When the destination peripheral of DMA channel i is memory, then the destination transaction complete interrupt, MaskDstTran[i], must be masked to prevent an erroneous triggering of an interrupt on the int_combined(_n) signal.

  A channel INT_MASK bit will be written only if the corresponding mask write enable bit in the INT_MASK_WE field is asserted on the same AHB write transfer. This allows software to set a mask bit without performing a read-modified write operation. For example, writing hex 01x1 to the MaskDstTran register writes a 1 into MaskDstTran[0], while MaskDstTran[7:1] remains unchanged. Writing hex 00xx leaves MaskDstTran[7:0] unchanged.

  Writing a 1 to any bit in this register unmasks the corresponding interrupt, thus allowing the DW_ahb_dmac to set the appropriate bit in the Status registers and int_* port signals.
- **Size:** 64 bits
- **Offset:** 0x328
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_MaskDstTran	INT_MASK_WE	Rsvd_MaskDstTran	INT_MASK

Table 5-31 Fields for Register: MaskDstTran

Bits	Name	Memory Access	Description
63:y	Rsvd_1_MaskDstTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-31 Fields for Register: MaskDstTran (Continued)

Bits	Name	Memory Access	Description
x:8	INT_MASK_WE	W	Interrupt Mask Write Enable **Values:** ■ 0x0 (DISABLED): Interrupt mask write disable ■ 0x1 (ENABLED): Interrupt mask write enable **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_MaskDstTran	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	INT_MASK	R/W	Mask for IntDstTran Interrupt **Values:** ■ 0x0 (MASK): Mask the interrupts ■ 0x1 (UNMASK): Unmask the interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.15 MaskErr

- **Name:** Mask for IntErr Interrupt
- **Description:** The contents of the Raw Status register RawErr is masked with the contents of the Mask register MaskErr. Each bit of register is allocated per channel; for example, MaskErr[2] is the mask bit for the Channel 2 error interrupt.

  A channel INT_MASK bit will be written only if the corresponding mask write enable bit in the INT_MASK_WE field is asserted on the same AHB write transfer. This allows software to set a mask bit without performing a read-modified write operation. For example, writing hex 01x1 to the MaskErr register writes a 1 into MaskErr[0], while MaskErr[7:1] remains unchanged. Writing hex 00xx leaves MaskErr[7:0] unchanged.

  Writing a 1 to any bit in this register unmasks the corresponding interrupt, thus allowing the DW_ahb_dmac to set the appropriate bit in the Status registers and int_* port signals.
- **Size:** 64 bits
- **Offset:** 0x330
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_MaskErr	INT_MASK_WE	Rsvd_MaskErr	INT_MASK

Table 5-32 Fields for Register: MaskErr

Bits	Name	Memory Access	Description
63:y	Rsvd_1_MaskErr	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-32 Fields for Register: MaskErr (Continued)

Bits	Name	Memory Access	Description
x:8	INT_MASK_WE	W	Interrupt Mask Write Enable **Values:** ■ 0x0 (DISABLED): Interrupt mask write disable ■ 0x1 (ENABLED): Interrupt mask write enable **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_MaskErr	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	INT_MASK	R/W	Mask for IntErr Interrupt **Values:** ■ 0x0 (MASK): Mask the interrupts ■ 0x1 (UNMASK): Unmask the interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.16 ClearTfr

- **Name:** Clear for IntTfr Interrupt
- **Description:** Each bit in the RawTfr and StatusTfr is cleared on the same cycle by writing a 1 to the corresponding location in the this registers. Each bit is allocated per channel; for example, ClearTfr[2] is the clear bit for the Channel 2 transfer done interrupt. Writing a 0 has no effect. This registers are not readable.
- **Size:** 64 bits
- **Offset:** 0x338
- **Exists:** Always

63:y	x:0
Rsvd_ClearTfr	CLEAR

Table 5-33 Fields for Register: ClearTfr

Bits	Name	Memory Access	Description
63:y	Rsvd_ClearTfr	W	Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CLEAR	W	Clear for IntTfr Interrupt **Values:** ■ 0x0 (NOT_CLEAR): No effect ■ 0x1 (CLEAR): Clears interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.17 ClearBlock

- **Name:** Clear for IntBlock Interrupt
- **Description:** Each bit in the RawBlock and StatusBlock is cleared on the same cycle by writing a 1 to the corresponding location in the this registers. Each bit is allocated per channel; for example, ClearBlock[2] is the clear bit for the Channel 2 block done interrupt. Writing a 0 has no effect. This registers are not readable.
- **Size:** 64 bits
- **Offset:** 0x340
- **Exists:** Always

63:y	x:0
Rsvd_ClearBlock	CLEAR

Table 5-34 Fields for Register: ClearBlock

Bits	Name	Memory Access	Description
63:y	Rsvd_ClearBlock	W	Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CLEAR	W	Clear for IntBlock Interrupt **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.18 ClearSrcTran

- **Name:** Clear for IntSrcTran Interrupt
- **Description:** Each bit in the RawSrcTran and StatusSrcTran is cleared on the same cycle by writing a 1 to the corresponding location in the this registers. Each bit is allocated per channel; for example, ClearSrcTran[2] is the clear bit for the Channel 2 source transaction done interrupt. Writing a 0 has no effect. This registers are not readable.
- **Size:** 64 bits
- **Offset:** 0x348
- **Exists:** Always

63:y	x:0
Rsvd_ClearSrcTran	CLEAR

Table 5-35 Fields for Register: ClearSrcTran

Bits	Name	Memory Access	Description
63:y	Rsvd_ClearSrcTran	W	Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CLEAR	W	Clear for IntSrcTran Interrupt **Values:** ■ 0x0 (NOT_CLEAR): No effect ■ 0x1 (CLEAR): Clears interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.19 ClearDstTran

- **Name:** Clear for IntDstTran Interrupt
- **Description:** Each bit in the RawDstTran and StatusDstTran is cleared on the same cycle by writing a 1 to the corresponding location in the this registers. Each bit is allocated per channel; for example, ClearDstTran[2] is the clear bit for the Channel 2 destination transaction done interrupt. Writing a 0 has no effect. This registers are not readable.
- **Size:** 64 bits
- **Offset:** 0x350
- **Exists:** Always

63:y	x:0
Rsvd_ClearDstTran	CLEAR

Table 5-36 Fields for Register: ClearDstTran

Bits	Name	Memory Access	Description
63:y	Rsvd_ClearDstTran	W	Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CLEAR	W	Clear for IntDstTran Interrupt **Values:** ■ 0x0 (NOT_CLEAR): No effect ■ 0x1 (CLEAR): Clears interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.20 ClearErr

- **Name:** Clear for IntErr Interrupt
- **Description:** Each bit in the RawErr and StatusErr is cleared on the same cycle by writing a 1 to the corresponding location in the this registers. Each bit is allocated per channel; for example, ClearErr[2] is the clear bit for the Channel 2 error interrupt. Writing a 0 has no effect. This registers are not readable.
- **Size:** 64 bits
- **Offset:** 0x358
- **Exists:** Always

63:y	x:0
Rsvd_ClearErr	CLEAR

Table 5-37 Fields for Register: ClearErr

Bits	Name	Memory Access	Description
63:y	Rsvd_ClearErr	W	Reserved field **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CLEAR	W	Clear for IntErr Interrupt **Values:** ■ 0x0 (NOT_CLEAR): No effect ■ 0x1 (CLEAR): Clears interrupts **Value After Reset:** 0x0 **Exists:** Always **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.2.21 StatusInt

- **Name:** Status for each Interrupt type
- **Description:** The contents of each of the five Status registers StatusTfr, StatusBlock, StatusSrcTran, StatusDstTran, StatusErr is ORed to produce a single bit for each interrupt type in the Combined Status register (StatusInt). This register is read-only.
- **Size:** 64 bits
- **Offset:** 0x360
- **Exists:** Always

63:5	4	3	2	1	0
Rsvd_StatusInt	ERR	DSTT	SRCT	BLOCK	TFR

Table 5-38 Fields for Register: StatusInt

Bits	Name	Memory Access	Description
63:5	Rsvd_StatusInt	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
4	ERR	R	OR of the contents of StatusErr **Values:** ■ 0x0 (INACTIVE): OR of the contents of StatusErr register is 0 ■ 0x1 (ACTIVE): OR of the contents of StatusErr register is 1 **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

Table 5-38 Fields for Register: StatusInt (Continued)

Bits	Name	Memory Access	Description
3	DSTT	R	OR of the contents of StatusDstTran **Values:** ■ 0x0 (INACTIVE): OR of the contents of StatusDstTran register is 0 ■ 0x1 (ACTIVE): OR of the contents of StatusDstTran register is 1 **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
2	SRCT	R	OR of the contents of StatusSrcTran **Values:** ■ 0x0 (INACTIVE): OR of the contents of StatusSrcTran register is 0 ■ 0x1 (ACTIVE): OR of the contents of StatusSrcTran register is 1 **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
1	BLOCK	R	OR of the contents of StatusBlock register **Values:** ■ 0x0 (INACTIVE): OR of the contents of StatusBlock register is 0 ■ 0x1 (ACTIVE): OR of the contents of StatusBlock register is 1 **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
0	TFR	R	OR of the contents of StatusTfr register **Values:** ■ 0x0 (INACTIVE): OR of the contents of StatusTfr register is 0 ■ 0x1 (ACTIVE): OR of the contents of StatusTfr register is 1 **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.3 DMAC/Software_Handshake Registers

Software Handshaking Registers. Follow the link for the register to see a detailed description of the register.

Table 5-39 Registers for Address Block: DMAC/Software_Handshake_Registers

Register	Offset	Description
ReqSrcReg on page 234	0x368	A bit is assigned for each channel in this register. ReqSrcReg[n] is ignored when software handshaking...
ReqDstReg on page 236	0x370	A bit is assigned for each channel in this register. ReqDstReg[n] is ignored when software handshaking...
SglRqSrcReg on page 238	0x378	A bit is assigned for each channel in this register. SglReqSrcReg[n] is ignored when software handshaking...
SglRqDstReg on page 240	0x380	A bit is assigned for each channel in this register. SglReqDstReg[n] is ignored when software handshaking...
LstSrcReg on page 242	0x388	A bit is assigned for each channel in this register. LstSrcReg[n] is ignored when software handshaking...
LstDstReg on page 244	0x390	A bit is assigned for each channel in this register. LstDstReg[n] is ignored when software handshaking...

5.3.1 ReqSrcReg

- **Name:** Source Software Transaction Request register
- **Description:** A bit is assigned for each channel in this register. ReqSrcReg[n] is ignored when software handshaking is not enabled for the source of channel n.

  A channel SRC_REQ bit is written only if the corresponding channel write enable bit in the SRC_REQ_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register. For example, writing hex 0101 writes a 1 into ReqSrcReg[0], while ReqSrcReg[7:1] remains unchanged. Writing hex 00xx leaves ReqSrcReg[7:0] unchanged. This allows software to set a bit in the ReqSrcReg register without performing a read-modified write operation.

  The functionality of this register depends on whether the source is a flow control peripheral or not. For a description of when the source is not a flow controller, refer to "Software Handshaking Peripheral Is Not Flow Controller". For a description of when the source is a flow controller, refer to "Software Handshaking Peripheral Is Flow Controller"
- **Size:** 64 bits
- **Offset:** 0x368
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_ReqSrcReg	SRC_REQ_WE	Rsvd_ReqSrcReg	SRC_REQ

Table 5-40 Fields for Register: ReqSrcReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_ReqSrcReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-40 Fields for Register: ReqSrcReg (Continued)

Bits	Name	Memory Access	Description
x:8	SRC_REQ_WE	W	Source Software Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Source request write Disable ■ 0x1 (ENABLED): Source request write Enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_ReqSrcReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	SRC_REQ	R/W	Source Software Transaction Request **Values:** ■ 0x0 (INACTIVE): Source request is not active ■ 0x1 (ACTIVE): Source request is active **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.3.2 ReqDstReg

- **Name:** Destination Software Transaction Request register
- **Description:** A bit is assigned for each channel in this register. ReqDstReg[n] is ignored when software handshaking is not enabled for the source of channel n.

  A channel DST_REQ bit is written only if the corresponding channel write enable bit in the DST_REQ_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register.

  The functionality of this register depends on whether the destination is a flow control peripheral or not. For a description of when the destination is not a flow controller, refer to "Software Handshaking Peripheral Is Not Flow Controller". For a description of when the destination is a flow controller, refer to "Software Handshaking Peripheral Is Flow Controller".
- **Size:** 64 bits
- **Offset:** 0x370
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_ReqDstReg	DST_REQ_WE	Rsvd_ReqDstReg	DST_REQ

Table 5-41 Fields for Register: ReqDstReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_ReqDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-41 Fields for Register: ReqDstReg (Continued)

Bits	Name	Memory Access	Description
x:8	DST_REQ_WE	W	Destination Software Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Destination request write Disable ■ 0x1 (ENABLED): Destination request write Enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_ReqDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	DST_REQ	R/W	Destination Software Transaction Request **Values:** ■ 0x0 (INACTIVE): Destination request is not active ■ 0x1 (ACTIVE): Destination request is active **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.3.3 SglRqSrcReg

- **Name:** Source Single Transaction Request register
- **Description:** A bit is assigned for each channel in this register. SglReqSrcReg[n] is ignored when software handshaking is not enabled for the source of channel n.

  A channel SRC_SGLREQ bit is written only if the corresponding channel write enable bit in the SRC_SGLREQ_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register.

  The functionality of this register depends on whether the source is a flow control peripheral or not. For a description of when the source is not a flow controller, refer to "Software Handshaking Peripheral Is Not Flow Controller". For a description of when the source is a flow controller, refer to "Software Handshaking Peripheral Is Flow Controller".
- **Size:** 64 bits
- **Offset:** 0x378
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_SglRqSrcReg	SRC_SGLREQ_WE	Rsvd_SglRqSrcReg	SRC_SGLREQ

Table 5-42 Fields for Register: SglRqSrcReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_SglRqSrcReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-42 Fields for Register: SglRqSrcReg (Continued)

Bits	Name	Memory Access	Description
x:8	SRC_SGLREQ_WE	W	Source Single Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Single write Disable ■ 0x1 (ENABLED): Single write Enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_SglRqSrcReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	SRC_SGLREQ	R/W	Source Single Transaction Request **Values:** ■ 0x0 (INACTIVE): Source request is not active ■ 0x1 (ACTIVE): Source request is active **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.3.4 SglRqDstReg

- **Name:** Destination Single Transaction Request register
- **Description:** A bit is assigned for each channel in this register. SglReqDstReg[n] is ignored when software handshaking is not enabled for the destination of channel n.

  A channel DST_SGLREQ bit is written only if the corresponding channel write enable bit in the DST_SGLREQ_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register.

  The functionality of this register depends on whether the destination is a flow control peripheral or not. For a description of when the destination is not a flow controller, refer to "Software Handshaking Peripheral Is Not Flow Controller". For a description of when the destination is a flow controller, refer to "Software Handshaking Peripheral Is Flow Controller".
- **Size:** 64 bits
- **Offset:** 0x380
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_SglRqDstReg	DST_SGLREQ_WE	Rsvd_SglRqDstReg	DST_SGLREQ

Table 5-43 Fields for Register: SglRqDstReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_SglRqDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-43 Fields for Register: SglRqDstReg (Continued)

Bits	Name	Memory Access	Description
x:8	DST_SGLREQ_WE	W	Destination Single Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Destination write Disable ■ 0x1 (ENABLED): Destination write Enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_SglRqDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	DST_SGLREQ	R/W	Destination Single Transaction Request **Values:** ■ 0x0 (INACTIVE): Destination Single or burst request is not active ■ 0x1 (ACTIVE): Destination Single or burst request is active **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.3.5 LstSrcReg

- **Name:** Source Last Transaction Request register
- **Description:** A bit is assigned for each channel in this register. LstSrcReg[n] is ignored when software handshaking is not enabled for the source of channel n, or when the source of channel n is not a flow controller.

  A channel LSTSRC bit is written only if the corresponding channel write enable bit in the LSTSRC_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register.

  For a description of this register, refer to "Software Handshaking Peripheral Is Flow Controller".
- **Size:** 64 bits
- **Offset:** 0x388
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_LstSrcReg	LSTSRC_WE	Rsvd_LstSrcReg	LSTSRC

Table 5-44 Fields for Register: LstSrcReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_LstSrcReg	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-44 Fields for Register: LstSrcReg (Continued)

Bits	Name	Memory Access	Description
x:8	LSTSRC_WE	W	Source Last Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Source last transaction request write disable ■ 0x1 (ENABLED): Source last transaction request write enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_LstSrcReg	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	LSTSRC	R/W	Source Last Transaction Request register **Values:** ■ 0x0 (NOT_LAST): Not last transaction in current block ■ 0x1 (LAST): Last transaction in current block **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.3.6 LstDstReg

- **Name:** Destination Last Transaction Request register
- **Description:** A bit is assigned for each channel in this register. LstDstReg[n] is ignored when software handshaking is not enabled for the destination of channel n or when the destination of channel n is not a flow controller.

  A channel LSTDST bit is written only if the corresponding channel write enable bit in the LSTDST_WE field is asserted on the same AHB write transfer, and if the channel is enabled in the ChEnReg register.

  For a description of this register, refer to "Software Handshaking Peripheral Is Flow Controller".
- **Size:** 64 bits
- **Offset:** 0x390
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_LstDstReg	LSTDST_WE	Rsvd_LstDstReg	LSTDST

Table 5-45 Fields for Register: LstDstReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_LstDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-45 Fields for Register: LstDstReg (Continued)

Bits	Name	Memory Access	Description
x:8	LSTDST_WE	W	Source Last Transaction Request write enable **Values:** ■ 0x0 (DISABLED): Destination last transaction request write disable ■ 0x1 (ENABLED): Destination last transaction request write enable **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_LstDstReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	LSTDST	R/W	Destination Last Transaction Request **Values:** ■ 0x0 (NOT_LAST): Not last transaction in current block ■ 0x1 (LAST): Last transaction in current block **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.4 DMAC/Miscellaneous Registers

Miscellaneous Registers. Follow the link for the register to see a detailed description of the register.

Table 5-46 Registers for Address Block: DMAC/Miscellaneous_Registers

Register	Offset	Description
DmaCfgReg on page 247	0x398	This register is used to enable the DW_ahb_dmac, which must be done before any channel activity...
ChEnReg on page 248	0x3a0	This is the DW_ahb_dmac Channel Enable Register. If software needs to set up a new channel, then...
DmaIdReg on page 250	0x3a8	This is the DW_ahb_dmac ID register, which is a read-only register that reads back the coreConsultant-configured...
DmaTestReg on page 251	0x3b0	This register is used to put the AHB slave interface into test mode, during which the readback value...
DmaLpTimeoutReg on page 252	0x3b8	This register holds the timeout value of Low Power Counter. The reset value of the register is...
DMA_COMP_PARAMS_6 on page 253	0x3c8	DMA_COMP_PARAMS_6 is a constant read-only register that contains encoded information about the component...
DMA_COMP_PARAMS_5 on page 261	0x3d0	DMA_COMP_PARAMS_5 is a constant read-only register that contains encoded information about the component...
DMA_COMP_PARAMS_4 on page 274	0x3d8	DMA_COMP_PARAMS_4 is a constant read-only register that contains encoded information about the component...
DMA_COMP_PARAMS_3 on page 287	0x3e0	DMA_COMP_PARAMS_3 is a constant read-only register that contains encoded information about the component...
DMA_COMP_PARAMS_2 on page 300	0x3e8	DMA_COMP_PARAMS_2 is a constant read-only register that contains encoded information about the component...
DMA_COMP_PARAMS_1 on page 315	0x3f0	DMA_COMP_PARAMS_1 is a constant read-only register that contains encoded information about the component...
DmaCompsID on page 329	0x3f8	This is the DW_ahb_dmac Component Version register, which is a read-only register that specifies...

5.4.1 DmaCfgReg

- **Name:** DW_ahb_dmac Configuration Register
- **Description:** This register is used to enable the DW_ahb_dmac, which must be done before any channel activity can begin.

  If the global channel enable bit is cleared while any channel is still active, then DmaCfgReg.DMA_EN still returns 1 to indicate that there are channels still active until hardware has terminated all activity on all channels, at which point the DmaCfgReg.DMA_EN bit returns 0. For more information, refer to "Abnormal Transfer Termination".
- **Size:** 64 bits
- **Offset:** 0x398
- **Exists:** Always

63:1	0
Rsvd_DmaCfgReg	DMA_EN

Table 5-47 Fields for Register: DmaCfgReg

Bits	Name	Memory Access	Description
63:1	Rsvd_DmaCfgReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true
0	DMA_EN	R/W	**DW_ahb_dmac Enable bit.** **Values:** ■ 0x0 (DISABLED): DW_ahb_dmac Disabled ■ 0x1 (ENABLED): DW_ahb_dmac Enabled **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true

5.4.2 ChEnReg

- **Name:** DW_ahb_dmac Channel Enable Register
- **Description:** This is the DW_ahb_dmac Channel Enable Register. If software needs to set up a new channel, then it can read this register in order to find out which channels are currently inactive; it can then enable an inactive channel with the required priority.

  All bits of this register are cleared to 0 when the global DW_ahb_dmac channel enable bit, DmaCfgReg[0], is 0. When the global channel enable bit is 0, then a write to the ChEnReg register is ignored and a read will always read back 0.

  The channel enable bit, ChEnReg.CH_EN, is written only if the corresponding channel write enable bit, ChEnReg.CH_EN_WE, is asserted on the same AHB write transfer. For example, writing hex 01x1 writes a 1 into ChEnReg[0], while ChEnReg[7:1] remains unchanged. Writing hex 00xx leaves ChEnReg[7:0] unchanged. Note that a read-modified write is not required.

  For information on software disabling a channel by writing 0 to ChEnReg.CH_EN, refer to "Disabling a Channel Prior to Transfer Completion".
- **Size:** 64 bits
- **Offset:** 0x3a0
- **Exists:** Always

63:y	x:8	7:y	x:0
Rsvd_1_ChEnReg	CH_EN_WE	Rsvd_ChEnReg	CH_EN

Table 5-48 Fields for Register: ChEnReg

Bits	Name	Memory Access	Description
63:y	Rsvd_1_ChEnReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS + 8

Table 5-48 Fields for Register: ChEnReg (Continued)

Bits	Name	Memory Access	Description
x:8	CH_EN_WE	W	Channel enable register **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS + 7
7:y	Rsvd_ChEnReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[y]:** DMAH_NUM_CHANNELS
x:0	CH_EN	R/W	**Channel Enable.** The ChEnReg.CH_EN bit is automatically cleared by hardware to disable the channel after the last AMBA transfer of the DMA transfer to the destination has completed. Software can therefore poll this bit to determine when this channel is free for a new DMA transfer. **Values:** ■ 0x0 (DISABLED): Disable the channel ■ 0x1 (ENABLED): Enable the channel **Value After Reset:** 0x0 **Exists:** Always **Volatile:** true **Range Variable[x]:** DMAH_NUM_CHANNELS - 1

5.4.3 DmaIdReg

- **Name:** DW_ahb_dmac ID register
- **Description:** This is the DW_ahb_dmac ID register, which is a read-only register that reads back the coreConsultant-configured hardcoded ID number, DMAH_ID_NUM.
- **Size:** 64 bits
- **Offset:** 0x3a8
- **Exists:** Always

63:32	31:0
Rsvd_DmaIdReg	DMA_ID

Table 5-49 Fields for Register: DmaIdReg

Bits	Name	Memory Access	Description
63:32	Rsvd_DmaIdReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always
31:0	DMA_ID	R	Hardcoded DW_ahb_dmac peripheral ID. **Value After Reset:** DMAH_ID_NUM **Exists:** Always

5.4.4 DmaTestReg

- **Name:** DMA Test registers
- **Description:** This register is used to put the AHB slave interface into test mode, during which the readback value of the writable registers match the value written, assuming the DW_ahb_dmac configuration has not optimized the same registers. In normal operation, the readback value of some registers is a function of the DW_ahb_dmac state and does not match the value written.
- **Size:** 64 bits
- **Offset:** 0x3b0
- **Exists:** Always

63:1	0
Rsvd_DmaTestReg	TEST_SLV_IF

Table 5-50 Fields for Register: DmaTestReg

Bits	Name	Memory Access	Description
63:1	Rsvd_DmaTestReg	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
0	TEST_SLV_IF	R/W	DMA Test register **Values:** ■ 0x0 (NORMAL_MODE): Puts the AHB slave interface into Normal mode ■ 0x1 (TEST_MODE): Puts the AHB slave interface into Test mode. In this mode, the readback value of the writable registers always matches the values written. **Value After Reset:** 0x0 **Exists:** Always

5.4.5 DmaLpTimeoutReg

- **Name:** DMAC Low Power Timeout Register
- **Description:** This register holds the timeout value of Low Power Counter. The reset value of the register is DMAH_LP_TIMEOUT_VALUE. The default reset value can be further modified if DMAH_HC_LP_TIMEOUT_VALUE = 0. The final programmed value (or the default reset value if not programmed) determines what is the timeout value the of low power counter.
- **Size:** 64 bits
- **Offset:** 0x3b8
- **Exists:** DMAH_LP_TIMEOUT_WIDTH>2

63:y	x:0
Rsvd_DmaLpTimeoutReg	DMA_LP_TIMEOUT

Table 5-51 Fields for Register: DmaLpTimeoutReg

Bits	Name	Memory Access	Description
63:y	Rsvd_DmaLpTimeoutReg	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always **Range Variable[y]:** DMAH_LP_TIMEOUT_WIDTH
x:0	DMA_LP_TIMEOUT	* Varies	This field holds timeout value of low power counter register. **Value After Reset:** DMAH_LP_TIMEOUT_VALUE **Exists:** Always **Range Variable[x]:** DMAH_LP_TIMEOUT_WIDTH - 1 **Memory Access:** {(DMAH_HC_LP_TIMEOUT_VALUE==0) ? "read-write" : "read-only"}

5.4.6 DMA_COMP_PARAMS_6

- **Name:** DW_ahb_dmac Component Parameters Register 6
- **Description:** DMA_COMP_PARAMS_6 is a constant read-only register that contains encoded information about the component parameter settings for Channel 7. The reset value depends on coreConsultant parameter(s).

  Note: If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3c8
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63	Rsvd_1_DMA_COMP_PARAMS_6
62:60	CH7_FIFO_DEPTH
59:57	CH7_SMS
56:54	CH7_LMS
53:51	CH7_DMS
50:48	CH7_MAX_MULT_SIZE
47:46	CH7_FC
45	CH7_HC_LLP
44	CH7_CTL_WB_EN
43	CH7_MULTI_BLK_EN
42	CH7_LOCK_EN
41	CH7_SRC_GAT_EN
40	CH7_DST_SCA_EN
39	CH7_STAT_SRC
38	CH7_STAT_DST
37:35	CH7_STW
34:32	CH7_DTW
31:0	Rsvd_DMA_COMP_PARAMS_6

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6

Bits	Name	Memory Access	Description
63	Rsvd_1_DMA_COMP_PARAMS_6	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
62:60	CH7_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH7_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 7 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 7 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 7 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 7 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 7 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 7 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH7_FIFO_DEPTH_RST **Exists:** Always
59:57	CH7_SMS	R	The value of this register is derived from the DMAH_CH7_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 7 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 7 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 7 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 7 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH7_SMS_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
56:54	CH7_LMS	R	The value of this register is derived from the DMAH_CH7_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 7 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 7 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 7 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 7 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH7_LMS_RST **Exists:** Always
53:51	CH7_DMS	R	The value of this register is derived from the DMAH_CH7_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 7 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 7 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 7 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 7 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH7_DMS_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
50:48	CH7_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH7_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 7 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 7 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 7 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 7 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 7 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 7 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 7 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH7_MAX_MULT_SIZE_RST **Exists:** Always
47:46	CH7_FC	R	The value of this register is derived from the DMAH_CH7_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 7 ■ 0x1 (FC_SRC): Flow controller is Source for channel 7 ■ 0x2 (FC_DST): Flow controller is Destination for channel 7 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 7 **Value After Reset:** DMAH_CH7_FC_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
45	CH7_HC_LLP	R	The value of this register is derived from the DMAH_CH7_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 7 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 7 LLP register to 0 **Value After Reset:** DMAH_CH7_HC_LLP_RST **Exists:** Always
44	CH7_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH7_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 7 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 7 **Value After Reset:** DMAH_CH7_CTL_WB_EN_RST **Exists:** Always
43	CH7_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH7_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 7 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 7 **Value After Reset:** DMAH_CH7_MULTI_BLK_EN_RST **Exists:** Always
42	CH7_LOCK_EN	R	The value of this register is derived from the DMAH_CH7_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 7 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 7 **Value After Reset:** DMAH_CH7_LOCK_EN_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
41	CH7_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH7_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 7 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 7 **Value After Reset:** DMAH_CH7_SRC_GAT_EN_RST **Exists:** Always
40	CH7_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH7_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 7 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 7 **Value After Reset:** DMAH_CH7_DST_SCA_EN_RST **Exists:** Always
39	CH7_STAT_SRC	R	The value of this register is derived from the DMAH_CH7_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 7 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 7 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH7_STAT_SRC_RST **Exists:** Always
38	CH7_STAT_DST	R	The value of this register is derived from the DMAH_CH7_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 7 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 7 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH7_STAT_DST_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
37:35	CH7_STW	R	The value of this register is derived from the DMAH_CH7_STW coreConsultant parameter. **Values:** ■ 0x0 (NO_HARDCODE): No hardcode ■ 0x1 (STW_8): Hardcode the channel 7's source transfer width to 8 bits ■ 0x2 (STW_16): Hardcode the channel 7's source transfer width to 16 bits ■ 0x3 (STW_32): Hardcode the channel 7's source transfer width to 32 bits ■ 0x4 (STW_64): Hardcode the channel 7's source transfer width to 64 bits ■ 0x5 (STW_128): Hardcode the channel 7's source transfer width to 128 bits ■ 0x6 (STW_256): Hardcode the channel 7's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH7_SRC_TR_RST **Exists:** Always
34:32	CH7_DTW	R	The value of this register is derived from the DMAH_CH7_DTW coreConsultant parameter. **Values:** ■ 0x0 (NO_HARDCODE): NO Hardcode ■ 0x1 (DTW_8): Hardcode the channel 7's destination transfer width to 8 bits ■ 0x2 (DTW_16): Hardcode the channel 7's destination transfer width to 16 bits ■ 0x3 (DTW_32): Hardcode the channel 7's destination transfer width to 32 bits ■ 0x4 (DTW_64): Hardcode the channel 7's destination transfer width to 64 bits ■ 0x5 (DTW_128): Hardcode the channel 7's destination transfer width to 128 bits ■ 0x6 (DTW_256): Hardcode the channel 7's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH7_DST_TR_RST **Exists:** Always

Table 5-52 Fields for Register: DMA_COMP_PARAMS_6 (Continued)

Bits	Name	Memory Access	Description
31:0	Rsvd_DMA_COMP_PARAMS_6	R	Reserved field - read-only **Value After Reset:** 0x0 **Exists:** Always

5.4.7 DMA_COMP_PARAMS_5

- **Name:** DW_ahb_dmac Component Parameters Register 5
- **Description:** DMA_COMP_PARAMS_5 is a constant read-only register that contains encoded information about the component parameter settings for Channel 5 and Channel 6. The reset value depends on coreConsultant parameter(s).

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3d0
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63	Rsvd_1_DMA_COMP_PARAMS_5
62:60	CH5_FIFO_DEPTH
59:57	CH5_SMS
56:54	CH5_LMS
53:51	CH5_DMS
50:48	CH5_MAX_MULT_SIZE
47:46	CH5_FC
45	CH5_HC_LLP
44	CH5_CTL_WB_EN
43	CH5_MULTI_BLK_EN
42	CH5_LOCK_EN
41	CH5_SRC_GAT_EN
40	CH5_DST_SCA_EN
39	CH5_STAT_SRC
38	CH5_STAT_DST
37:35	CH5_STW
34:32	CH5_DTW
31	Rsvd_DMA_COMP_PARAMS_5
30:28	CH6_FIFO_DEPTH
27:25	CH6_SMS
24:22	CH6_LMS
21:19	CH6_DMS
18:16	CH6_MAX_MULT_SIZE
15:14	CH6_FC
13	CH6_HC_LLP
12	CH6_CTL_WB_EN
11	CH6_MULTI_BLK_EN
10	CH6_LOCK_EN
9	CH6_SRC_GAT_EN
8	CH6_DST_SCA_EN
7	CH6_STAT_SRC
6	CH6_STAT_DST
5:3	CH6_STW
2:0	CH6_DTW

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5

Bits	Name	Memory Access	Description
63	Rsvd_1_DMA_COMP_PARAMS_5	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
62:60	CH5_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH5_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 5 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 5 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 5 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 5 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 5 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 5 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH5_FIFO_DEPTH_RST **Exists:** Always
59:57	CH5_SMS	R	The value of this register is derived from the DMAH_CH5_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 5 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 5 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 5 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 5 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH5_SMS_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
56:54	CH5_LMS	R	The value of this register is derived from the DMAH_CH5_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 5 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 5 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 5 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 5 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH5_LMS_RST **Exists:** Always
53:51	CH5_DMS	R	The value of this register is derived from the DMAH_CH5_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 5 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 5 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 5 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 5 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH5_DMS_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
50:48	CH5_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH5_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 5 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 5 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 5 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 5 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 5 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 5 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 5 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH5_MAX_MULT_SIZE_RST **Exists:** Always
47:46	CH5_FC	R	The value of this register is derived from the DMAH_CH5_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 5 ■ 0x1 (FC_SRC): Flow controller is Source for channel 5 ■ 0x2 (FC_DST): Flow controller is Destination for channel 5 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 5 **Value After Reset:** DMAH_CH5_FC_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
45	CH5_HC_LLP	R	The value of this register is derived from the DMAH_CH5_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 5 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 5 LLP register to 0 **Value After Reset:** DMAH_CH5_HC_LLP_RST **Exists:** Always
44	CH5_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH5_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 5 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 5 **Value After Reset:** DMAH_CH5_CTL_WB_EN_RST **Exists:** Always
43	CH5_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH5_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 5 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 5 **Value After Reset:** DMAH_CH5_MULTI_BLK_EN_RST **Exists:** Always
42	CH5_LOCK_EN	R	The value of this register is derived from the DMAH_CH5_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 5 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 5 **Value After Reset:** DMAH_CH5_LOCK_EN_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
41	CH5_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH5_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 5 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 5 **Value After Reset:** DMAH_CH5_SRC_GAT_EN_RST **Exists:** Always
40	CH5_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH5_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 5 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 5 **Value After Reset:** DMAH_CH5_DST_SCA_EN_RST **Exists:** Always
39	CH5_STAT_SRC	R	The value of this register is derived from the DMAH_CH5_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 5 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 5 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH5_STAT_SRC_RST **Exists:** Always
38	CH5_STAT_DST	R	The value of this register is derived from the DMAH_CH5_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 5 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 5 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH5_STAT_DST_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
37:35	CH5_STW	R	The value of this register is derived from the DMAH_CH5_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 5's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 5's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 5's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 5's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 5's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 5's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH5_SRC_TR_RST **Exists:** Always
34:32	CH5_DTW	R	The value of this register is derived from the DMAH_CH5_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 5's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 5's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 5's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 5's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 5's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 5's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH5_DST_TR_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
31	Rsvd_DMA_COMP_PARAMS_5	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
30:28	CH6_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH6_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (IFO_DEPTH_8): Channel 6 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 6 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 6 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 6 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 6 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 6 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH6_FIFO_DEPTH_RST **Exists:** Always
27:25	CH6_SMS	R	The value of this register is derived from the DMAH_CH6_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 6 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 6 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 6 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 6 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH6_SMS_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
24:22	CH6_LMS	R	The value of this register is derived from the DMAH_CH6_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 6 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 6 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 6 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 6 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH6_LMS_RST **Exists:** Always
21:19	CH6_DMS	R	The value of this register is derived from the DMAH_CH6_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 6 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 6 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 6 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 6 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH6_DMS_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
18:16	CH6_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH6_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 6 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 6 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 6 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 6 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 6 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 6 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 6 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH6_MAX_MULT_SIZE_RST **Exists:** Always
15:14	CH6_FC	R	The value of this register is derived from the DMAH_CH6_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 6 ■ 0x1 (FC_SRC): Flow controller is Source for channel 6 ■ 0x2 (FC_DST): Flow controller is Destination for channel 6 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 6 **Value After Reset:** DMAH_CH6_FC_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
13	CH6_HC_LLP	R	The value of this register is derived from the DMAH_CH6_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 6 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 6 LLP register to 0 **Value After Reset:** DMAH_CH6_HC_LLP_RST **Exists:** Always
12	CH6_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH6_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 6 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 6 **Value After Reset:** DMAH_CH6_CTL_WB_EN_RST **Exists:** Always
11	CH6_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH6_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 6 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 6 **Value After Reset:** DMAH_CH6_MULTI_BLK_EN_RST **Exists:** Always
10	CH6_LOCK_EN	R	The value of this register is derived from the DMAH_CH6_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 6 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 6 **Value After Reset:** DMAH_CH6_LOCK_EN_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
9	CH6_SRC_GAT_EN	R	The value of this register is derived from the CH6_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 6 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 6 **Value After Reset:** DMAH_CH6_SRC_GAT_EN_RST **Exists:** Always
8	CH6_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH6_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 6 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 6 **Value After Reset:** DMAH_CH6_DST_SCA_EN_RST **Exists:** Always
7	CH6_STAT_SRC	R	The value of this register is derived from the DMAH_CH6_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 6 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 6 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH6_STAT_SRC_RST **Exists:** Always
6	CH6_STAT_DST	R	The value of this register is derived from the DMAH_CH6_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 6 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 6 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH6_STAT_DST_RST **Exists:** Always

Table 5-53 Fields for Register: DMA_COMP_PARAMS_5 (Continued)

Bits	Name	Memory Access	Description
5:3	CH6_STW	R	The value of this register is derived from the DMAH_CH6_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): No hardcode ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 6's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 6's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 6's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 6's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 6's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 6's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH6_SRC_TR_RST **Exists:** Always
2:0	CH6_DTW	R	The value of this register is derived from the DMAH_CH6_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 6's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 6's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 6's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 6's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 6's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 6's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH6_DST_TR_RST **Exists:** Always

5.4.8 DMA_COMP_PARAMS_4

- **Name:** DW_ahb_dmac Component Parameters Register 4
- **Description:** DMA_COMP_PARAMS_4 is a constant read-only register that contains encoded information about the component parameter settings for Channel 3 and Channel 4. The reset value depends on coreConsultant parameter(s).

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3d8
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63	Rsvd_1_DMA_COMP_PARAMS_4
62:60	CH3_FIFO_DEPTH
59:57	CH3_SMS
56:54	CH3_LMS
53:51	CH3_DMS
50:48	CH3_MAX_MULT_SIZE
47:46	CH3_FC
45	CH3_HC_LLP
44	CH3_CTL_WB_EN
43	CH3_MULTI_BLK_EN
42	CH3_LOCK_EN
41	CH3_SRC_GAT_EN
40	CH3_DST_SCA_EN
39	CH3_STAT_SRC
38	CH3_STAT_DST
37:35	CH3_STW
34:32	CH3_DTW
31	Rsvd_DMA_COMP_PARAMS_4
30:28	CH4_FIFO_DEPTH
27:25	CH4_SMS
24:22	CH4_LMS
21:19	CH4_DMS
18:16	CH4_MAX_MULT_SIZE
15:14	CH4_FC
13	CH4_HC_LLP
12	CH4_CTL_WB_EN
11	CH4_MULTI_BLK_EN
10	CH4_LOCK_EN
9	CH4_SRC_GAT_EN
8	CH4_DST_SCA_EN
7	CH4_STAT_SRC
6	CH4_STAT_DST
5:3	CH4_STW
2:0	CH4_DTW

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4

Bits	Name	Memory Access	Description
63	Rsvd_1_DMA_COMP_PARAMS_4	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
62:60	CH3_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH3_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 3 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 3 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 3 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 3 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 3 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 3 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH3_FIFO_DEPTH_RST **Exists:** Always
59:57	CH3_SMS	R	The value of this register is derived from the DMAH_CH3_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 3 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 3 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 3 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 3 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH3_SMS_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
56:54	CH3_LMS	R	The value of this register is derived from the DMAH_CH3_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 3 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 3 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 3 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 3 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH3_LMS_RST **Exists:** Always
53:51	CH3_DMS	R	The value of this register is derived from the DMAH_CH3_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 3 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 3 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 3 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 3 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH3_DMS_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
50:48	CH3_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH3_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 3 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 3 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 3 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 3 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 3 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 3 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 3 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH3_MAX_MULT_SIZE_RST **Exists:** Always
47:46	CH3_FC	R	The value of this register is derived from the DMAH_CH3_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 3 ■ 0x1 (FC_SRC): Flow controller is Source for channel 3 ■ 0x2 (FC_DST): Flow controller is Destination for channel 3 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 3 **Value After Reset:** DMAH_CH3_FC_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
45	CH3_HC_LLP	R	The value of this register is derived from the DMAH_CH3_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 3 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 3 LLP register to 0 **Value After Reset:** DMAH_CH3_HC_LLP_RST **Exists:** Always
44	CH3_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH3_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 3 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 3 **Value After Reset:** DMAH_CH3_CTL_WB_EN_RST **Exists:** Always
43	CH3_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH3_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 3 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 3 **Value After Reset:** DMAH_CH3_MULTI_BLK_EN_RST **Exists:** Always
42	CH3_LOCK_EN	R	The value of this register is derived from the DMAH_CH3_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 3 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 3 **Value After Reset:** DMAH_CH3_LOCK_EN_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
41	CH3_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH3_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 3 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 3 **Value After Reset:** DMAH_CH3_SRC_GAT_EN_RST **Exists:** Always
40	CH3_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH3_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 3 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 3 **Value After Reset:** DMAH_CH3_DST_SCA_EN_RST **Exists:** Always
39	CH3_STAT_SRC	R	The value of this register is derived from the DMAH_CH3_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 3 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 3 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH3_STAT_SRC_RST **Exists:** Always
38	CH3_STAT_DST	R	The value of this register is derived from the DMAH_CH3_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 3 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 3 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH3_STAT_DST_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
37:35	CH3_STW	R	The value of this register is derived from the DMAH_CH3_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 3's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 3's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 3's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 3's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 3's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 3's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH3_SRC_TR_RST **Exists:** Always
34:32	CH3_DTW	R	The value of this register is derived from the DMAH_CH3_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 3's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 3's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 3's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 3's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 3's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 3's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH3_DST_TR_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
31	Rsvd_DMA_COMP_PARAMS_4	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
30:28	CH4_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH4_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 4 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 4 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 4 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 4 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 4 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 4 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH4_FIFO_DEPTH_RST **Exists:** Always
27:25	CH4_SMS	R	The value of this register is derived from the DMAH_CH4_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 4 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 4 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 4 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 4 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH4_SMS_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
24:22	CH4_LMS	R	The value of this register is derived from the DMAH_CH4_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 4 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 4 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 4 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 4 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH4_LMS_RST **Exists:** Always
21:19	CH4_DMS	R	The value of this register is derived from the DMAH_CH4_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 4 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 4 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 4 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 4 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH4_DMS_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
18:16	CH4_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH4_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 4 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 4 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 4 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 4 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 4 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 4 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 4 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH4_MAX_MULT_SIZE_RST **Exists:** Always
15:14	CH4_FC	R	The value of this register is derived from the DMAH_CH4_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 4 ■ 0x1 (FC_SRC): Flow controller is Source for channel 4 ■ 0x2 (FC_DST): Flow controller is Destination for channel 4 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 4 **Value After Reset:** DMAH_CH4_FC_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
13	CH4_HC_LLP	R	The value of this register is derived from the DMAH_CH4_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 4 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 4 LLP register to 0 **Value After Reset:** DMAH_CH4_HC_LLP_RST **Exists:** Always
12	CH4_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH4_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 4 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 4 **Value After Reset:** DMAH_CH4_CTL_WB_EN_RST **Exists:** Always
11	CH4_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH4_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 4 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 4 **Value After Reset:** DMAH_CH4_MULTI_BLK_EN_RST **Exists:** Always
10	CH4_LOCK_EN	R	The value of this register is derived from the DMAH_CH4_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 4 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 4 **Value After Reset:** DMAH_CH4_LOCK_EN_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
9	CH4_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH4_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 4 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 4 **Value After Reset:** DMAH_CH4_SRC_GAT_EN_RST **Exists:** Always
8	CH4_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH4_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 4 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 4 **Value After Reset:** DMAH_CH4_DST_SCA_EN_RST **Exists:** Always
7	CH4_STAT_SRC	R	The value of this register is derived from the DMAH_CH4_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 4 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 4 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH4_STAT_SRC_RST **Exists:** Always
6	CH4_STAT_DST	R	The value of this register is derived from the DMAH_CH4_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 4 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 4 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH4_STAT_DST_RST **Exists:** Always

Table 5-54 Fields for Register: DMA_COMP_PARAMS_4 (Continued)

Bits	Name	Memory Access	Description
5:3	CH4_STW	R	The value of this register is derived from the DMAH_CH4_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 4's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 4's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 4's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 4's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 4's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 4's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH4_SRC_TR_RST **Exists:** Always
2:0	CH4_DTW	R	The value of this register is derived from the DMAH_CH4_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode the channel 4's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode the channel 4's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode the channel 4's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode the channel 4's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode the channel 4's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode the channel 4's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH4_DST_TR_RST **Exists:** Always

5.4.9 DMA_COMP_PARAMS_3

- **Name:** DW_ahb_dmac Component Parameters Register 3
- **Description:** DMA_COMP_PARAMS_3 is a constant read-only register that contains encoded information about the component parameter settings for Channel 1 and Channel 2. The reset value depends on coreConsultant parameter(s).

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3e0
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63	Rsvd_1_DMA_COMP_PARAMS_3
62:60	CH1_FIFO_DEPTH
59:57	CH1_SMS
56:54	CH1_LMS
53:51	CH1_DMS
50:48	CH1_MAX_MULT_SIZE
47:46	CH1_FC
45	CH1_HC_LLP
44	CH1_CTL_WB_EN
43	CH1_MULTI_BLK_EN
42	CH1_LOCK_EN
41	CH1_SRC_GAT_EN
40	CH1_DST_SCA_EN
39	CH1_STAT_SRC
38	CH1_STAT_DST
37:35	CH1_STW
34:32	CH1_DTW
31	Rsvd_DMA_COMP_PARAMS_3
30:28	CH2_FIFO_DEPTH
27:25	CH2_SMS
24:22	CH2_LMS
21:19	CH2_DMS
18:16	CH2_MAX_MULT_SIZE
15:14	CH2_FC
13	CH2_HC_LLP
12	CH2_CTL_WB_EN
11	CH2_MULTI_BLK_EN
10	CH2_LOCK_EN
9	CH2_SRC_GAT_EN
8	CH2_DST_SCA_EN
7	CH2_STAT_SRC
6	CH2_STAT_DST
5:3	CH2_STW
2:0	CH2_DTW

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3

Bits	Name	Memory Access	Description
63	Rsvd_1_DMA_COMP_PARAMS_3	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
62:60	CH1_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH1_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 1 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 1 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 1 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 1 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 1 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 1 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH1_FIFO_DEPTH_RST **Exists:** Always
59:57	CH1_SMS	R	The value of this register is derived from the DMAH_CH1_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 1 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 1 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 1 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 1 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH1_SMS_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
56:54	CH1_LMS	R	The value of this register is derived from the DMAH_CH1_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 1 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 1 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 1 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 1 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH1_LMS_RST **Exists:** Always
53:51	CH1_DMS	R	The value of this register is derived from the DMAH_CH1_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 1 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 1 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 1 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 1 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH1_DMS_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
50:48	CH1_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH1_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 1 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 1 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 1 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 1 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 1 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 1 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 1 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH1_MAX_MULT_SIZE_RST **Exists:** Always
47:46	CH1_FC	R	The value of this register is derived from the DMAH_CH1_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 1 ■ 0x1 (FC_SRC): Flow controller is Source for channel 1 ■ 0x2 (FC_DST): Flow controller is Destination for channel 1 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 1 **Value After Reset:** DMAH_CH1_FC_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
45	CH1_HC_LLP	R	The value of this register is derived from the DMAH_CH1_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 1 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 1 LLP register to 0 **Value After Reset:** DMAH_CH1_HC_LLP_RST **Exists:** Always
44	CH1_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH1_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 1 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 1 **Value After Reset:** DMAH_CH1_CTL_WB_EN_RST **Exists:** Always
43	CH1_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH1_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 1 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 1 **Value After Reset:** DMAH_CH1_MULTI_BLK_EN_RST **Exists:** Always
42	CH1_LOCK_EN	R	The value of this register is derived from the DMAH_CH1_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 1 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 1 **Value After Reset:** DMAH_CH1_LOCK_EN_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
41	CH1_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH1_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 1 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 1 **Value After Reset:** DMAH_CH1_SRC_GAT_EN_RST **Exists:** Always
40	CH1_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH1_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 1 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 1 **Value After Reset:** DMAH_CH1_DST_SCA_EN_RST **Exists:** Always
39	CH1_STAT_SRC	R	The value of this register is derived from the DMAH_CH1_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 1 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 1 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH1_STAT_SRC_RST **Exists:** Always
38	CH1_STAT_DST	R	The value of this register is derived from the DMAH_CH1_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 1 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 1 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH1_STAT_DST_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
37:35	CH1_STW	R	The value of this register is derived from the DMAH_CH1_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 1's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 1's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 1's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 1's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 1's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 1's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH1_SRC_TR_RST **Exists:** Always
34:32	CH1_DTW	R	The value of this register is derived from the DMAH_CH1_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 1's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 1's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 1's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 1's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 1's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 1's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH1_DST_TR_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
31	Rsvd_DMA_COMP_PARAMS_3	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
30:28	CH2_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH2_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 2 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 2 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 2 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 2 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 2 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 2 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH2_FIFO_DEPTH_RST **Exists:** Always
27:25	CH2_SMS	R	The value of this register is derived from the DMAH_CH2_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 2 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 2 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 2 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 2 ■ 0x4 (PROGRAMMALE): Programmable **Value After Reset:** DMAH_CH2_SMS_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
24:22	CH2_LMS	R	The value of this register is derived from the DMAH_CH2_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 2 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 2 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 2 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 2 ■ 0x4 (PROGRAMMALE): Programmable **Value After Reset:** DMAH_CH2_LMS_RST **Exists:** Always
21:19	CH2_DMS	R	The value of this register is derived from the DMAH_CH2_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 2 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 2 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 2 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 2 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH2_DMS_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
18:16	CH2_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH2_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 2 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 2 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 2 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 2 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 2 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 2 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 2 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH2_MAX_MULT_SIZE_RST **Exists:** Always
15:14	CH2_FC	R	The value of this register is derived from the DMAH_CH2_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 2 ■ 0x1 (FC_SRC): Flow controller is Source for channel 2 ■ 0x2 (FC_DST): Flow controller is Destination for channel 2 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 2 **Value After Reset:** DMAH_CH2_FC_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
13	CH2_HC_LLP	R	The value of this register is derived from the DMAH_CH2_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 2 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 2 LLP register to 0 **Value After Reset:** DMAH_CH2_HC_LLP_RST **Exists:** Always
12	CH2_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH2_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 2 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 2 **Value After Reset:** DMAH_CH2_CTL_WB_EN_RST **Exists:** Always
11	CH2_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH2_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 2 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 2 **Value After Reset:** DMAH_CH2_MULTI_BLK_EN_RST **Exists:** Always
10	CH2_LOCK_EN	R	The value of this register is derived from the DMAH_CH2_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 2 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 2 **Value After Reset:** DMAH_CH2_LOCK_EN_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
9	CH2_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH2_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 2 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 2 **Value After Reset:** DMAH_CH2_SRC_GAT_EN_RST **Exists:** Always
8	CH2_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH2_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 2 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 2 **Value After Reset:** DMAH_CH2_DST_SCA_EN_RST **Exists:** Always
7	CH2_STAT_SRC	R	The value of this register is derived from the DMAH_CH2_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 2 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 2 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH2_STAT_SRC_RST **Exists:** Always
6	CH2_STAT_DST	R	The value of this register is derived from the DMAH_CH2_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 2 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 2 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH2_STAT_DST_RST **Exists:** Always

Table 5-55 Fields for Register: DMA_COMP_PARAMS_3 (Continued)

Bits	Name	Memory Access	Description
5:3	CH2_STW	R	The value of this register is derived from the DMAH_CH2_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 2's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 2's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 2's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 2's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 2's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 2's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH2_SRC_TR_RST **Exists:** Always
2:0	CH2_DTW	R	The value of this register is derived from the DMAH_CH2_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 2's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 2's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 2's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 2's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 2's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 2's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH2_DST_TR_RST **Exists:** Always

5.4.10 DMA_COMP_PARAMS_2

- **Name:** DW_ahb_dmac Component Parameters Register 2
- **Description:** DMA_COMP_PARAMS_2 is a constant read-only register that contains encoded information about the component parameter settings. The reset value depends on coreConsultant parameter(s).

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3e8
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63:60	CH7_MULTI_BLK_TYPE
59:56	CH6_MULTI_BLK_TYPE
55:52	CH5_MULTI_BLK_TYPE
51:48	CH4_MULTI_BLK_TYPE
47:44	CH3_MULTI_BLK_TYPE
43:40	CH2_MULTI_BLK_TYPE
39:36	CH1_MULTI_BLK_TYPE
35:32	CH0_MULTI_BLK_TYPE
31	Rsvd_DMA_COMP_PARAMS_2
30:28	CH0_FIFO_DEPTH
27:25	CH0_SMS
24:22	CH0_LMS
21:19	CH0_DMS
18:16	CH0_MAX_MULT_SIZE
15:14	CH0_FC
13	CH0_HC_LLP
12	CH0_CTL_WB_EN
11	CH0_MULTI_BLK_EN
10	CH0_LOCK_EN
9	CH0_SRC_GAT_EN
8	CH0_DST_SCA_EN
7	CH0_STAT_SRC
6	CH0_STAT_DST
5:3	CH0_STW
2:0	CH0_DTW

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2

Bits	Name	Memory Access	Description
63:60	CH7_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH7_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR7 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR7 and CTL7 are reloaded from their initial values; DAR7 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR7, DAR7,and CTL7 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR7 is contiguous;DAR7, CTL7, and LLP7 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR7 is reloaded from its initial value; DAR7, CTL7, and LLP7,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR7, CTL7, and LLP7 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR7, CTL7, and LLP7,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR7, DAR7, CTL7, and LLP7 are loaded from the next linked list item. **Value After Reset:** DMAH_CH7_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
59:56	CH6_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH6_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR6 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR6 and CTL6 are reloaded from their initial values; DAR6 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR6, DAR6,and CTL6 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR6 is contiguous;DAR6, CTL6, and LLP6 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR6 is reloaded from its initial value; DAR6, CTL6, and LLP6,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR6, CTL6, and LLP6 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR6, CTL6, and LLP6,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR6, DAR6, CTL6, and LLP6 are loaded from the next linked list item. **Value After Reset:** DMAH_CH6_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
55:52	CH5_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH5_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR5 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR5 and CTL5 are reloaded from their initial values; DAR5 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR5, DAR5,and CTL5 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR5 is contiguous;DAR5, CTL5, and LLP5 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR5 is reloaded from its initial value; DAR5, CTL5, and LLP5,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR5, CTL5, and LLP5 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR5, CTL5, and LLP5,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR5, DAR5, CTL5, and LLP5 are loaded from the next linked list item. **Value After Reset:** DMAH_CH5_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
51:48	CH4_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH4_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR4 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR4 and CTL4 are reloaded from their initial values; DAR4 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR4, DAR4,and CTL4 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR4 is contiguous;DAR4, CTL4, and LLP4 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR4 is reloaded from its initial value; DAR4, CTL4, and LLP4,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR4, CTL4, and LLP4 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR4, CTL4, and LLP4,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR4, DAR4, CTL4, and LLP4 are loaded from the next linked list item. **Value After Reset:** DMAH_CH4_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
47:44	CH3_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH3_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR3 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR3 and CTL3 are reloaded from their initial values; DAR3 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR3, DAR3,and CTL3 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR3 is contiguous;DAR3, CTL3, and LLP3 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR3 is reloaded from its initial value; DAR3, CTL3, and LLP3,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR3, CTL3, and LLP3 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR3, CTL3, and LLP3,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR3, DAR3, CTL3, and LLP3 are loaded from the next linked list item. **Value After Reset:** DMAH_CH3_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
43:40	CH2_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH2_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR2 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR2 and CTL2 are reloaded from their initial values; DAR2 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR2, DAR2,and CTL2 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR2 is contiguous;DAR2, CTL2, and LLP2 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR2 is reloaded from its initial value; DAR2, CTL2, and LLP2,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR2, CTL2, and LLP2 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR2, CTL2, and LLP2,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR2, DAR2, CTL2, and LLP2 are loaded from the next linked list item. **Value After Reset:** DMAH_CH2_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
39:36	CH1_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH1_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR1 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR1 and CTL1 are reloaded from their initial values; DAR1 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR1, DAR1,and CTL1 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR1 is contiguous;DAR1, CTL1, and LLP1 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR1 is reloaded from its initial value; DAR1, CTL1, and LLP1,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR1, CTL1, and LLP1 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR1, CTL1, and LLP1,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR1, DAR1, CTL1, and LLP1 are loaded from the next linked list item. **Value After Reset:** DMAH_CH1_MULTI_BLK_TYPE_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
35:32	CHO_MULTI_BLK_TYPE	R	The values of these bit fields are derived from the DMAH_CH0_MULTI_BLK_TYPE coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Allow all types of multi-support ■ 0x1 (CONT_RELOAD): Allow only multi-block transfers where SAR0 is contiguous; DAR and CTL are reloaded from their initial values ■ 0x2 (RELOAD_CONT): Allow only multi-block transfers where SAR0 and CTL0 are reloaded from their initial values; DAR0 is contiguous ■ 0x3 (RELOAD_RELOAD): Allow only multi-block transfers where SAR0, DAR0,and CTL0 are reloaded from their initial values ■ 0x4 (CONT_LLP): Allow only multi-block transfers where SAR0 is contiguous;DAR0, CTL0, and LLP0 are loaded from the next linked list item ■ 0x5 (RELOAD_LLP): Allow only multi-block transfers where SAR0 is reloaded from its initial value; DAR0, CTL0, and LLP0,are loaded from the next linked list item ■ 0x6 (CNT_LLP): Allow only multi-block transfers where SAR0, CTL0, and LLP0 are loaded from the next linked list item; DARx is contiguous ■ 0x7 (LLP_RELOAD): Allow only multi-block transfers where SAR0, CTL0, and LLP0,are loaded from the next linked list item; DARx is reloaded from its initial values. ■ 0x8 (LLP_LLP): Allow only multi-block transfers where SAR0, DAR0, CTL0, and LLP0 are loaded from the next linked list item. **Value After Reset:** DMAH_CH0_MULTI_BLK_TYPE_RST **Exists:** Always
31	Rsvd_DMA_COMP_PARAMS_2	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
30:28	CH0_FIFO_DEPTH	R	The value of this register is derived from the DMAH_CH0_FIFO_DEPTH coreConsultant parameter. **Values:** ■ 0x0 (FIFO_DEPTH_8): Channel 0 FIFO depth is 8 bytes ■ 0x1 (FIFO_DEPTH_16): Channel 0 FIFO depth is 16 bytes ■ 0x2 (FIFO_DEPTH_32): Channel 0 FIFO depth is 32 bytes ■ 0x3 (FIFO_DEPTH_64): Channel 0 FIFO depth is 64 bytes ■ 0x4 (FIFO_DEPTH_128): Channel 0 FIFO depth is 128 bytes ■ 0x5 (FIFO_DEPTH_256): Channel 0 FIFO depth is 256 bytes **Value After Reset:** DMAH_CH0_FIFO_DEPTH_RST **Exists:** Always
27:25	CH0_SMS	R	The value of this register is derived from the DMAH_CH0_SMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the source of channel 0 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the source of channel 0 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the source of channel 0 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the source of channel 0 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH0_SMS_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
24:22	CH0_LMS	R	The value of this register is derived from the DMAH_CH0_LMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the LLP peripherals of channel 0 ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the LLP peripherals of channel 0 ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the LLP peripherals of channel 0 ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the LLP peripherals of channel 0 ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH0_LMS_RST **Exists:** Always
21:19	CH0_DMS	R	The value of this register is derived from the DMAH_CH0_DMS coreConsultant parameter. **Values:** ■ 0x0 (MASTER_1): Hardcode the AHB master 1 interface attached to the channel 0 destination ■ 0x1 (MASTER_2): Hardcode the AHB master 2 interface attached to the channel 0 destination ■ 0x2 (MASTER_3): Hardcode the AHB master 3 interface attached to the channel 0 destination ■ 0x3 (MASTER_4): Hardcode the AHB master 4 interface attached to the channel 0 destination ■ 0x4 (PROGRAMMABLE): Programmable **Value After Reset:** DMAH_CH0_DMS_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
18:16	CH0_MAX_MULT_SIZE	R	The value of this register is derived from the DMAH_CH0_MULT_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_MULT_SIZE_4): Maximum value of burst transaction size that can be programmed for channel 0 is 4 ■ 0x1 (MAX_MULT_SIZE_8): Maximum value of burst transaction size that can be programmed for channel 0 is 8 ■ 0x2 (MAX_MULT_SIZE_16): Maximum value of burst transaction size that can be programmed for channel 0 is 16 ■ 0x3 (MAX_MULT_SIZE_32): Maximum value of burst transaction size that can be programmed for channel 0 is 32 ■ 0x4 (MAX_MULT_SIZE_64): Maximum value of burst transaction size that can be programmed for channel 0 is 64 ■ 0x5 (MAX_MULT_SIZE_128): Maximum value of burst transaction size that can be programmed for channel 0 is 128 ■ 0x6 (MAX_MULT_SIZE_256): Maximum value of burst transaction size that can be programmed for channel 0 is 256 ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH0_MAX_MULT_SIZE_RST **Exists:** Always
15:14	CH0_FC	R	The value of this register is derived from the DMAH_CH0_FC coreConsultant parameter. **Values:** ■ 0x0 (FC_DMA): Flow controller is DMA for channel 0 ■ 0x1 (FC_SRC): Flow controller is Source for channel 0 ■ 0x2 (FC_DST): Flow controller is Destination for channel 0 ■ 0x3 (FC_ANY): Flow controller is ANY for channel 0 **Value After Reset:** DMAH_CH0_FC_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
13	CH0_HC_LLP	R	The value of this register is derived from the DMAH_CH0_HC_LLP coreConsultant parameter. **Values:** ■ 0x0 (PROGRAMMABLE): Exclude logic to hardcode Channel 0 LLP register to 0 ■ 0x0 (HARDCODED): Hardcode Channel 0 LLP register to 0 **Value After Reset:** DMAH_CH0_HC_LLP_RST **Exists:** Always
12	CH0_CTL_WB_EN	R	The value of this register is derived from the DMAH_CH0_CTL_WB_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable control register writeback after each block transfer on channel 0 ■ 0x1 (TRUE): Include logic to enable control register writeback after each block transfer on channel 0 **Value After Reset:** DMAH_CH0_CTL_WB_EN_RST **Exists:** Always
11	CH0_MULTI_BLK_EN	R	The value of this register is derived from the DMAH_CH0_MULTI_BLK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel multi-block DMA transfers on channel 0 ■ 0x1 (TRUE): Include logic to enable channel multi-block DMA transfers on channel 0 **Value After Reset:** DMAH_CH0_MULTI_BLK_EN_RST **Exists:** Always
10	CH0_LOCK_EN	R	The value of this register is derived from the DMAH_CH0_LOCK_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable channel or bus locking on channel 0 ■ 0x1 (TRUE): Include logic to enable channel or bus locking on channel 0 **Value After Reset:** DMAH_CH0_LOCK_EN_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
9	CH0_SRC_GAT_EN	R	The value of this register is derived from the DMAH_CH0_SRC_GAT_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the gather feature on channel 0 ■ 0x1 (TRUE): Include logic to enable the gather feature on channel 0 **Value After Reset:** DMAH_CH0_SRC_GAT_EN_RST **Exists:** Always
8	CH0_DST_SCA_EN	R	The value of this register is derived from the DMAH_CH0_DST_SCA_EN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to enable the scatter feature on channel 0 ■ 0x1 (TRUE): Include logic to enable the scatter feature on channel 0 **Value After Reset:** DMAH_CH0_DST_SCA_EN_RST **Exists:** Always
7	CH0_STAT_SRC	R	The value of this register is derived from the DMAH_CH0_STAT_SRC coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from source peripheral of channel 0 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from source peripheral of channel 0 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH0_STAT_SRC_RST **Exists:** Always
6	CH0_STAT_DST	R	The value of this register is derived from the DMAH_CH0_STAT_DST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Exclude logic to fetch a status register from destination peripheral of channel 0 and write this status information to memory at end of each block transfer ■ 0x1 (TRUE): Include logic to fetch a status register from destination peripheral of channel 0 and write this status information to memory at end of each block transfer **Value After Reset:** DMAH_CH0_STAT_DST_RST **Exists:** Always

Table 5-56 Fields for Register: DMA_COMP_PARAMS_2 (Continued)

Bits	Name	Memory Access	Description
5:3	CH0_STW	R	The value of this register is derived from the DMAH_CH0_STW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 0's source transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 0's source transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 0's source transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 0's source transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 0's source transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 0's source transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH0_SRC_TR_RST **Exists:** Always
2:0	CH0_DTW	R	The value of this register is derived from the DMAH_CH0_DTW coreConsultant parameter. **Values:** ■ 0x0 (TRANS_WIDTH_PROGRAMMABLE): Programmable ■ 0x1 (TRANS_WIDTH_8): Hardcode channel 0's destination transfer width to 8 bits ■ 0x2 (TRANS_WIDTH_16): Hardcode channel 0's destination transfer width to 16 bits ■ 0x3 (TRANS_WIDTH_32): Hardcode channel 0's destination transfer width to 32 bits ■ 0x4 (TRANS_WIDTH_64): Hardcode channel 0's destination transfer width to 64 bits ■ 0x5 (TRANS_WIDTH_128): Hardcode channel 0's destination transfer width to 128 bits ■ 0x6 (TRANS_WIDTH_256): Hardcode channel 0's destination transfer width to 256 bits ■ 0x7 (RESERVED): Reserved **Value After Reset:** DMAH_CH0_DST_TR_RST **Exists:** Always

5.4.11 DMA_COMP_PARAMS_1

- **Name:** DW_ahb_dmac Component Parameters Register 1
- **Description:** DMA_COMP_PARAMS_1 is a constant read-only register that contains encoded information about the component parameter settings. The reset value depends on coreConsultant parameter(s).

  Note:If DMAH_RETURN_ERR_RESP is set to True, the DW_ahb_dmac returns an ERROR response to an illegal register access, which includes accessing registers that have been removed during DW_ahb_dmac configuration. If DMAH_RETURN_ERR_RESP is set to False, DW_ahb_dmac always returns an OK response. For more information, refer to "Illegal Register Access".
- **Size:** 64 bits
- **Offset:** 0x3f0
- **Exists:** DMAH_ADD_ENCODED_PARAMS

Bits	Field
63:62	Rsvd_1_DMA_COMP_PARAMS_1
61	STATIC_ENDIAN_SELECT
60	ADD_ENCODED_PARAMS
59:55	NUM_HS_INT
54:53	M1_HDATA_WIDTH
52:51	M2_HDATA_WIDTH
50:49	M3_HDATA_WIDTH
48:47	M4_HDATA_WIDTH
46:45	S_HDATA_WIDTH
44:43	NUM_MASTER_INT
42:40	NUM_CHANNELS
39:36	Rsvd_DMA_COMP_PARAMS_1
35	MAX_ABRST
34:33	INTR_IO
32	BIG_ENDIAN
31:28	CH7_MAX_BLK_SIZE
27:24	CH6_MAX_BLK_SIZE
23:20	CH5_MAX_BLK_SIZE
19:16	CH4_MAX_BLK_SIZE
15:12	CH3_MAX_BLK_SIZE
11:8	CH2_MAX_BLK_SIZE
7:4	CH1_MAX_BLK_SIZE
3:0	CH0_MAX_BLK_SIZE

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1

Bits	Name	Memory Access	Description
63:62	Rsvd_1_DMA_COMP_PARAMS_1	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
61	STATIC_ENDIAN_SELECT	R	The value of this register is derived from the DMAH_STATIC_ENDIAN_SELECT coreConsultant parameter. **Value After Reset:** DMAH_STATIC_ENDIAN_SELECT_RST **Exists:** Always
60	ADD_ENCODED_PARAMS	R	The value of this register is derived from the DMAH_ADD_ENCODED_PARAMS coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Add encoded parameter is not enabled ■ 0x1 (TRUE): Add encoded parameter is enabled ■ 0x0 (STATIC_ENDIAN_FALSE): Endianness is not configured through coreConsultant ■ 0x1 (STATIC_ENDIAN_TRUE): Endianness is statically configured through coreConsultant **Value After Reset:** DMAH_ADD_ENCODED_PARAMS **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
59:55	NUM_HS_INT	R	The value of this register is derived from the DMAH_NUM_HS_INT coreConsultant parameter. **Values:** ■ 0x0 (HS_INTERFACE_0): Number of handshaking interfaces is 0 ■ 0x1 (HS_INTERFACE_1): Number of handshaking interfaces is 1 ■ 0x2 (HS_INTERFACE_2): Number of handshaking interfaces is 2 ■ 0x3 (HS_INTERFACE_3): Number of handshaking interfaces is 3 ■ 0x4 (HS_INTERFACE_4): Number of handshaking interfaces is 4 ■ 0x5 (HS_INTERFACE_5): Number of handshaking interfaces is 5 ■ 0x6 (HS_INTERFACE_6): Number of handshaking interfaces is 6 ■ 0x7 (HS_INTERFACE_7): Number of handshaking interfaces is 7 ■ 0x8 (HS_INTERFACE_8): Number of handshaking interfaces is 8 ■ 0x9 (HS_INTERFACE_9): Number of handshaking interfaces is 9 ■ 0xa (HS_INTERFACE_a): Number of handshaking interfaces is 10 ■ 0xb (HS_INTERFACE_b): Number of handshaking interfaces is 11 ■ 0xc (HS_INTERFACE_c): Number of handshaking interfaces is 12 ■ 0xd (HS_INTERFACE_d): Number of handshaking interfaces is 13 ■ 0xe (HS_INTERFACE_e): Number of handshaking interfaces is 14 ■ 0xf (HS_INTERFACE_f): Number of handshaking interfaces is 15 ■ 0x10 (HS_INTERFACE_10): Number of handshaking interfaces is 16 **Value After Reset:** DMAH_NUM_HS_INT_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
54:53	M1_HDATA_WIDTH	R	The value of this register is derived from the DMAH_M1_HDATA_WIDTH coreConsultant parameter. **Values:** ■ 0x0 (DATA_BUS_WIDTH_32): Master1 interface data bus width is 32 bits ■ 0x1 (DATA_BUS_WIDTH_64): Master1 interface data bus width is 64 bits ■ 0x2 (DATA_BUS_WIDTH_128): Master1 interface data bus width is 128 bits ■ 0x3 (DATA_BUS_WIDTH_256): Master1 interface data bus width is 256 bits **Value After Reset:** DMAH_M1_HDATA_WIDTH_RST **Exists:** Always
52:51	M2_HDATA_WIDTH	R	The value of this register is derived from the DMAH_M2_HDATA_WIDTH coreConsultant parameter. **Values:** ■ 0x0 (DATA_BUS_WIDTH_32): Master2 interface data bus width is 32 bits ■ 0x1 (DATA_BUS_WIDTH_64): Master2 interface data bus width is 64 bits ■ 0x2 (DATA_BUS_WIDTH_128): Master2 interface data bus width is 128 bits ■ 0x3 (DATA_BUS_WIDTH_256): Master2 interface data bus width is 256 bits **Value After Reset:** DMAH_M2_HDATA_WIDTH_RST **Exists:** Always
50:49	M3_HDATA_WIDTH	R	The value of this register is derived from the DMAH_M3_HDATA_WIDTH coreConsultant parameter. **Values:** ■ 0x0 (DATA_BUS_WIDTH_32): Master3 interface data bus width is 32 bits ■ 0x1 (DATA_BUS_WIDTH_64): Master3 interface data bus width is 64 bits ■ 0x2 (DATA_BUS_WIDTH_128): Master3 interface data bus width is 128 bits ■ 0x3 (DATA_BUS_WIDTH_256): Master3 interface data bus width is 256 bits **Value After Reset:** DMAH_M3_HDATA_WIDTH_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
48:47	M4_HDATA_WIDTH	R	The value of this register is derived from the DMAH_M4_HDATA_WIDTH coreConsultant parameter. **Values:** ■ 0x0 (DATA_BUS_WIDTH_32): Master4 interface data bus width is 32 bits ■ 0x1 (DATA_BUS_WIDTH_64): Master4 interface data bus width is 64 bits ■ 0x2 (DATA_BUS_WIDTH_128): Master4 interface data bus width is 128 bits ■ 0x3 (DATA_BUS_WIDTH_256): Master4 interface data bus width is 256 bits **Value After Reset:** DMAH_M4_HDATA_WIDTH_RST **Exists:** Always
46:45	S_HDATA_WIDTH	R	The value of this register is derived from the DMAH_S_HDATA_WIDTH coreConsultant parameter. **Values:** ■ 0x0 (DATA_BUS_WIDTH_32): Slave interface data bus width is 32 bits ■ 0x1 (DATA_BUS_WIDTH_64): Slave interface data bus width is 64 bits ■ 0x2 (DATA_BUS_WIDTH_128): Slave interface data bus width is 128 bits ■ 0x3 (DATA_BUS_WIDTH_256): Slave interface data bus width is 256 bits **Value After Reset:** DMAH_S_HDATA_WIDTH_RST **Exists:** Always
44:43	NUM_MASTER_INT	R	The value of this register is derived from the DMAH_NUM_MASTER_INT coreConsultant parameter. **Values:** ■ 0x0 (NUM_MST_INTERFACE_1): Number of MASTER interface is 1 ■ 0x1 (NUM_MST_INTERFACE_2): Number of MASTER interface is 2 ■ 0x2 (NUM_MST_INTERFACE_3): Number of MASTER interface is 3 ■ 0x3 (NUM_MST_INTERFACE_4): Number of MASTER interface is 4 **Value After Reset:** DMAH_NUM_MASTER_INT_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
42:40	NUM_CHANNELS	R	The value of this register is derived from the DMAH_NUM_CHANNELS coreConsultant parameter. **Values:** ■ 0x0 (NUM_CHANNEL_1): Number of DMA Channels is 1 ■ 0x1 (NUM_CHANNEL_2): Number of DMA Channels is 2 ■ 0x2 (NUM_CHANNEL_3): Number of DMA Channels is 3 ■ 0x3 (NUM_CHANNEL_4): Number of DMA Channels is 4 ■ 0x4 (NUM_CHANNEL_5): Number of DMA Channels is 5 ■ 0x5 (NUM_CHANNEL_6): Number of DMA Channels is 6 ■ 0x6 (NUM_CHANNEL_7): Number of DMA Channels is 7 ■ 0x7 (NUM_CHANNEL_8): Number of DMA Channels is 8 **Value After Reset:** DMAH_NUM_CHANNELS_RST **Exists:** Always
39:36	Rsvd_DMA_COMP_PARAMS_1	R	Reserved field- read-only **Value After Reset:** 0x0 **Exists:** Always
35	MAX_ABRST	R	The value of this register is derived from the DMAH_MABRST coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Maximum AMBA burst length is not under the control of software ■ 0x1 (TRUE): Limit the maximum AMBA burst length to a value under software control by writing to the channel configuration register **Value After Reset:** DMAH_MABRST_RST **Exists:** Always
34:33	INTR_IO	R	The value of this register is derived from the DMAH_INTR_IO coreConsultant parameter. **Values:** ■ 0x0 (ALL_INT): ALL interrupt-related signals appear as outputs on the design ■ 0x1 (TYPE_INT): Only TYPE interrupt-related signals appear as outputs on the design ■ 0x2 (COMBINED_INT): Only COMBINED interrupt-related signals appear as outputs on the design ■ 0x3 (RESERVED): Reserved **Value After Reset:** DMAH_INTR_IO_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
32	BIG_ENDIAN	R	The value of this register is derived from the DMAH_BIG_ENDIAN coreConsultant parameter. **Values:** ■ 0x0 (FALSE): Big Endian ■ 0x1 (TRUE): Little Endian **Value After Reset:** DMAH_BIG_ENDIAN_RST **Exists:** Always
31:28	CH7_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH7_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 7 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 7 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 7 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 7 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 7 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 7 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 7 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 7 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 7 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 7 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 7 **Value After Reset:** DMAH_CH7_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
27:24	CH6_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH6_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 6 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 6 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 6 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 6 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 6 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 6 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 6 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 6 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 6 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 6 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 6 **Value After Reset:** DMAH_CH6_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
23:20	CH5_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH5_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 5 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 5 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 5 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 5 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 5 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 5 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 5 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 5 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 5 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 5 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 5 **Value After Reset:** DMAH_CH5_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
19:16	CH4_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH4_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 4 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 4 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 4 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 4 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 4 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 4 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 4 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 4 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 4 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 4 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 4 **Value After Reset:** DMAH_CH4_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
15:12	CH3_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH3_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 3 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 3 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 3 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 3 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 3 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 3 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 3 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 3 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 3 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 3 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 3 **Value After Reset:** DMAH_CH3_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
11:8	CH2_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH2_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 2 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 2 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 2 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 2 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 2 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 2 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 2 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 2 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 2 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 2 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 2 **Value After Reset:** DMAH_CH2_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
7:4	CH1_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH1_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 1 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 1 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 1 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 1 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 1 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 1 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 1 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 1 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 1 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 1 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 1 **Value After Reset:** DMAH_CH1_MAX_BLK_SIZE_RST **Exists:** Always

Table 5-57 Fields for Register: DMA_COMP_PARAMS_1 (Continued)

Bits	Name	Memory Access	Description
3:0	CHO_MAX_BLK_SIZE	R	The values of these bit fields are derived from the DMAH_CH0_MAX_BLK_SIZE coreConsultant parameter. **Values:** ■ 0x0 (MAX_BLOCK_SIZE_3): Maximum block size in source transfer widths is 3 for channel 0 ■ 0x1 (MAX_BLOCK_SIZE_7): Maximum block size in source transfer widths is 7 for channel 0 ■ 0x2 (MAX_BLOCK_SIZE_15): Maximum block size in source transfer widths is 15 for channel 0 ■ 0x3 (MAX_BLOCK_SIZE_31): Maximum block size in source transfer widths is 31 for channel 0 ■ 0x4 (MAX_BLOCK_SIZE_63): Maximum block size in source transfer widths is 63 for channel 0 ■ 0x5 (MAX_BLOCK_SIZE_127): Maximum block size in source transfer widths is 127 for channel 0 ■ 0x6 (MAX_BLOCK_SIZE_255): Maximum block size in source transfer widths is 255 for channel 0 ■ 0x7 (MAX_BLOCK_SIZE_511): Maximum block size in source transfer widths is 511 for channel 0 ■ 0x8 (MAX_BLOCK_SIZE_1023): Maximum block size in source transfer widths is 1023 for channel 0 ■ 0x9 (MAX_BLOCK_SIZE_2047): Maximum block size in source transfer widths is 2047 for channel 0 ■ 0xa (MAX_BLOCK_SIZE_4095): Maximum block size in source transfer widths is 4095 for channel 0 **Value After Reset:** DMAH_CH0_MAX_BLK_SIZE_RST **Exists:** Always

5.4.12 DmaCompsID

- **Name:** DMA Component ID register
- **Description:** This is the DW_ahb_dmac Component Version register, which is a read-only register that specifies the version of the packaged component in the upper 32 bits and the component type in the lower 32 bits.
- **Size:** 64 bits
- **Offset:** 0x3f8
- **Exists:** Always

63:32	31:0
DMA_COMP_VERSION	DMA_COMP_TYPE

Table 5-58 Fields for Register: DmaCompsID

Bits	Name	Memory Access	Description
63:32	DMA_COMP_VERSION	R	DMA Component Version - See release notes. **Value After Reset:** DMAH_VERSION_ID **Exists:** Always
31:0	DMA_COMP_TYPE	R	DMA Component Type Number = `h44571110. This assigned unique hex value is constant and is derived from the two ASCII letters "DW" followed by a 32-bit unsigned number **Value After Reset:** DMAH_COMP_ID **Exists:** Always

6
Programming the DW_ahb_dmac

The DW_ahb_dmac can be programmed through software registers or the DW_ahb_dmac low-level software driver; software registers are described in more detail in "Register Descriptions" on page 161.

Note There are references to both software and hardware parameters throughout this chapter. The software parameters are the field names in each register description table and are prefixed by the register name; for example, the Block Transfer Size field in the Control Register for Channel *x* is designated as "CTL*x*.BLOCK_TS."

The hardware parameters are prefixed with an DMAH_* and are configured once using Synopsys coreConsultant.

Shipped with the DW_ahb_dmac component is an address definition (memory map) C header file. This can be used when the DW_ahb_dmac is programmed in a C environment.

6.1 Software Drivers

The family of DesignWare Synthesizable Components includes a Driver Kit for the DW_ahb_dmac component. This low-level Driver Kit allows you to program a DW_ahb_dmac component and integrate your code into a larger software system. The Driver Kit provides the following benefits to IP designers:

- Proven method of access to DW_ahb_dmac minimizing usage errors
- Rapid software development with minimum overhead
- Detailed knowledge of DW_ahb_dmac register bit fields not required
- Easy integration of DW_ahb_dmac into existing software system
- Programming at register level eliminated

You must purchase a source code license (DWC-DMA-Controller-Source) to use the DW_ahb_dmac Driver Kit. However, you can access some Driver Kit files and documentation in $DESIGNWARE_HOME/drivers/DW_ahb_dmac/latest. For more information about the Driver Kit, refer to the *DW_ahb_dmac Driver Kit User Guide*. For more information about purchasing the source code license and obtaining a download of the Driver Kit, contact Synopsys at designware@synopsys.com for details.

6.2 Register Access

All registers are aligned to a 64-bit boundary and are 64 bits wide. In general, the upper 32 bits of a register are reserved. A write to reserved bits within the register is ignored. A read from reserved bits in the register reads back 0. To avoid address aliasing, do one of the following:

1. The DW_ahb_dmac should not be allocated more than 1 KB of address space in the system memory map. If it is, then addresses selected above 1 KB from the base address are aliased to an address within the 1 KB space, and a transfer takes place involving this register.
2. Software should not attempt to access non-register locations when hsel is asserted.

Note The hsel signal is asserted by the system decoder when the address on the bus is within the system address assigned for DW_ahb_dmac.

6.3 Illegal Register Access

An illegal access can be any of the following:

1. A AHB transfer of hsize greater than 64 is attempted.
2. The hsel signal is asserted, but the address does not decode to a valid address.
3. A write to the SARx, DARx, LLPx, CTLx, SSTATx, DSTATx, SSTATARx, DSTATARx, SGRx, or DSRx registers occurs when the channel is enabled.
4. A read from the ClearBlock, ClearDstTran, ClearErr, ClearSrcTran, ClearTfris attempted.
5. A write to the StatusBlock, StatusDstTran, StatusErr, StatusSrcTran, StatusTfris attempted.
6. A write to the StatusIntregister is attempted.
7. A write to either the DmaIdReg or DMA Component ID Register register is attempted.

The response to an illegal access is configured using the configuration parameter DMAH_RETURN_ERR_RESP. When DMAH_RETURN_ERR_RESP is set to True, an illegal access (read/write) returns an error response.

If DMAH_RETURN_ERR_RESP is set to False, an OKAY response is returned, a read reads back 0x0, and a write is ignored.

6.4 DW_ahb_dmac Transfer Types

A DMA transfer may consist of single or multi-block transfers. On successive blocks of a multi-block transfer, the SAR*x*/DAR*x* register in the DW_ahb_dmac is reprogrammed using either of the following methods:

- Block chaining using linked lists
- Auto-reloading
- Contiguous address between blocks

On successive blocks of a multi-block transfer, the CTL*x* register in the DW_ahb_dmac is reprogrammed using either of the following methods:

- Block chaining using linked lists
- Auto-reloading

When block chaining, using Linked Lists is the multi-block method of choice. On successive blocks, the LLP*x* register in the DW_ahb_dmac is reprogrammed using block chaining with linked lists.

A block descriptor consists of six registers: SARx, DARx, LLPx, CTLx, SSTATx, and DSTATx. The first four registers, along with the CFG*x* register, are used by the DW_ahb_dmac to set up and describe the block transfer.

Note The term Link List Item (LLI) and block descriptor are synonymous.

6.4.1 Multi-Block Transfers

Multi-block transfers are enabled by setting the DMAH_CHX_MULTI_BLK_EN configuration parameter to True.

Note Multi-block transfers—in which the source and destination are swapped during the transfer—are not supported. In a multi-block transfer, the direction must not change for the duration of the transfer.

6.4.1.1 Block Chaining Using Linked Lists

To enable multi-block transfers using block chaining, you must set the configuration parameter DMAH_CHx_MULTI_BLK_EN to True and the DMAH_CHx_HC_LLP parameter to False.

In this case, the DW_ahb_dmac reprograms the channel registers prior to the start of each block by fetching the block descriptor for that block from system memory. This is known as an LLI update.

DW_ahb_dmac block chaining uses a Linked List Pointer register (LLPx) that stores the address in memory of the next linked list item. Each LLI contains the corresponding block descriptors:

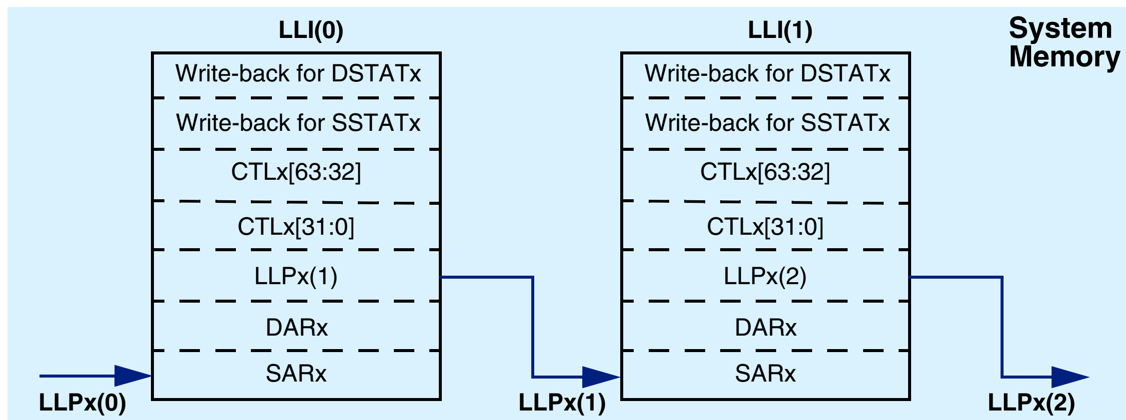
1. SARx
2. DARx
3. LLPx
4. CTLx
5. SSTATx
6. DSTATx

To set up block chaining, you program a sequence of Linked Lists in memory.

LLI accesses are always 32-bit accesses (Hsize = 2) aligned to 32-bit boundaries and cannot be changed or programmed to anything other than 32-bit, even if the AHB master interface of the LLI supports more than a 32-bit data width.

The SAR*x*, DAR*x*, LLP*x*, and CTL*x* registers are fetched from system memory on an LLI update. If configuration parameter DMAH_CHx_CTL_WB_EN = True, then the updated contents of the CTL*x*, SSTAT*x*, and DSTAT*x* registers are written back to memory on block completion. Figure 6-1 and Figure 6-2 show how you use chained linked lists in memory to define multi-block transfers using block chaining.

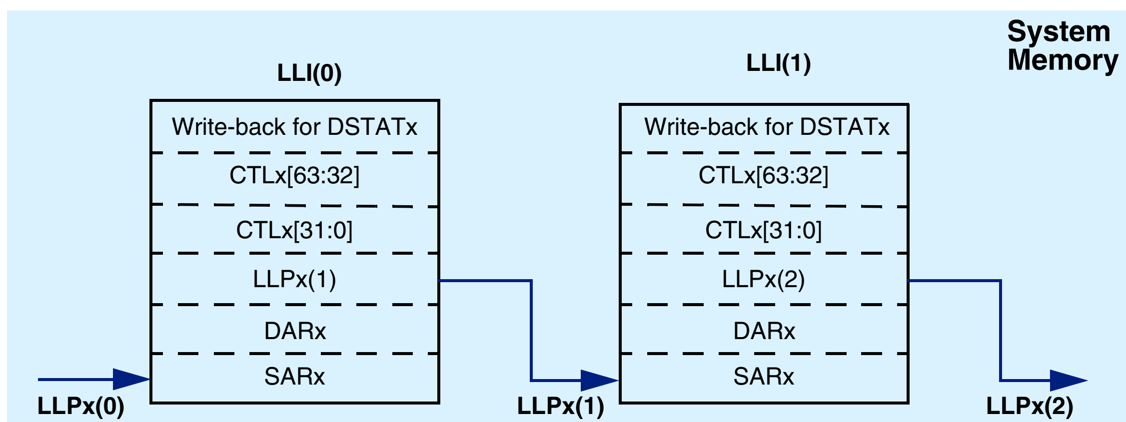
Figure 6-1 Multi-Block Transfer Using Linked Lists When DMAH_CHx_STAT_SRC Set to True



It is assumed that no allocation is made in system memory for the source status when the configuration parameter DMAH_CHx_STAT_SRC is set to False. If this parameter is False, then the order of a Linked List item is as follows:

1. SARx
2. DARx
3. LLPx
4. CTLx
5. DSTATx

Figure 6-2 Multi-Block Transfer Using Linked Lists When DMAH_CHx_STAT_SRC Set to False



Note In order to not confuse the SAR*x*, DAR*x*, LLP*x*, CTL*x*, STAT*x*, and DSTAT*x* register locations of the LLI with the corresponding DW_ahb_dmac memory mapped register locations, the LLI register locations are prefixed with LLI; that is, LLI.SAR*x*, LLI.DAR*x*, LLI.LLP*x*, LLI.CTL*x*, LLI.SSTAT*x*, and LLI.DSTAT*x*.

Figure 6-3 and Figure 6-4 show the mapping of a Linked List Item stored in memory to the channel registers block descriptor.

Rows 6 through 10 of Table 6-1 show the required values of LLP*x*, CTL*x*, and CFG*x* for multi-block DMA transfers using block chaining.

Note For rows 6 through 10 of Table 6-1, the LLI.CTL*x*, LLI.LLP*x*, LLI.SAR*x*, and LLI.DAR*x* register locations of the LLI are always affected at the start of every block transfer. The LLI.LLP*x* and LLI.CTL*x* locations are always used to reprogram the DW_ahb_dmac LLP*x* and CTL*x* registers. However, depending on the Table 6-1 row number, the LLI.SAR*x*/LLI.DAR*x* address may or may not be used to reprogram the DW_ahb_dmac SAR*x*/DAR*x* registers.

Table 6-1 Programming of Transfer Types and Channel Register Update Method

Transfer Type	LLP. LOC = 0	LLP_ SRC_EN (CTLx)	RELOAD _SRC (CFGx)	LLP_ DST_EN (CTLx)	RELOAD _DST (CFGx)	CTLx, LLPx Update Method	SARx Update Method	DARx Update Method	Write Back[a]
1. Single-block or last transfer of multi-block.	Yes	0	0	0	0	None, user reprograms	None (single)	None (single)	No
2. Auto-reload multi-block transfer with contiguous SAR	Yes	0	0	0	1	CTLx, LLPx are reloaded from initial values.	Con-tiguous	Auto-Reload	No
3. Auto-reload multi-block transfer with contiguous DAR.	Yes	0	1	0	0	CTLx, LLPx are reloaded from initial values	Auto-Reload	Con-tiguous	No
4. Auto-reload multi-block transfer	Yes	0	1	0	1	CTLx, LLPx are reloaded from initial values	Auto-reload	Auto-Reload	No
5. Single-block or last transfer of multi-block.	No	0	0	0	0	None, user reprograms	None (single)	None (single)	Yes
6. Linked list multi-block transfer with contiguous SAR	No	0	0	1	0	CTLx, LLPx loaded from next Linked List item.	Con-tiguous	Linked List	Yes

Transfer Type	LLP. LOC = 0	LLP_ SRC_EN (CTLx)	RELOAD _SRC (CFGx)	LLP_ DST_EN (CTLx)	RELOAD _DST (CFGx)	CTLx, LLPx Update Method	SARx Update Method	DARx Update Method	Write Back[a]
7. Linked list multi-block transfer with auto-reload SAR	No	0	1	1	0	CTLx, LLPx loaded from next Linked List item.	Auto-Reload	Linked List	Yes
8. Linked list multi-block transfer with contiguous DAR	No	1	0	0	0	CTLx, LLPx loaded from next Linked List item.	Linked List	Con-tiguous	Yes
9. Linked list multi-block transfer with auto-reload DAR	No	1	0	0	1	CTLx, LLPx loaded from next Linked List item.	Linked List	Auto-Reload	Yes
10. Linked list multi-block transfer	No	1	0	1	0	CTLx, LLPx loaded from next Linked List item.	Linked List	Linked List	Yes

a. This column assumes that the configuration parameter DMAH_CHx_CTL_WB_EN = True. If DMAH_CHx_CTL_WB_EN = False, then there is never writeback of the control and status registers regardless of transfer type, and all rows of this column are "No".

Figure 6-3 Mapping of Block Descriptor (LLI) in Memory to Channel Registers When DMAH_CHx_STAT_SRC Set to True

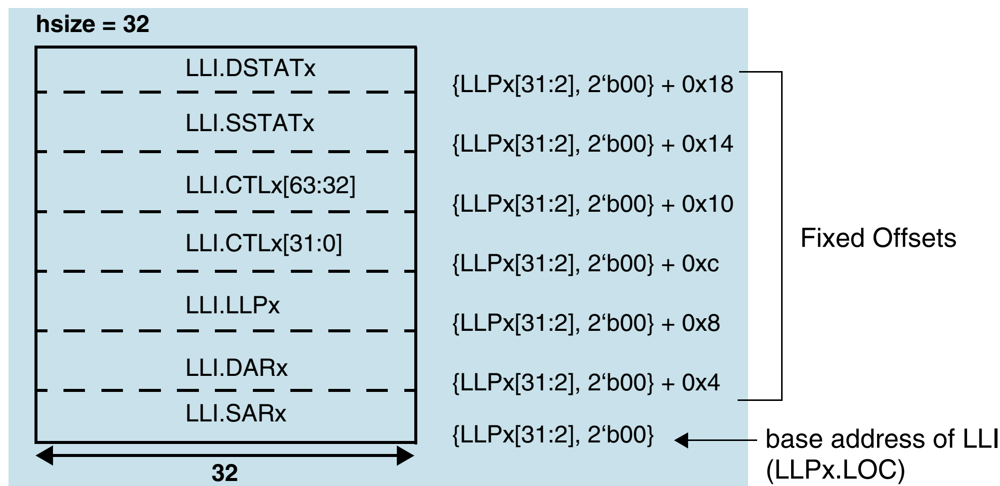
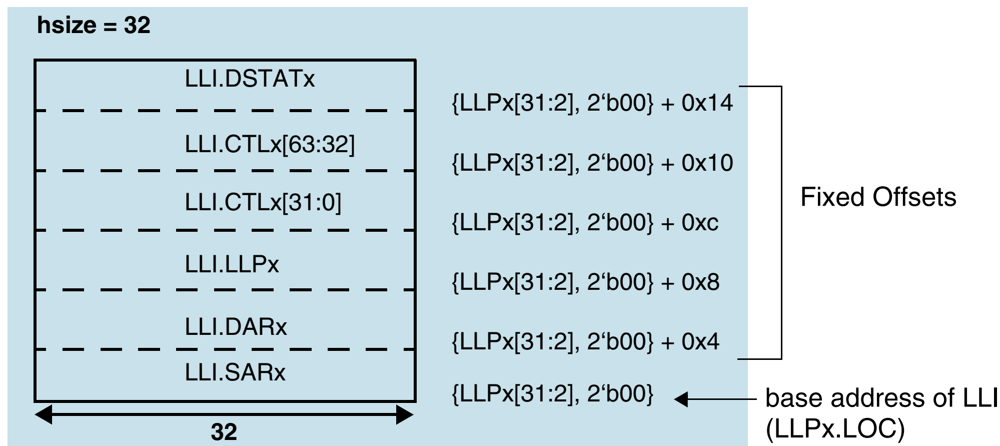


Figure 6-4 Mapping of Block Descriptor (LLI) in Memory to Channel Registers When DMAH_CHx_STAT_SRC Set to False



- Throughout this databook, there are descriptions about fetching the LLI.CTL*x* register from the location pointed to by the LLP*x* register. This exact location is the LLI base address (stored in LLP*x* register) plus the fixed offset. For example, in Figure 6-3, the location of the LLI.CTL*x* register is LLP*x*.LOC + 0xc.
- Referring to Table 6-1, if the Write Back column entry is "Yes" and the configuration parameter DMAH_CHx_CTL_WB_EN = True, then the CTL*x*[63:32] register is always written to system memory (to LLI.CTL*x*[63:32]) at the end of every block transfer.

  The source status is fetched and written to system memory at the end of every block transfer if the Write Back column entry is "Yes," DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.*SS_UPD_EN* is enabled.

  The destination status is fetched and written to system memory at the end of every block transfer if the Write Back column entry is "Yes," DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.*DS_UPD_EN* is enabled.

6.4.2 Auto-Reloading of Channel Registers

During auto-reloading, the channel registers are reloaded with their initial values at the completion of each block and the new values used for the new block. Depending on the row number in Table 6-1, some or all of the SAR*x*, DAR*x*, and CTL*x* channel registers are reloaded from their initial value at the start of a block transfer.

6.4.3 Contiguous Address Between Blocks

In this case, the address between successive blocks is selected as a continuation from the end of the previous block.

Enabling the source or destination address to be contiguous between blocks is a function of the CTLx.LLP_SRC_EN, CFGx.RELOAD_SRC, CTLx.LLP_DST_EN, and CTLx.RELOAD_DST registers (see Table 6-1).

Note You cannot select both SAR*x* and DAR*x* updates to be contiguous. If you want this functionality, you should increase the size of the Block Transfer (CTL*x*.*BLOCK_TS*), or if this is at the maximum value, use Row 10 of Table 6-1 and set up the LLI.SAR*x* address of the block descriptor to be equal to the end SAR*x* address of the previous block. Similarly, set up the LLI.DAR*x* address of the block descriptor to be equal to the end DAR*x* address of the previous block. For more information, refer to "Multi-Block Transfer with Linked List for Source and Linked List for Destination (Row 10)" on page 345.

6.4.4 Suspension of Transfers Between Blocks

At the end of every block transfer, an end-of-block interrupt is asserted if:

1. Interrupts are enabled, CTLx.INT_EN = 1, and
2. The channel block interrupt is unmasked, MaskBlock[*n*] = 1, where *n* is the channel number.

Note The block-complete interrupt is generated at the completion of the block transfer to the destination.

For rows 6, 8, and 10 of Table 6-1, the DMA transfer does not stall between block transfers. For example, at the end-of-block *N*, the DW_ahb_dmac automatically proceeds to block *N* + 1.

For rows 2, 3, 4, 7, and 9 of Table 6-1 (SAR*x* and/or DAR*x* auto-reloaded between block transfers), the DMA transfer automatically stalls after the end-of-block interrupt is asserted, if the end-of-block interrupt is enabled and unmasked.

The DW_ahb_dmac does not proceed to the next block transfer until a write to the ClearBlock[*n*] block interrupt clear register, done by software to clear the channel block-complete interrupt, is detected by hardware.

For rows 2, 3, 4, 7, and 9 of Table 6-1 (SAR*x* and/or DAR*x* auto-reloaded between block transfers), the DMA transfer does not stall if either:

- Interrupts are disabled, CTLx.INT_EN = 0, or
- The channel block interrupt is masked, MaskBlock[*n*] = 0, where *n* is the channel number.

Channel suspension between blocks is used to ensure that the end-of-block ISR (interrupt service routine) of the next-to-last block is serviced before the start of the final block commences. This ensures that the ISR has cleared the CFGx.RELOAD_SRC and/or CFGx.RELOAD_DST bits before completion of the final block. The reload bits CFGx.RELOAD_SRC and/or CFGx.RELOAD_DST should be cleared in the end-of-block ISR for the next-to-last block transfer.

6.4.5 Ending Multi-Block Transfers

All multi-block transfers must end as shown in either Row 1 or Row 5 of Table 6-1. At the end of every block transfer, the DW_ahb_dmac samples the row number, and if the DW_ahb_dmac is in the Row 1 or Row 5 state, then the previous block transferred is the last block and the DMA transfer is terminated.

Note Row 1 and Row 5 are used for single-block transfers or terminating multi-block transfers. Transfers initiated in rows 2, 3 or 4 can only end in row 1; similarly, transfers initiated in rows 6 through 10 can only end in row 5. Ending in the Row 5 state enables status fetch and write-back for the last block. Ending in the Row 1 state disables status fetch and write-back for the last block.

For rows 2, 3, and 4 of Table 6-1, (LLP*x*.LOC = 0 and CFGx.RELOAD_SRC and/or CFGx.RELOAD_DST is set), multi-block DMA transfers continue until both the CFGx.RELOAD_SRC and CFGx.RELOAD_DST registers are cleared by software. They should be programmed to 0 in the end-of-block interrupt service routine that services the next-to-last block transfer; this puts the DW_ahb_dmac into the Row 1 state.

For rows 6, 8, and 10 of Table 6-1 (both CFGx.RELOAD_SRC and CFGx.RELOAD_DST cleared), you must set up the last block descriptor in memory so that both LLI.CTL*x*.LLP_SRC_EN and LLI.CTL*x*.LLP_DST_EN are 0.

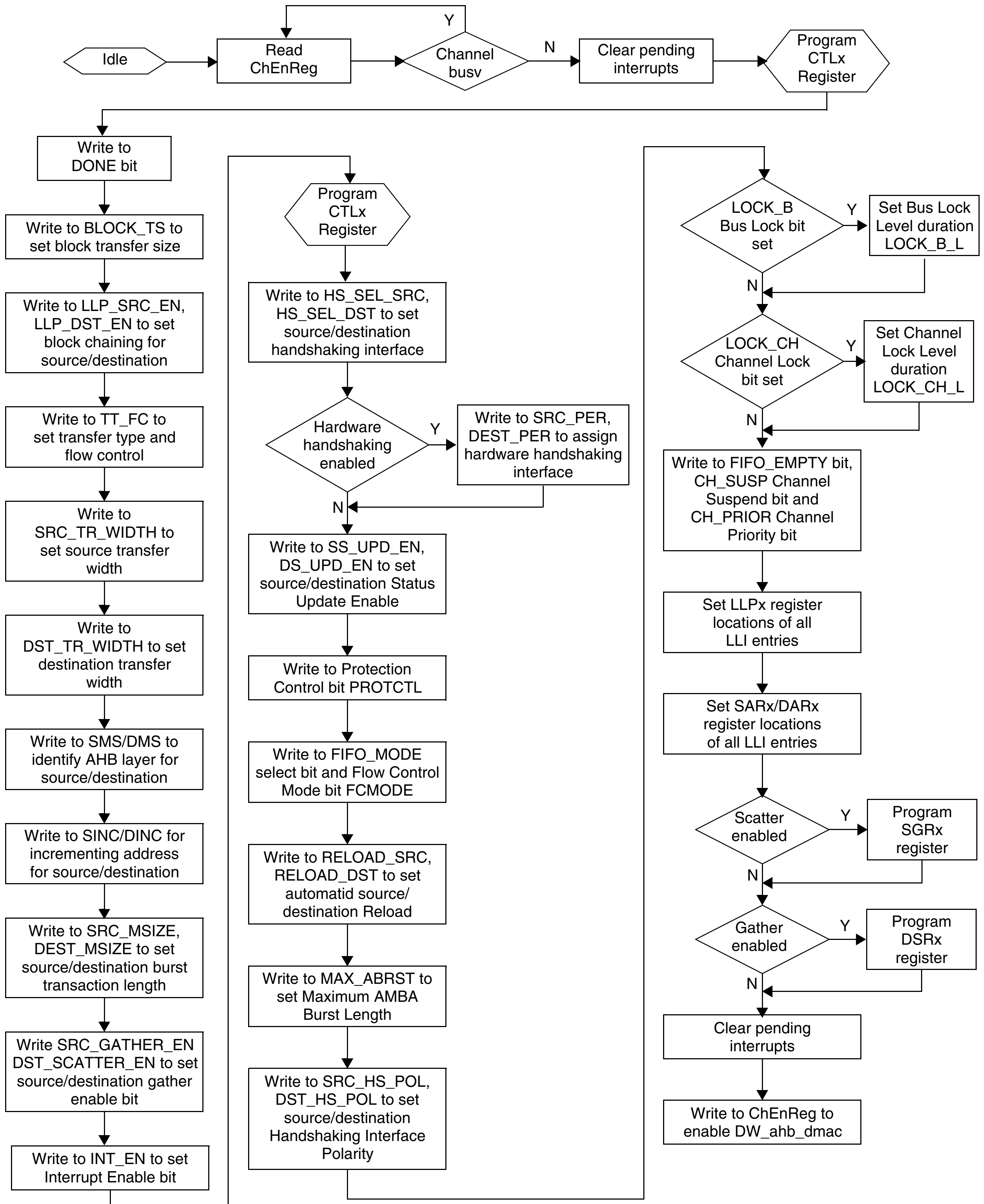
The sampling of the LLPx.LOC bit takes place exclusively at the beginning of the transfer when the channel is enabled. This determines whether writeback is enabled throughout the complete transfer, and changing the value of this bit in subsequent blocks on the same transfer does not have any effect.

Note The only allowed transitions between the rows of Table 6-1 are from any row into Row 1 or Row 5. As already stated, a transition into row 1 or row 5 is used to terminate the DMA transfer; all other transitions between rows are not allowed. Software must ensure that illegal transitions between rows do not occur between blocks of a multi-block transfer. For example, if block *N* is in row 10, then the only allowed rows for block *N* +1 are rows are rows 10 or 5.

6.5 Programing Examples

The flow diagram in Figure 6-5 shows an overview of programming the DMA described in "Programming Example for Linked List Multi-Block Transfer" on page 341.

Figure 6-5 Flowchart for DMA Programming Example



6.5.1 Programming Example for Linked List Multi-Block Transfer

This section explains the step-by-step programming of the DW_ahb_dmac. The example demonstrates row 10 of Table 6-1 for Multi-Block Transfer with Linked List for Source and Linked List for Destination. This example uses the DW_ahb_dmac to move four blocks of contiguous data from source to destination memory using the Linked List feature.

1. Set up the chain of Linked List items—otherwise known as block descriptors—in memory. Write the control information in the LLI.CTLx register location of the block descriptor for each LLI in memory for Channel 1. In the LLI.CTLx register, the following is programmed:

   a. Set up the transfer type for a memory-to-memory transfer:

   ```
   - ctlx[22:20] = 3'b000;
   ```

   b. Set up the transfer characteristics:

      i. Transfer width for the source in the SRC_TR_WIDTH field

      ```
      - ctlx[6:4] = 3'b001;
      ```

      ii. Transfer width for the destination in the DST_TR_WIDTH field

      ```
      - ctlx[3:1] = 3'b001;
      ```

      iii. Source master layer in the SMS field where the source resides

      ```
      - ctlx[26:25] = 2'b00;
      ```

      iv. Destination master layer in the DMS field where the destination resides

      ```
      - ctlx[24:23] = 2'b00;
      ```

      v. Incrementing address for the source in the SINC field

      ```
      - ctlx[10:9] = 2'b00;
      ```

      vi. Incrementing address for the destination in the DINC field

      ```
      - ctlx[8:7] = 2'b00;
      ```

2. Write the channel configuration information into the CFGx register for Channel 1:

   a. HS_SEL_SRC/HS_SEL_DST bits select which of the handshaking interfaces—hardware or software—is active for source requests on this channel.

   ```
   - cfgx[11] = 1'b0;
   - cfgx[10] = 1'b0;
   ```

   These settings are ignored because both the source and destination are memory types.

   b. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripheral by programming the SRC_PER and DEST_PER bits:

   ```
   - cfgx[46:43] = 1'b0;
   - cfgx[42:39] = 1'b0;
   ```

   These settings are ignored because both the source and destination are memory types.

3. The following For loop, shown as a programming example, sets the following:

   - LLI.LLPx register locations of all LLI entries in memory (except the last) to non-zero and point to the base address of the next Linked List Item
   - LLI.SARx/LLI.DARx register locations of all LLI entries in memory point to the start source/destination block address preceding that LLI fetch

The For statement below configures the LLPx entries:

```
for(i=0; i < 4; i=i+1) begin
   if (i == 3)
   llpx = 0;   // end of LLI
else
   llpx = llp_addr + 20; // start of next LLI

 //-: Program SAR
 `AHB_MASTER.write(0, llp_addr, sarx, AhbWord32Attrb, handle[0]);
 //-: Program DAR
 `AHB_MASTER.write(0, (llp_addr + 4), darx, AhbWord32Attrb, handle[0]);

 //-: Program LLP
 `AHB_MASTER.write(0, (llp_addr + 8), llpx, AhbWord32Attrb, handle[0]);

 //-: Program CTL
 `AHB_MASTER.write(0, (llp_addr + 12),  ctlx[31:0], AhbWord32Attrb, handle[0]);
 `AHB_MASTER.write(0, (llp_addr + 16), ctlx[63:32], AhbWord32Attrb, handle[0]);

 // update pointers
 llp_addr = llp_addr + 20; // start of next LLI

 // 4  16-bit words each with scatter/gather interval in each block
 // (works only with scatter_gather count of 2)
 sarx = sarx + 24;
 darx = darx + 24;

end
```

4. If Gather is enabled—DMAH_CHx_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled— program the SGRx register for Channel 1.
5. If Scatter is enabled—DMAH_CHx_DST_SCA_EN = True and CTLx.DST_SCATTER_EN is enabled—program the DSRx register for Channel 1.
6. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers.
7. Finally, enable the channel by writing a 1 to the ChEnReg.CH_EN bit; the transfer is performed.

6.6 Programming a Channel

Three registers – LLPx, CTLx, and CFGx – need to be programmed to determine whether single- or multi-block transfers occur, and which type of multi-block transfer is used. The different transfer types are shown in Table 6-1.

The DW_ahb_dmac can be programmed to fetch the status from the source or destination peripheral; this status is stored in the SSTATx and DSTATx registers. When the DW_ahb_dmac is programmed to fetch the status from the source or destination peripheral, it writes this status and the contents of the CTL*x* register back to memory at the end of a block transfer. The Write Back column of Table 6-1 shows when this occurs.

The "Update Method" columns indicate where the values of SARx, DARx, CTLx, and LLPx are obtained for the next block transfer when multi-block DW_ahb_dmac transfers are enabled.

Note In Table 6-1, all other combinations of LLPx.*LOC* = 0, CTLx.LLP_SRC_EN, CFGx.RELOAD_SRC, CTLx.LLP_DST_EN, and CFGx.RELOAD_DST are illegal, and causes indeterminate or erroneous behavior.

6.6.1 Programming Examples

- "Single-block Transfer (Row 1)" on page 343
- "Multi-Block Transfer with Linked List for Source and Linked List for Destination (Row 10)" on page 345
- "Multi-Block Transfer with Source Address Auto-Reloaded and Destination Address Auto-Reloaded (Row 4)" on page 350
- "Multi-Block Transfer with Source Address Auto-Reloaded and Linked List Destination Address (Row 7)" on page 353
- "Multi-Block Transfer with Source Address Auto-Reloaded and Contiguous Destination Address (Row 3)" on page 358
- "Multi-Block DMA Transfer with Linked List for Source and Contiguous Destination Address (Row 8)" on page 362

6.6.1.1 Single-block Transfer (Row 1)

This section describes a single-block transfer, Row 1 in Table 6-1.

Note Row 5 in Table 6-1 is also a single-block transfer with write-back of control and status information enabled at the end of the single-block transfer.

1. Read the Channel Enable register to choose a free (disabled) channel; refer to "ChEnReg".
2. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
3. Program the following channel registers:
   a. Write the starting source address in the SARx register for channel *x*.
   b. Write the starting destination address in the DARx register for channel *x*.
   c. Program CTL*x* and CFG*x* according to Row 1, as shown in Table 6-1. Program the LLPx register with 0.
   d. Write the control information for the DMA transfer in the CTLx register for channel *x*. For example, in the register, you can program the following:
      i. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control device by programming the TT_FC of the CTL*x* register. Table 2-1 lists the decoding for this field.

ii. Set up the transfer characteristics, such as:
   - Transfer width for the source in the SRC_TR_WIDTH field. Table 2-6 lists the decoding for this field.
   - Transfer width for the destination in the DST_TR_WIDTH field. Table 2-6 lists the decoding for this field.
   - Source master layer in the SMS field where the source resides.
   - Destination master layer in the DMS field where the destination resides.
   - Incrementing/decrementing or fixed address for the source in the SINC field.
   - Incrementing/decrementing or fixed address for the destination in the DINC field.

e. Write the channel configuration information into the CFGx register for channel *x*.

i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

   This step requires programming the HS_SEL_SRC/HS_SEL_DST bits, respectively. Writing a 0 activates the hardware handshaking interface to handle source/destination requests. Writing a 1 activates the software handshaking interface to handle source and destination requests.

ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign a handshaking interface to the source and destination peripheral; this requires programming the SRC_PER and DEST_PER bits, respectively.

f. If gather is enabled (parameter DMAH_CHx_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRx register for channel *x*.

g. If scatter is enabled (parameter DMAH_CHx_DST_SCA_EN = True and CTLx.DST_SCATTER_EN), program the DSRxregister for channel *x*.

4. Ensure that bit 0 of the DmaCfgReg register is enabled before writing to ChEnReg.

5. Source and destination request single and burst DMA transactions in order to transfer the block of data (assuming non-memory peripherals). The DW_ahb_dmac acknowledges at the completion of every transaction (burst and single) in the block and carries out the block transfer.

6. Once the transfer completes, hardware sets the interrupts and disables the channel. At this time, you can respond to either the Block Complete or Transfer Complete interrupts, or poll for the transfer complete raw interrupt status register (RawTfr[*n*], *n* = channel number) until it is set by hardware, in order to detect when the transfer is complete. Note that if this polling is used, the software must ensure that the transfer complete interrupt is cleared by writing to the Interrupt Clear register, ClearTfr[n], before the channel is enabled.

6.6.1.2 Multi-Block Transfer with Linked List for Source and Linked List for Destination (Row 10)

Note

This type of multi-block transfer can only be enabled when either of the following parameters is set:

- DMAH_CHx_MULTI_BLK_TYPE = NO_HARDCODE

or

- DMAH_CHx_MULTI_BLK_TYPE = LLP_LLP

1. Read the Channel Enable register (see "ChEnReg") to choose a free (disabled) channel.
2. Set up the chain of Linked List Items (otherwise known as block descriptors) in memory. Write the control information in the LLI.CTLx register location of the block descriptor for each LLI in memory (see Figure 6-1) for channel *x*. For example, in the register, you can program the following:
   a. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control device by programming the TT_FC of the CTLx register. Table 2-1 lists the decoding for this field.
   b. Set up the transfer characteristics, such as:
      i. Transfer width for the source in the SRC_TR_WIDTH field. Table 2-6 lists the decoding for this field.
      ii. Transfer width for the destination in the DST_TR_WIDTH field. Table 2-6 lists the decoding for this field.
      iii. Source master layer in the SMS field where the source resides.
      iv. Destination master layer in the DMS field where the destination resides.
      v. Incrementing/decrementing or fixed address for the source in the SINC field.
      vi. Incrementing/decrementing or fixed address for the destination in the DINC field.
3. Write the channel configuration information into the CFGx register for channel *x*.
      i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

         This step requires programming the HS_SEL_SRC/HS_SEL_DST bits, respectively. Writing a 0 activates the hardware handshaking interface to handle source/destination requests for the specific channel. Writing a 1 activates the software handshaking interface to handle source/destination requests.
      ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripheral. This requires programming the SRC_PER and DEST_PER bits, respectively.
4. Make sure that the LLI.CTL*x* register locations of all LLI entries in memory (except the last) are set as shown in Row 10 of Table 6-1. The LLI.CTL*x* register of the last Linked List Item must be set as described in Row 1 or Row 5 of Table 6-1. Figure 6-1 shows a Linked List example with two list items.
5. Make sure that the LLI.LLP*x* register locations of all LLI entries in memory (except the last) are non-zero and point to the base address of the next Linked List Item.
6. Make sure that the LLI.SAR*x*/LLI.DAR*x* register locations of all LLI entries in memory point to the start source/destination block address preceding that LLI fetch.

7. If parameter DMAH_CHx_CTL_WB_EN = True, ensure that the LLI.CTL*x*.DONE field of the LLI.CTL*x* register locations of all LLI entries in memory is cleared.
8. If source status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled), program the SSTATARx register so that the source status information can be fetched from the location pointed to by the SSTATARx. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.
9. If destination status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled), program the DSTATARx register so that the destination status information can be fetched from the location pointed to by the DSTATARx register. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Table 6-1.
10. If gather is enabled (DMAH_CH*x*_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRx register for channel *x*.
11. If scatter is enabled (DMAH_CH*x*_DST_SCA_EN = True and CTLx.DST_SCATTER_EN is enabled) program the DSRx register for channel *x*.
12. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
13. Program the CTL*x* and CFG*x* registers according to Row 10, as shown in Table 6-1
14. Program the LLP*x* register with LLP(0), the pointer to the first linked list item.
15. Finally, enable the channel by writing a 1 to the ChEnReg.*CH_EN* bit; the transfer is performed.
16. The DW_ahb_dmac fetches the first LLI from the location pointed to by LLP*x*(0).

Note The LLI.SAR*x*, LLI.DAR*x*, LLI.LLP*x*, and LLI.CTL*x* registers are fetched. The DW_ahb_dmac automatically reprograms the SAR*x*, DAR*x*, LLP*x*, and CTL*x* channel registers from the LLP*x*(0).

17. Source and destination request single and burst DMA transactions to transfer the block of data (assuming non-memory peripheral). The DW_ahb_dmac acknowledges at the completion of every transaction (burst and single) in the block and carries out the block transfer.
18. Once the block of data is transferred, the source status information is fetched from the location pointed to by the SSTATARxregister and stored in the SSTATxregister if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.

    The destination status information is fetched from the location pointed to by the DSTATARxregister and stored in the DSTATxregister if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Table 6-1.

19. If DMAH_CHx_CTL_WB_EN = True, then the CTLx[63:32] register is written out to system memory. For conditions under which the CTLx[63:32] register is written out to system memory, refer to the Write Back column of Table 6-1.

    The CTLx[63:32] register is written out to the same location on the same layer (LLPx.LMS) where it was originally fetched; that is, the location of the CTLxregister of the linked list item fetched prior to the start of the block transfer. Only the second word of the CTL*x* register is written out – CTL*x*[63:32] – because only the CTL*x*.BLOCK_TS and CTL*x*.DONE fields have been updated by the DW_ahb_dmac hardware. Additionally, the CTLx.DONE bit is asserted to indicate block completion. Therefore, software can poll the LLI.CTL*x*.DONE bit of the CTL*x* register in the LLI to ascertain when a block transfer has completed.

Note Do not poll the CTL*x*.DONE bit in the DW_ahb_dmac memory map; instead, poll the LLI.CTL*x*.DONE bit in the LLI for that block. If the polled LLI.CTL*x*.DONE bit is asserted, then this block transfer has completed. This LLI.CTL*x*.DONE bit is cleared at the start of the transfer (Step 7).

20. The SSTATx register is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. It is written to the SSTAT*x* register location of the LLI pointed to by the previously saved LLPx.LOC register.

    The DSTATx register is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. It is written to the DSTAT*x* register location of the LLI pointed to by the previously saved LLP*x*.LOC register.

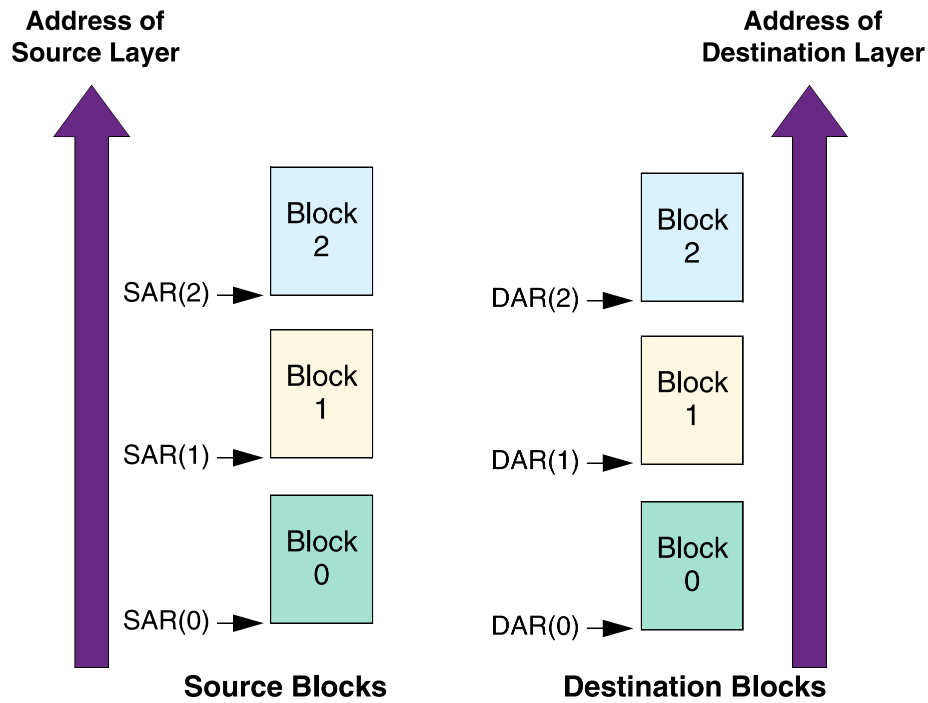
    The end-of-block interrupt, int_block, is generated after the write-back of the control and status registers has completed.

Note The write-back location for the control and status registers is the LLI pointed to by the previous value of the LLPx.LOC register, not the LLI pointed to by the current value of the LLP*x*.LOC register.

21. The DW_ahb_dmac does not wait for the block interrupt to be cleared, but continues fetching the next LLI from the memory location pointed to by the current LLPxregister and automatically reprograms the SAR*x*, DAR*x*, LLP*x*, and CTL*x* channel registers. The DMA transfer continues until the DW_ahb_dmac determines that the CTLx and LLPxregisters at the end of a block transfer match the ones described in Row 1 or Row 5 of Table 6-1 (as discussed earlier). The DW_ahb_dmac then knows that the previously transferred block was the last block in the DMA transfer.

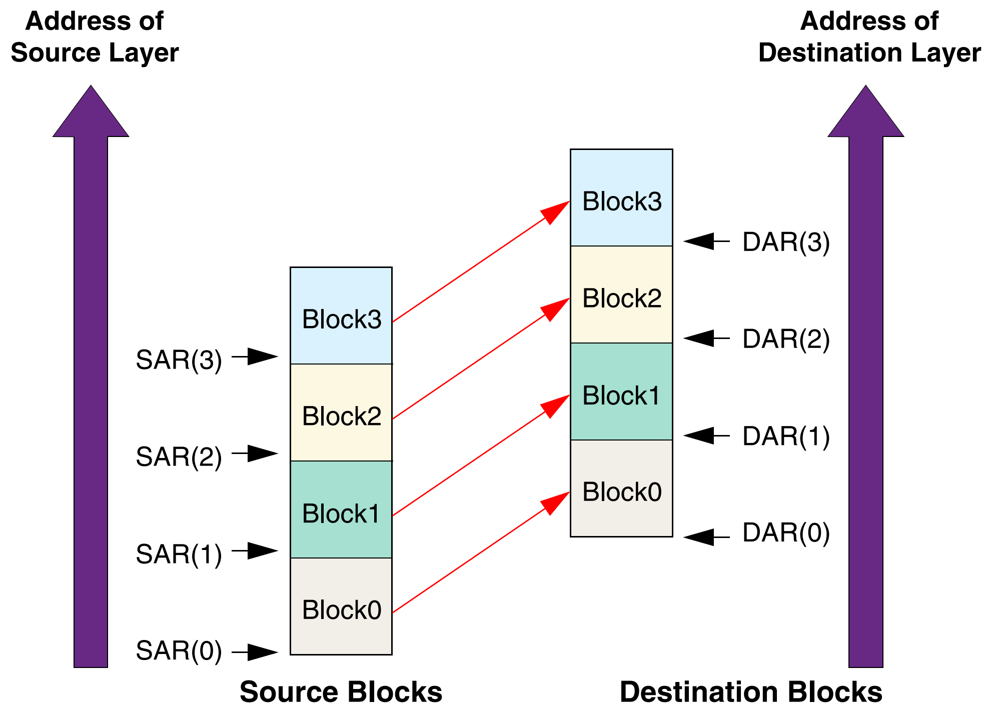
The DMA transfer might look like that shown in Figure 6-6.

Figure 6-6 Multi-Block with Linked Address for Source and Destination



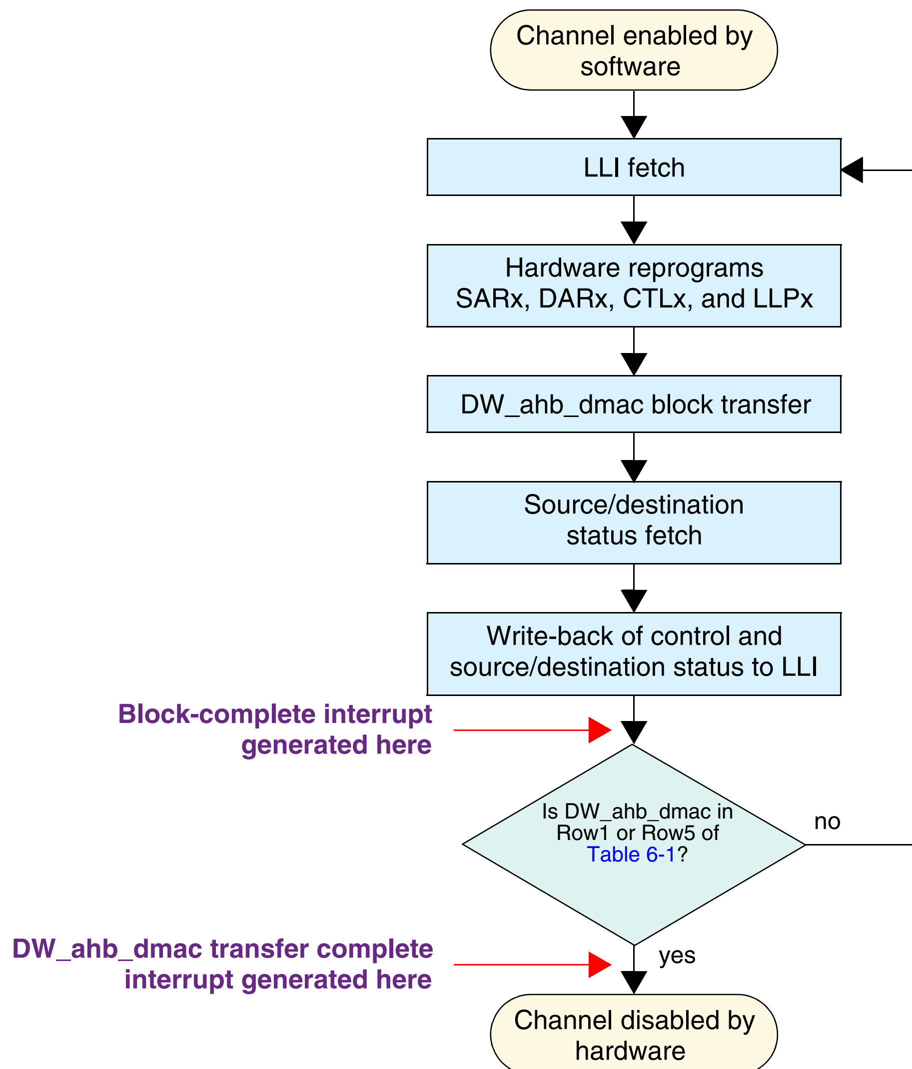
If you need to execute a DMA transfer where the source and destination address are contiguous, but where the amount of data to be transferred is greater than the maximum block size CTL*x*.BLOCK_TS, then this can be achieved using the type of multi-block transfer shown in Figure 6-7.

Figure 6-7 Multi-Block with Linked Address for Source and Destination Where SARx and DARx Between Successive Blocks are Contiguous



The DMA transfer flow is shown in Figure 6-8.

Figure 6-8 DMA Transfer Flow for Source and Destination Linked List Address



6.6.1.3 Multi-Block Transfer with Source Address Auto-Reloaded and Destination Address Auto-Reloaded (Row 4)

Note This type of multi-block transfer can only be enabled when either of the following parameters is set:

- DMAH_CHx_MULTI_BLK_TYPE = NO_HARDCODE

or

- DMAH_CHx_MULTI_BLK_TYPE = RELOAD_RELOAD

1. Read the Channel Enable register (see "ChEnReg") to choose an available (disabled) channel.

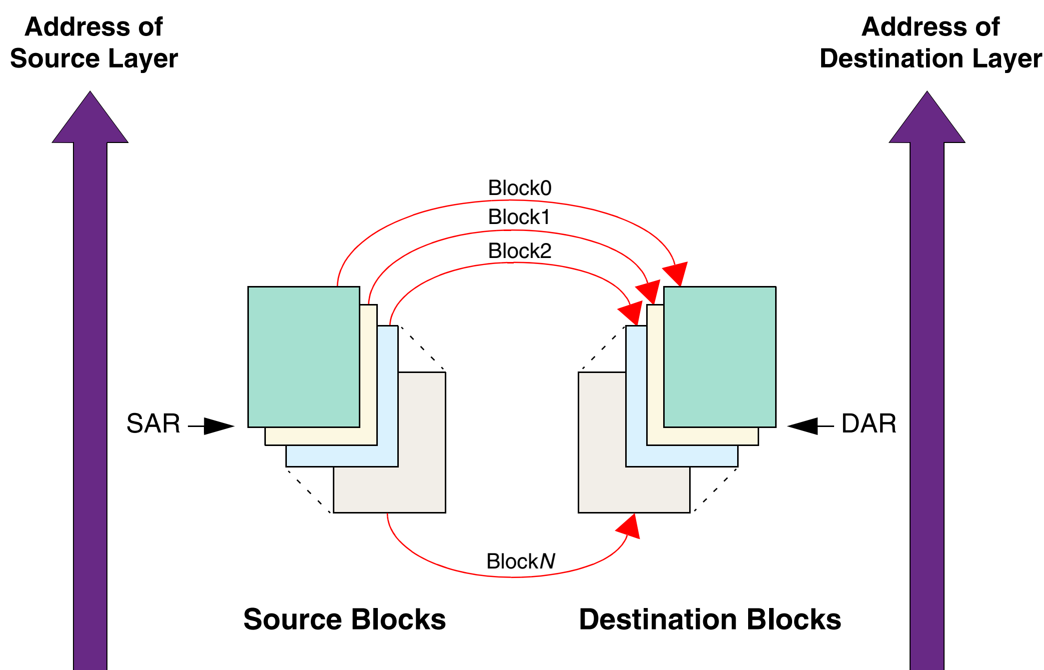
2. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
3. Program the following channel registers:
   a. Write the starting source address in the SARx register for channel *x*.
   b. Write the starting destination address in the DARx register for channel *x*.
   c. Program CTL*x* and CFG*x* according to Row 4, as shown in Table 6-1. Program the LLPx register with 0.
   d. Write the control information for the DMA transfer in the CTLx register for channel *x*. For example, in the register, you can program the following:
      i. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control device by programming the TT_FC of the CTLx register. Table 2-1 lists the decoding for this field.
      ii. Set up the transfer characteristics, such as:
         - Transfer width for the source in the SRC_TR_WIDTH field; Table 2-6 lists the decoding for this field.
         - Transfer width for the destination in the DST_TR_WIDTH field; Table 2-6 lists the decoding for this field.
         - Source master layer in the SMS field where the source resides.
         - Destination master layer in the DMS field where the destination resides.
         - Incrementing/decrementing or fixed address for the source in the SINC field.
         - Incrementing/decrementing or fixed address for the destination in the DINC field.
   e. If gather is enabled (DMAH_CH*x*_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRx register for channel *x*.
   f. If scatter is enabled (DMAH_CH*x*_DST_SCA_EN = True and CTLx.DST_SCATTER_EN), program the DSRx register for channel *x*.
   g. Write the channel configuration information into the CFGx register for channel *x*. Ensure that the reload bits, CFG*x*. RELOAD_SRC and CFG*x*.RELOAD_DST, are enabled.
      i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

         This step requires programming the HS_SEL_SRC/HS_SEL_DST bits, respectively. Writing a 0 activates the hardware handshaking interface to handle source/destination requests for the specific channel. Writing a 1 activates the software handshaking interface to handle source/destination requests.
      ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripheral. This requires programming the SRC_PER and DEST_PER bits, respectively.
4. Ensure that bit 0 of the DmaCfgReg register is enabled before writing to ChEnReg.
5. Source and destination request single and burst DW_ahb_dmac transactions to transfer the block of data (assuming non-memory peripherals). The DW_ahb_dmac acknowledges on completion of each burst/single transaction and carries out the block transfer.

6. When the block transfer has completed, the DW_ahb_dmac reloads the SAR*x*, DAR*x*, and CTL*x* registers. Hardware sets the block-complete interrupt. The DW_ahb_dmac then samples the row number, as shown in Table 6-1. If the DW_ahb_dmac is in Row 1, then the DMA transfer has completed. Hardware sets the transfer complete interrupt and disables the channel. You can either respond to the Block Complete or Transfer Complete interrupts, or poll for the transfer complete raw interrupt status register (RawTfr[*n*], where *n* is the channel number) until it is set by hardware, in order to detect when the transfer is complete. Note that if this polling is used, software must ensure that the transfer complete interrupt is cleared by writing to the Interrupt Clear register, ClearTfr[*n*], before the channel is enabled. If the DW_ahb_dmac is not in Row 1, the next step is performed.
7. The DMA transfer proceeds as follows:
   a. If interrupts are enabled (CTL*x*x.INT_EN = 1) and the block-complete interrupt is unmasked (MaskBlock[*x*] = 1’b1, where *x* is the channel number), hardware sets the block-complete interrupt when the block transfer has completed. It then stalls until the block-complete interrupt is cleared by software. If the next block is to be the last block in the DMA transfer, then the block-complete ISR (interrupt service routine) should clear the reload bits in the CFG*x*.RELOAD_SRC and CFG*x*.RELOAD_DST registers. This puts the DW_ahb_dmac into Row 1, as shown in Table 6-1. If the next block is not the last block in the DMA transfer, then the reload bits should remain enabled to keep the DW_ahb_dmac in Row 4.
   b. If interrupts are disabled (CTL*x*.INT_EN = 0) or the block-complete interrupt is masked (MaskBlock[*x*] = 1’b0, where *x* is the channel number), then hardware does not stall until it detects a write to the block-complete interrupt clear register; instead, it immediately starts the next block transfer. In this case, software must clear the reload bits in the CFG*x*.RELOAD_SRC and CFG*x*.RELOAD_DST registers to put the DW_ahb_dmac into Row 1 of Table 6-1 before the last block of the DMA transfer has completed.

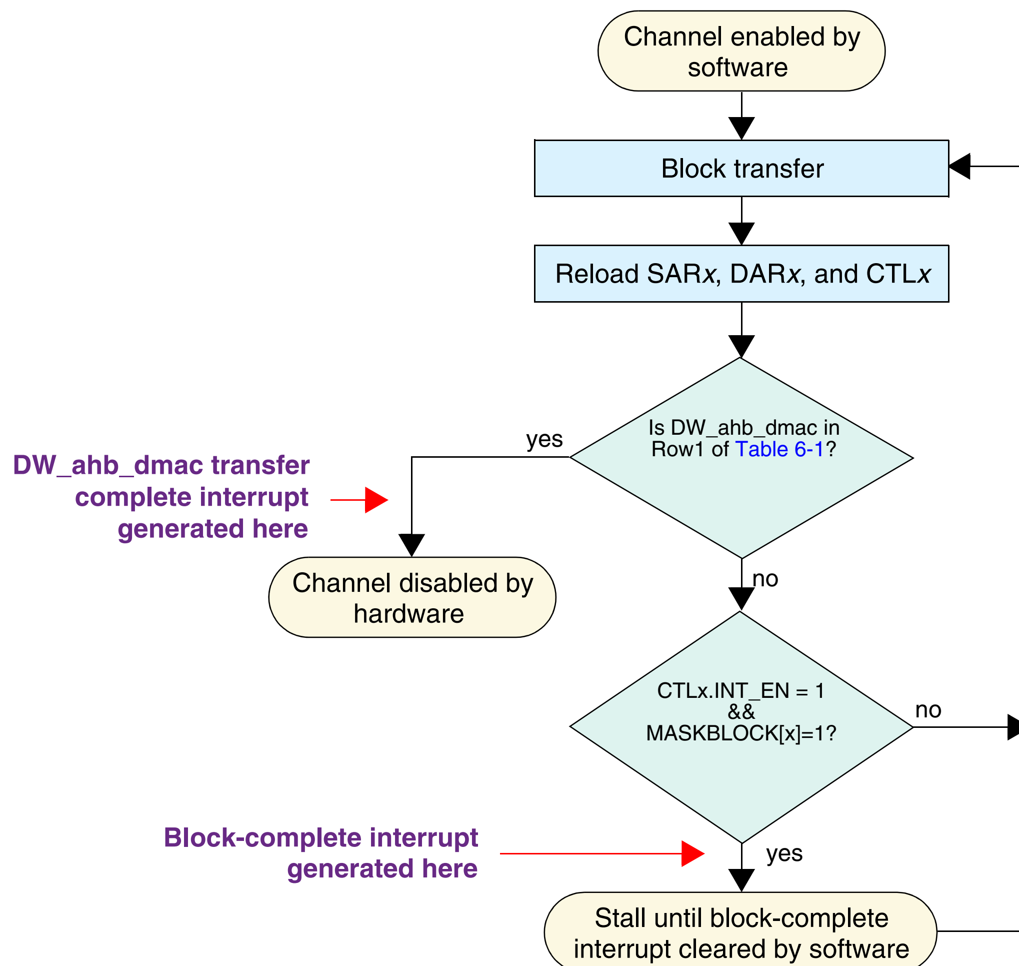
The transfer is similar to that shown in Figure 6-9.

Figure 6-9 Multi-Block DMA Transfer with Source and Destination Address Auto-Reloaded



The DMA transfer flow is shown in Figure 6-10.

Figure 6-10 DMA Transfer Flow for Source and Destination Address Auto-Reloaded



6.6.1.4 Multi-Block Transfer with Source Address Auto-Reloaded and Linked List Destination Address (Row 7)

This type of multi-block transfer can only be enabled when either of the following parameters is set:

- DMAH_CHx_MULTI_BLK_TYPE = 0

or

- DMAH_CHx_MULTI_BLK_TYPE = RELOAD_LLP

1. Read the Channel Enable register (see "ChEnReg") in order to choose a free (disabled) channel.
2. Set up the chain of linked list items (otherwise known as block descriptors) in memory. Write the control information in the LLI.CTLx register location of the block descriptor for each LLI in memory (see Figure 6-1) for channel *x*. For example, in the register you can program the following:

a. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control peripheral by programming the TT_FC of the CTLxregister. Table 2-1 lists the decoding for this field.

b. Set up the transfer characteristics, such as:

   i. Transfer width for the source in the SRC_TR_WIDTH field. Table 2-6 lists the decoding for this field.

   ii. Transfer width for the destination in the DST_TR_WIDTH field. Table 2-6 lists the decoding for this field.

   iii. Source master layer in the SMS field where the source resides.

   iv. Destination master layer in the DMS field where the destination resides.

   v. Incrementing/decrementing or fixed address for the source in the SINC field.

   vi. Incrementing/decrementing or fixed address for the destination in the DINC field.

3. Write the starting source address in the SARxregister for channel *x*.

Note The values in the LLI.SARx register locations of each of the Linked List Items (LLIs) set up in memory, although fetched during an LLI fetch, are not used.

4. Write the channel configuration information into the CFGxregister for channel *x*.

   i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

   This step requires programming the HS_SEL_SRC/HS_SEL_DST bits. Writing a 0 activates the hardware handshaking interface to handle source/destination requests for the specific channel. Writing a 1 activates the software handshaking interface source/destination requests.

   ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripheral; this requires programming the SRC_PER and DEST_PER bits, respectively.

5. Make sure that the LLI.CTL*x* register locations of all LLIs in memory (except the last) are set as shown in Row 7 of Table 6-1, while the LLI.CTL*x* register of the last Linked List item must be set as described in Row 1 or Row 5 of Table 6-1. Figure 6-1 shows a Linked List example with two list items.

6. Ensure that the LLI.LLP*x* register locations of all LLIs in memory (except the last) are non-zero and point to the next Linked List Item.

7. Ensure that the LLI.DAR*x* register location of all LLIs in memory point to the start destination block address preceding that LLI fetch.

8. If DMAH_CHx_CTL_WB_EN = True, ensure that the LLI.CTL*x*.DONE fields of the LLI.CTL*x* register locations of all LLIs in memory are cleared.

9. If source status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled), program the SSTATARxregister so that the source status information can be fetched from the location pointed to by the SSTATARx. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.

10. If destination status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled), program the DSTATARxregister so that the destination status information can be fetched from the location pointed to by the DSTATARxregister. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Table 6-1.
11. If gather is enabled (DMAH_CH*x*_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRxregister for channel *x*.
12. If scatter is enabled (DMAH_CH*x*_DST_SCA_EN = True and CTLx.DST_SCATTER_EN, program the DSRx register for channel *x*.
13. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
14. Program the CTLx and CFGxregisters according to Row 7, as shown in Table 6-1.
15. Program the LLPxregister with LLP*x*(0), the pointer to the first Linked List item.
16. Ensure that bit 0 of the DmaCfgRegregister is enabled before writing to ChEnReg.
17. The DW_ahb_dmac fetches the first LLI from the location pointed to by LLP*x*(0).

Note The LLI.SAR*x*, LLI.DAR*x*, LLI.LLP*x*, and LLI.CTL*x* registers are fetched. The LLI.SAR*x* register – although fetched – is not used.

18. Source and destination request single and burst DW_ahb_dmac transactions in order to transfer the block of data (assuming non-memory peripherals). The DW_ahb_dmac acknowledges at the completion of every transaction (burst and single) in the block and carries out the block transfer.
19. Once the block of data is transferred, the source status information is fetched from the location pointed to by the SSTATARx register and stored in the SSTATx register if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.

    The destination status information is fetched from the location pointed to by the DSTATARxregister and stored in the DSTATxregister if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Table 6-1.
20. If DMAH_CHx_CTL_WB_EN = True, then the CTLx[63:32] register is written out to system memory. For conditions under which the CTLx[63:32] register is written out to system memory, refer to the Write Back column of Table 6-1.

    The CTLx[63:32] register is written out to the same location on the same layer (LLPx.LMS) where it was originally fetched; that is, the location of the CTLxregister of the linked list item fetched prior to the start of the block transfer. Only the second word of the CTLxregister is written out – CTL*x*[63:32] – because only the CTL*x*.BLOCK_TS and CTL*x*.DONE fields have been updated by hardware within the DW_ahb_dmac. The LLI.CTLx.DONE bit is asserted to indicate block completion. Therefore,

software can poll the LLI.CTL*x*.DONE bit field of the CTL*x* register in the LLI to ascertain when a block transfer has completed.

Note Do not poll the CTL*x*.DONE bit in the DW_ahb_dmac memory map. Instead, poll the LLI.CTL*x*.DONE bit in the LLI for that block. If the polled LLI.CTL*x*.DONE bit is asserted, then this block transfer has completed. This LLI.CTL*x*.DONE bit is cleared at the start of the transfer (Step 8).

21. The SSTATxregister is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. It is written to the SSTAT*x* register location of the LLI pointed to by the previously saved LLP*x*.LOC register.

    The DSTATxregister is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. It is written to the DSTAT*x* register location of the LLI pointed to by the previously saved LLP*x*.LOC register.

    The end-of-block interrupt, int_block, is generated after the write-back of the control and status registers has completed.

Note The write-back location for the control and status registers is the LLI pointed to by the previous value of the LLP*x*.LOC register, not the LLI pointed to by the current value of the LLP*x*.LOC register.

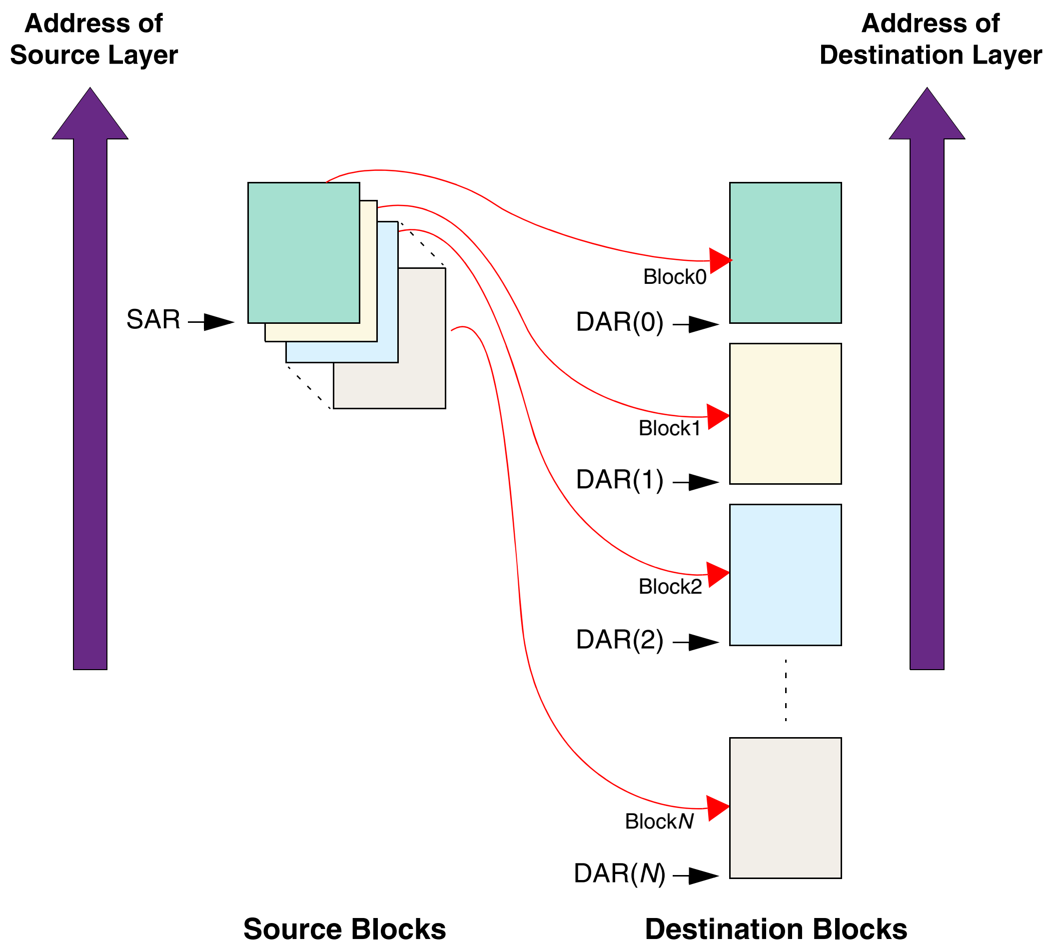
22. The DW_ahb_dmac reloads the SAR*x* register from the initial value. Hardware sets the block-complete interrupt. The DW_ahb_dmac samples the row number, as shown in Table 6-1. If the DW_ahb_dmac is in Row 1 or Row 5, then the DMA transfer has completed. Hardware sets the transfer complete interrupt and disables the channel. You can either respond to the Block Complete or Transfer Complete interrupts, or poll for the transfer complete raw interrupt status register (RawTfr[*n*], *n* = channel number) until it is set by hardware, in order to detect when the transfer is complete. Note that if this polling is used, software must ensure that the transfer complete interrupt is cleared by writing to the Interrupt Clear register, ClearTfr[*n*], before the channel is enabled. If the DW_ahb_dmac is not in Row 1 or Row 5 as shown in Table 6-1, the following steps are performed.
23. The DMA transfer proceeds as follows:
    a. If interrupts are enabled (CTL*x*.INT_EN = 1) and the block-complete interrupt is unmasked (MaskBlock[*x*] = 1'b1, where *x* is the channel number), hardware sets the block-complete interrupt when the block transfer has completed. It then stalls until the block-complete interrupt is cleared by software. If the next block is to be the last block in the DMA transfer, then the block-complete ISR (interrupt service routine) should clear the CFG*x*.RELOAD_SRC source reload bit. This puts the DW_ahb_dmac into Row 1, as shown in Table 6-1. If the next block is not the last block in the DMA transfer, then the source reload bit should remain enabled to keep the DW_ahb_dmac in Row 7, as shown in Table 6-1.
    b. If interrupts are disabled (CTL*x*.INT_EN = 0) or the block-complete interrupt is masked (MaskBlock[*x*] = 1'b0, where *x* is the channel number), then hardware does not stall until it detects a write to the block-complete interrupt clear register; instead, it immediately starts the next block transfer. In this case, software must clear the source reload bit, CFG*x*.RELOAD_SRC in

order to put the device into Row 1 of Table 6-1 before the last block of the DMA transfer has completed.

24. The DW_ahb_dmac fetches the next LLI from memory location pointed to by the current LLPx register and automatically reprograms the DARx, CTLx, and LLPx channel registers. Note that the SAR*x* is not reprogrammed, since the reloaded value is used for the next DMA block transfer. If the next block is the last block of the DMA transfer, then the CTL*x* and LLP*x* registers just fetched from the LLI should match Row 1 or Row 5 of Table 6-1.

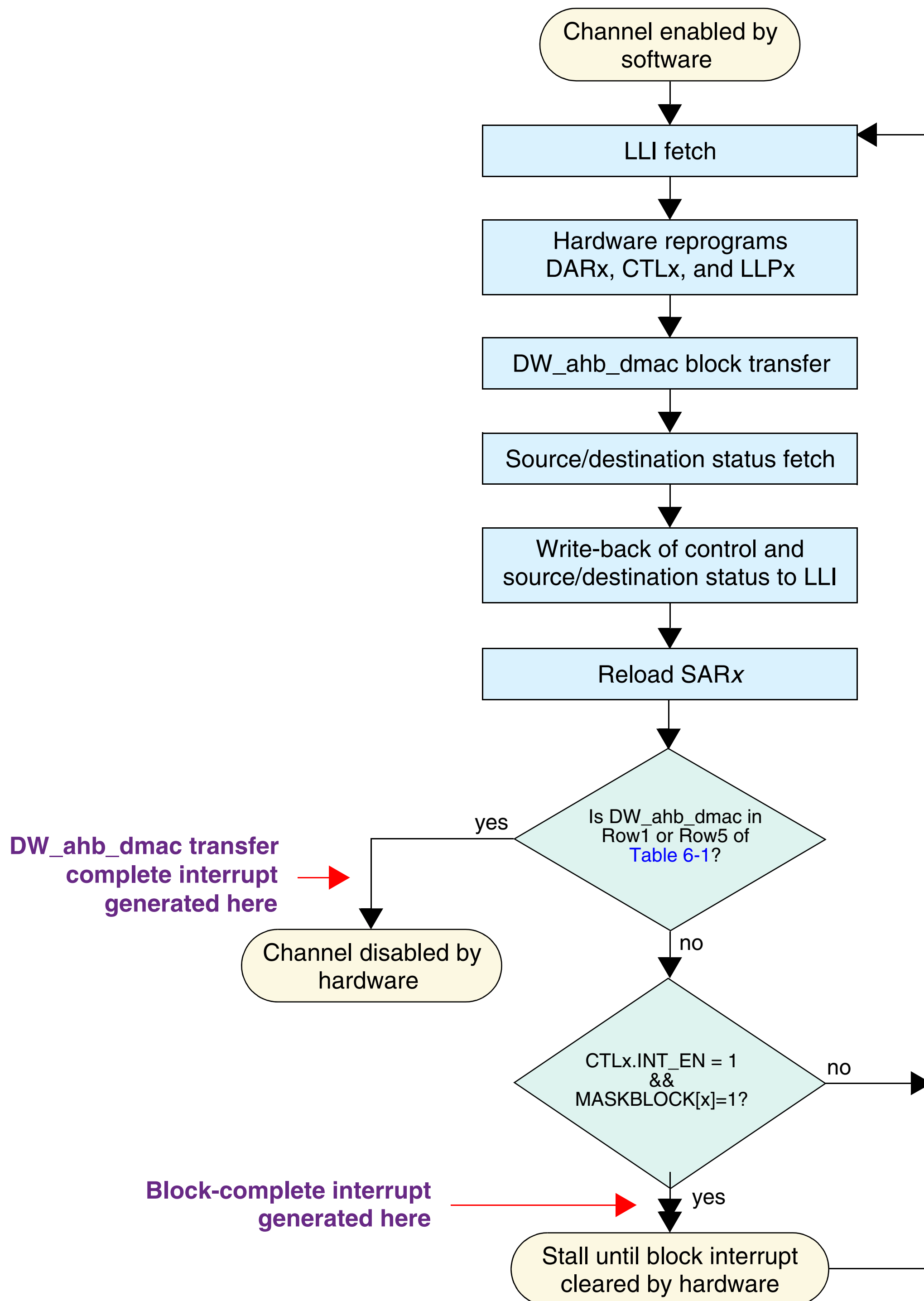
The DMA transfer might look like that shown in Figure 6-11.

Figure 6-11 Multi-Block DMA Transfer with Source Address Auto-Reloaded and Linked List Destination Address



The DMA transfer flow is shown in Figure 6-12.

Figure 6-12 DMA Transfer Flow for Source Address Auto-Reloaded and Linked List Destination Address



6.6.1.5 Multi-Block Transfer with Source Address Auto-Reloaded and Contiguous Destination

Address (Row 3)

Note

This type of multi-block transfer can only be enabled when either of the following parameters is set:

- DMAH_CHx_MULTI_BLK_TYPE = 0

or

- DMAH_CHx_MULTI_BLK_TYPE = RELOAD_CONT

1. Read the Channel Enable register (see "ChEnReg") to choose a free (disabled) channel.
2. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
3. Program the following channel registers:
   a. Write the starting source address in the SARxregister for channel *x*.
   b. Write the starting destination address in the DARxregister for channel *x*.
   c. Program CTL*x* and CFG*x* according to Row 3, shown in Table 6-1. Program the LLPx register with 0.
   d. Write the control information for the DMA transfer in the CTLx register for channel *x*. For example, in the register, you can program the following:
      i. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control device by programming the TT_FC of the CTL*x* register. Table 2-1 lists the decoding for this field.
      ii. Set up the transfer characteristics, such as:
         - Transfer width for the source in the SRC_TR_WIDTH field. Table 2-6 lists the decoding for this field.
         - Transfer width for the destination in the DST_TR_WIDTH field. Table 2-6 lists the decoding for this field.
         - Source master layer in the SMS field where the source resides.
         - Destination master layer in the DMS field where the destination resides.
         - Incrementing/decrementing or fixed address for the source in the SINC field.
         - Incrementing/decrementing or fixed address for the destination in the DINC field.
   e. If gather is enabled (DMAH_CH*x*_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRx register for channel *x*.
   f. If scatter is enabled (DMAH_CH*x*_DST_SCA_EN = True and CTLx.DST_SCATTER_EN is enabled), program the DSRx register for channel *x*.
   g. Write the channel configuration information into the CFGxregister for channel *x*.
      i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

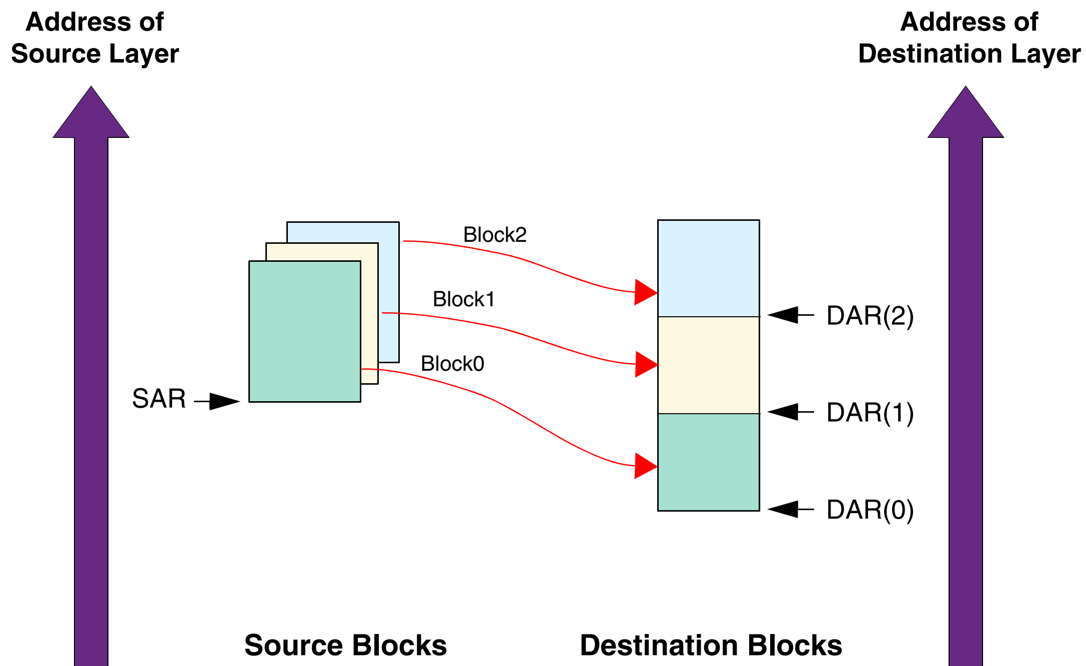
This step requires programming the HS_SEL_SRC/HS_SEL_DST bits, respectively. Writing a 0 activates the hardware handshaking interface to handle source/destination requests for the specific channel. Writing a 1 activates the software handshaking interface to handle source/destination requests.

ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripheral. This requires programming the SRC_PER and DEST_PER bits, respectively.

4. Ensure that bit 0 of the DmaCfgReg register is enabled before writing to ChEnReg.
5. Source and destination request single and burst DW_ahb_dmac transactions to transfer the block of data (assuming non-memory peripherals). The DW_ahb_dmac acknowledges at the completion of every transaction (burst and single) in the block and carries out the block transfer.
6. When the block transfer has completed, the DW_ahb_dmac reloads the SAR*x* register; the DAR*x* register remains unchanged. Hardware sets the block-complete interrupt. The DW_ahb_dmac then samples the row number, as shown in Table 6-1. If the DW_ahb_dmac is in Row 1, then the DMA transfer has completed. Hardware sets the transfer-complete interrupt and disables the channel. You can either respond to the Block Complete or Transfer Complete interrupts, or poll for the transfer complete raw interrupt status register (RawTfr[*n*], *n* = channel number) until it is set by hardware, in order to detect when the transfer is complete. Note that if this polling is used, software must ensure that the transfer complete interrupt is cleared by writing to the Interrupt Clear register, ClearTfr[*n*], before the channel is enabled. If the DW_ahb_dmac is not in Row 1, the next step is performed.
7. The DMA transfer proceeds as follows:
   a. If interrupts are enabled (CTL*x*.INT_EN = 1) and the block-complete interrupt is unmasked (MaskBlock[*x*] = 1’b1, where *x* is the channel number), hardware sets the block-complete interrupt when the block transfer has completed. It then stalls until the block-complete interrupt is cleared by software. If the next block is to be the last block in the DMA transfer, then the block-complete ISR (interrupt service routine) should clear the source reload bit, CFG*x*.RELOAD_SRC. This puts the DW_ahb_dmac into Row 1, as shown in Table 6-1. If the next block is not the last block in the DMA transfer, then the source reload bit should remain enabled to keep the DW_ahb_dmac in Row 3, as shown in Table 6-1.
   b. If interrupts are disabled (CTL*x*.INT_EN = 0) or the block-complete interrupt is masked (MaskBlock[*x*] = 1’b0, where *x* is the channel number), then hardware does not stall until it detects a write to the block-complete interrupt clear register; instead, it starts the next block transfer immediately. In this case, software must clear the source reload bit, CFG*x*.RELOAD_SRC, to put the device into Row 1 of Table 6-1 before the last block of the DMA transfer has completed.

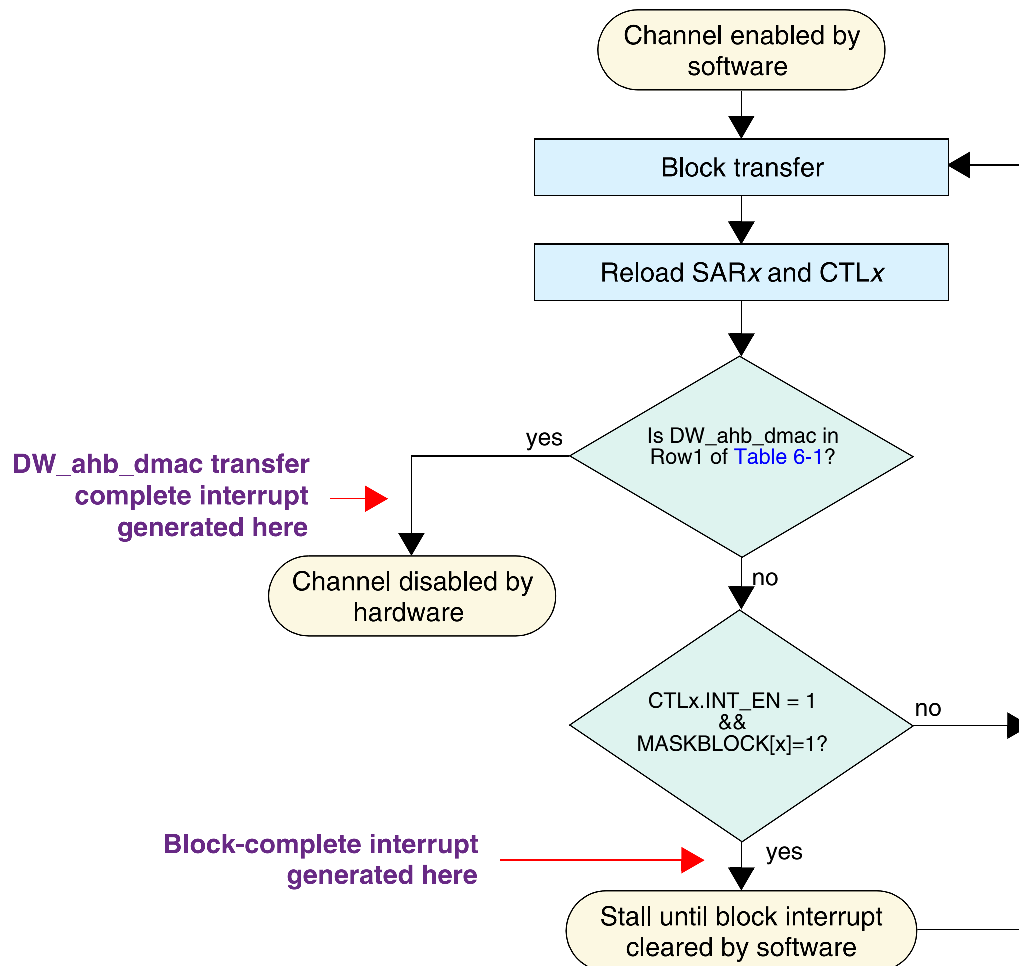
The transfer is similar to that shown in Figure 6-13.

Figure 6-13 Multi-Block DMA Transfer with Source Address Auto-Reloaded and Contiguous Destination Address



The DMA transfer flow is shown in Figure 6-14.

Figure 6-14 DMA Transfer Flow for Source Address Auto-Reloaded and Contiguous Destination Address



6.6.1.6 Multi-Block DMA Transfer with Linked List for Source and Contiguous Destination Address (Row 8)

Note

This type of multi-block transfer can only be enabled when either of the following parameters is set:

- DMAH_CHx_MULTI_BLK_TYPE = 0

or

- DMAH_CHx_MULTI_BLK_TYPE = LLP_CONT

1. Read the Channel Enable register (see "ChEnReg") to choose a free (disabled) channel.
2. Set up the linked list in memory. Write the control information in the LLI.CTLx register location of the block descriptor for each LLI in memory (see Figure 6-1) for channel *x*. For example, in the register, you can program the following:

a. Set up the transfer type (memory or non-memory peripheral for source and destination) and flow control device by programming the TT_FC of the CTLx register. Table 2-1 lists the decoding for this field.

b. Set up the transfer characteristics, such as:

   i. Transfer width for the source in the SRC_TR_WIDTH field. Table 2-6 lists the decoding for this field.

   ii. Transfer width for the destination in the DST_TR_WIDTH field. Table 2-6 lists the decoding for this field.

   iii. Source master layer in the SMS field where the source resides.

   iv. Destination master layer in the DMS field where the destination resides.

   v. Incrementing/decrementing or fixed address for the source in the SINC field.

   vi. Incrementing/decrementing or fixed address for the destination in the DINC field.

3. Write the starting destination address in the DARx register for channel *x*.

Note The values in the LLI.DAR*x* register location of each Linked List Item (LLI) in memory, although fetched during an LLI fetch, are not used.

4. Write the channel configuration information into the CFGx register for channel *x*.

   i. Designate the handshaking interface type (hardware or software) for the source and destination peripherals; this is not required for memory.

   This step requires programming the HS_SEL_SRC/HS_SEL_DST bits. Writing a 0 activates the hardware handshaking interface to handle source/destination requests for the specific channel. Writing a 1 activates the software handshaking interface to handle source/destination requests.

   ii. If the hardware handshaking interface is activated for the source or destination peripheral, assign the handshaking interface to the source and destination peripherals. This requires programming the SRC_PER and DEST_PER bits, respectively.

5. Ensure that all LLI.CTL*x* register locations of the LLI (except the last) are set as shown in Row 8 of Table 6-1, while the LLI.CTL*x* register of the last Linked List item must be set as described in Row 1 or Row 5 of Table 6-1. Figure 6-1 shows a Linked List example with two list items.

6. Ensure that the LLI.LLP*x* register locations of all LLIs in memory (except the last) are non-zero and point to the next Linked List Item.

7. Ensure that the LLI.SAR*x* register location of all LLIs in memory point to the start source block address preceding that LLI fetch.

8. If DMAH_CHx_CTL_WB_EN = True, ensure that the LLI.CTL*x*.DONE fields of the LLI.CTL*x* register locations of all LLIs in memory are cleared.

9. If source status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled), program the SSTATARx register so that the source status information can be fetched from the location pointed to by SSTATARx. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.

10. If destination status fetching is enabled (DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled), program the DSTATARx register so that the destination status information can be fetched from the location pointed to by the DSTATARx register. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Table 6-1.
11. If gather is enabled (DMAH_CH*x*_SRC_GAT_EN = True and CTLx.SRC_GATHER_EN is enabled), program the SGRx register for channel *x*.
12. If scatter is enabled (DMAH_CH*x*_DST_SCA_EN = True and CTLx.DST_SCATTER_EN) program the DSRx register for channel *x*.
13. Clear any pending interrupts on the channel from the previous DMA transfer by writing to the Interrupt Clear registers: ClearTfr, ClearBlock, ClearSrcTran, ClearDstTran, and ClearErr. Reading the Interrupt Raw Status and Interrupt Status registers confirms that all interrupts have been cleared.
14. Program the CTLx and CFGx registers according to Row 8, as shown in Table 6-1.
15. Program the LLPx register with LLP*x*(0), the pointer to the first Linked List item.
16. Ensure that bit 0 of the DmaCfgReg register is enabled before writing to ChEnReg.
17. The DW_ahb_dmac fetches the first LLI from the location pointed to by LLP*x*(0).

Note The LLI.SAR*x*, LLI.DAR*x*, LLI.LLP*x*, and LLI.CTL*x* registers are fetched. The LLI.DAR*x* register location of the LLI – although fetched – is not used. The DARx register in the DW_ahb_dmac remains unchanged.

18. Source and destination request single and burst DW_ahb_dmac transactions to transfer the block of data (assuming non-memory peripherals). The DW_ahb_dmac acknowledges at the completion of every transaction (burst and single) in the block and carries out the block transfer.
19. Once the block of data is transferred, the source status information is fetched from the location pointed to by the SSTATARx register and stored in the SSTATx register if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. For conditions under which the source status information is fetched from system memory, refer to the Write Back column of Table 6-1.

    The destination status information is fetched from the location pointed to by the DSTATARx register and stored in the DSTATx register if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. For conditions under which the destination status information is fetched from system memory, refer to the Write Back column of Figure 6-1.
20. If DMAH_CHx_CTL_WB_EN = True, then the CTLx[63:32] register is written out to system memory. For conditions under which the CTLx[63:32] register is written out to system memory, refer to the Write Back column of Table 6-1.

    The CTLx[63:32] register is written out to the same location on the same layer (LLPx.LMS) where it was originally fetched; that is, the location of the CTLx register of the linked list item fetched prior to the start of the block transfer. Only the second word of the CTLx register is written out, CTLx[63:32], because only the CTLx.BLOCK_TS and CTLx.DONE fields have been updated by hardware within the DW_ahb_dmac. Additionally, the CTLx.DONE bit is asserted to indicate block completion.

Therefore, software can poll the LLI.CTL*x*.DONE bit field of the CTL*x* register in the LLI to ascertain when a block transfer has completed.

Note

Do not poll the CTL*x*.DONE bit in the DW_ahb_dmac memory map. Instead, poll the LLI.CTL*x*.DONE bit in the LLI for that block. If the polled LLI.CTL*x*.DONE bit is asserted, then this block transfer has completed. This LLI.CTL*x*.DONE bit was cleared at the start of the transfer (Step 8).

21. The SSTATx register is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_SRC = True, and CFG*x*.SS_UPD_EN is enabled. It is written to the SSTAT*x* register location of the LLI pointed to by the previously saved LLP*x*.LOC register.

    The DSTATx register is now written out to system memory if DMAH_CHx_CTL_WB_EN = True, DMAH_CH*x*_STAT_DST = True, and CFG*x*.DS_UPD_EN is enabled. It is written to the DSTAT*x* register location of the LLI pointed to by the previously saved LLP*x*.LOC register.

    The end-of-block interrupt, int_block, is generated after the write-back of the control and status registers has completed.

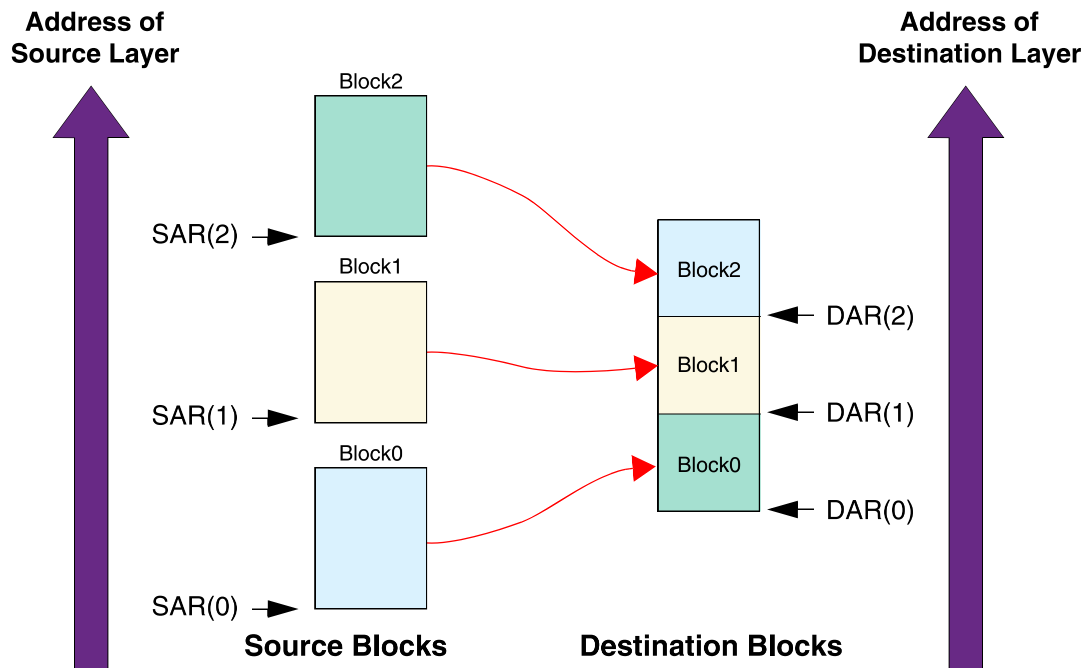
Note

The write-back location for the control and status registers is the LLI pointed to by the previous value of the LLP*x*.LOC register, not the LLI pointed to by the current value of the LLP*x*.LOC register.

22. The DW_ahb_dmac does not wait for the block interrupt to be cleared, but continues and fetches the next LLI from the memory location pointed to by the current LLP*x* register and automatically reprograms the SARx, CTLx, and LLPx channel registers. The DARx register is left unchanged. The DMA transfer continues until the DW_ahb_dmac samples that the CTL*x* and LLP*x* registers at the end of a block transfer match those described in Row 1 or Row 5 of Table 2-1 (as discussed earlier). The DW_ahb_dmac then knows that the previously transferred block was the last block in the DMA transfer.

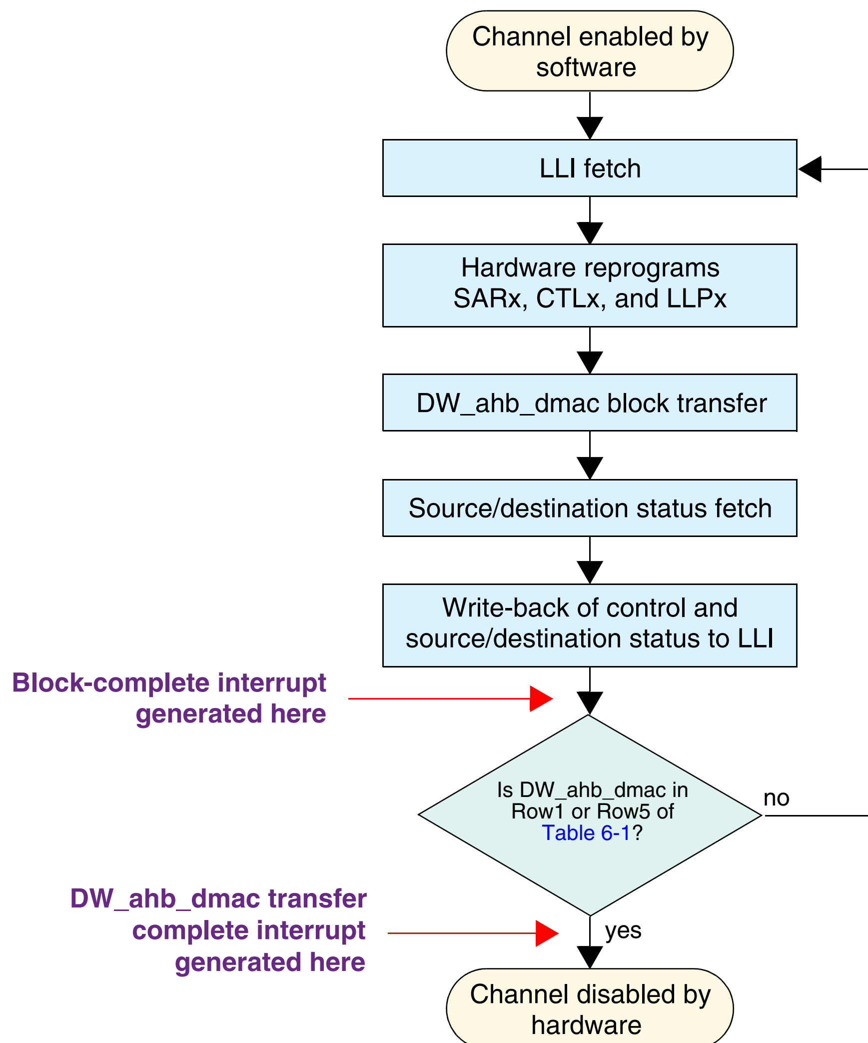
The DW_ahb_dmac transfer might look like that shown in Figure 6-15. Note that the destination address is decrementing.

Figure 6-15 Multi-Block DMA Transfer with Linked List Source Address and Contiguous Destination Address



The DMA transfer flow is shown in Figure 6-16.

Figure 6-16 DMA Transfer Flow for Source Address Auto-Reloaded and Contiguous Destination Address



6.7 Disabling a Channel Prior to Transfer Completion

Under normal operation, software enables a channel by writing a 1 to the channel enable register, ChEnReg.CH_EN, and hardware disables a channel on transfer completion by clearing the ChEnReg.CH_EN register bit.

The recommended way for software to disable a channel without losing data is to use the CH_SUSP bit in conjunction with the FIFO_EMPTY bit in the Channel Configuration Register (CFGx).

1. If software wishes to disable a channel prior to the DMA transfer completion, then it can set the CFG*x*.CH_SUSP bit to tell the DW_ahb_dmac to halt all transfers from the source peripheral. Therefore, the channel FIFO receives no new data.
2. Software can now poll the CFG*x*.FIFO_EMPTY bit until it indicates that the channel FIFO is empty.
3. The ChEnReg.CH_EN bit can then be cleared by software once the channel FIFO is empty.

When CTLx.SRC_TR_WIDTH < CTLx.DST_TR_WIDTH and the CFGx.CH_SUSP bit is high, the CFGx.FIFO_EMPTY is asserted once the contents of the FIFO do not permit a single word of CTL*x*.DST_TR_WIDTH to be formed. However, there may still be data in the channel FIFO, but not enough to form a single transfer of CTL*x*.DST_TR_WIDTH. In this scenario, once the channel is disabled, the remaining data in the channel FIFO is not transferred to the destination peripheral.

It is permissible to remove the channel from the suspension state by writing a 0 to the CFGx.CH_SUSP register. The DMA transfer completes in the normal manner.

Note If a channel is disabled by software, an active single or burst transaction is not guaranteed to receive an acknowledgment.

6.7.1 Abnormal Transfer Termination

A DW_ahb_dmac DMA transfer may be terminated abruptly by software by clearing the channel enable bit, ChEnReg.CH_EN. You must not assume that the channel is disabled immediately after the ChEnReg. The CH_EN bit is cleared over the AHB slave interface. Consider this as a request to disable the channel. You must poll ChEnReg.CH_EN and confirm that the channel is disabled by reading back 0. A case where the channel is not disabled after a channel disable request is where either the source or destination has received a split or retry response. The DW_ahb_dmac must keep re-attempting the transfer to the system HADDR that originally received the split or retry response until an OKAY response is returned; to do otherwise is an AMBA protocol violation.

Software may terminate all channels abruptly by clearing the global enable bit in the DW_ahb_dmac Configuration Register (DmaCfgReg[0]). Again, you must not assume that all channels are disabled immediately after the DmaCfgReg[0] is cleared over the AHB slave interface. Consider this as a request to disable all channels. You must poll ChEnReg and confirm that all channels are disabled by reading back 0.

Note If the channel enable bit is cleared while there is data in the channel FIFO, this data is not sent to the destination peripheral and is not present when the channel is re-enabled. For read-sensitive source peripherals, such as a source FIFO, this data is therefore lost. When the source is not a read-sensitive device (such as memory), disabling a channel without waiting for the channel FIFO to empty may be acceptable, since the data is available from the source peripheral upon request and is not lost.

If a channel is disabled by software, an active single or burst transaction is not guaranteed to receive an acknowledgement.

If the DW_ahb_dmac is configured to use defined length bursts (DMAH_INCR_BURSTS = 0), disabling the channel through software prior to completing a transfer is not allowed. Clearing the CH_EN bit prior to channel suspend may violate the AHB protocol.

6.8 Defined-Length Burst Support on DW_ahb_dmac

By default, the DW_ahb_dmac support incremental (INCR) bursts only. To achieve better performance, defined length bursts, such as INCR4, INCR8 and INCR16 are required. The DW_ahb_dmac can be configured to use defined-length bursts by setting the configuration parameter DMAH_INCR_BURSTS to 0. In this mode, the DW_ahb_dmac selects the largest valid defined-length burst to complete the transfer.

7 Verification

This chapter provides an overview of the testbench available for DW_ahb_dmac verification. Once the DW_ahb_dmac has been configured and the verification environment set up, simulations can be run automatically. For information on running simulations for DW_ahb_dmac in coreAssembler or coreConsultant, see "Building and Verifying a Component or Subsystem" in DesignWare Synthesizable Components for AMBA 2 User Guide.

Note The DW_ahb_dmac verification testbench is built with DesignWare Verification IP (VIP). Make sure that you have the supported version of the VIP components for this release, otherwise, you may experience some tool compatibility problems. For more information about supported tools in this release, refer to the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide.*

7.1 Overview of Vera Tests

The DW_ahb_dmac verification testbench performs the following set of tests that have been written to exhaustively verify the functionality and have also achieved maximum RTL code coverage:

- A single block transfer
- A single block transfer with register write back
- An LLP block chaining transfer
- A reloading transfer
- A transfer that is suspended and the channel terminated
- A transfer that is suspended and then resumed (without terminating the channel)
- A transfer where the various peripherals (SRC, DST and LLP) issue error responses
- A transfer where the channel is terminated part way through (without suspend)
- A transfer where the channel is terminated part way through (without suspend) using the global DW_ahb_dmac enable bit; terminates transfers running on all other channels as well

Within each of these transfers, all parameters are randomized.

The testbench constantly verifies whether conditions including the following are met:

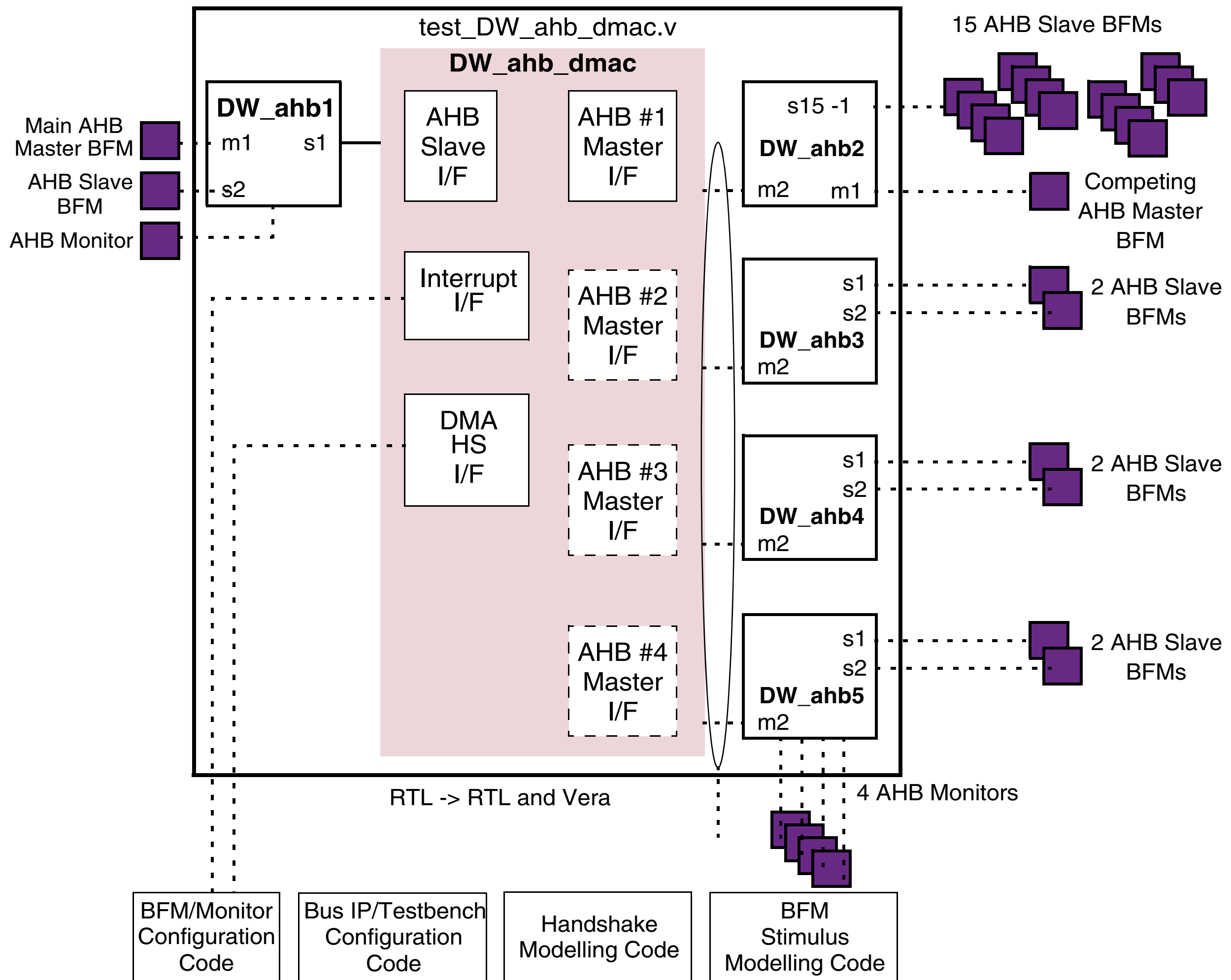
- Transfers are correct
- Registers are updated correctly
- Registers are written out correctly
- Interrupts are set correctly
- FIFO mode and flow control mode is not violated
- Bus and channel locking is correct
- Channel arbitration is correct

7.2 Overview of DW_ahb_dmac Testbench

As illustrated in Figure 7-1, the DW_ahb_dmac testbench is a Verilog testbench that includes:

- Verilog DUT (DW_ahb_dmac) and supporting bus IP (DW_ahb)
- Vera BFMs (AHB master and slave) and monitors (AHB only)
- Vera test harness and stimulus code

Figure 7-1 DW_ahb_dmac Testbench



The file, test_DW_ahb_dmac.v, shows the instantiation of the top-level design in a testbench and resides in the *workspace*/sim/testbench directory. The testbench tests your configuration specified in the Specify Configuration task of coreConsultant and is self-checking. When a coreKit has been configured, the verification environment is stored in *workspace*/sim. Files in *workspace*/sim/testbench form the actual testbench for DW_ahb_dmac. The *workspace*/sim/*test_name* directory contains the *test*.vrh header file and the encrypted *test*.vrp file.

The Competing AHB master BFM is used to compete with the DW_ahb_dmac when requesting the AHB bus and also to generate AHB traffic to modulate the otherwise uninhibited flow of data between the DW_ahb_dmac and AHB slaves. When the AMBA-lite version of the DW_ahb_dmac is being verified, the competing AHB master BFM is not used, and the DW_ahb components are replaced with AMBA-lite versions of the bus IP.

The top-level RTL code contains checkers to check for illegal programming of the DW_ahb_dmac. These monitors are removed upon synthesis. Because all simulators apart from VCS use a GTECH netlist, then the monitors are triggered only when you illegally program the DW_ahb_dmac while running an RTL simulation using the VCS simulator. An example of an illegal programming condition is where you program the source master select field of the control register, CTL0.SMS = 3, in a DW_ahb_dmac configuration where DMAH_NUM_MASTER_INT = 2. This triggers the following monitor illegal programming warning:

```
DW_ahb_dmac_CHECKER ILLEGAL_PROG: SMS field in ctl register channel 0
refers to master interface 4, but there are only 2 master interfaces
in the system ***
```

8
Integration Considerations

After you have configured, tested, and synthesized your component with the coreTools flow, you can integrate the component into your own design environment.

8.1 Performance

This section discusses performance and the parameters—software and hardware configuration parameters— that affect the performance of the DW_ahb_dmac.

8.1.1 Power Consumption, Frequency, and Area Results

Table 8-1 provides information about the synthesis results (power consumption, frequency, and area) of the DW_ahb_dmac using the industry standard 28nm technology library and how it affects performance.

Table 8-1 Power Consumption, Frequency, and Area Results for DW_ahb_dmac Using 28nm Technology Library

Configuration	Operating Frequency	Gate Count	Static Power Consumption	Dynamic Power Consumption
Default Configuration	hclk: 300 MHz	6534 gates	107 nW	20.4 uW
Default Configuration with clock gating: DMAH_LP_EN=1 DMAH_CH_LP_EN=1	hclk: 300 MHz	7156 gates	115nW	17.9 uW

Configuration	Operating Frequency	Gate Count	Static Power Consumption	Dynamic Power Consumption
Minimum Configuration: DMAH_NUM_MASTER_INT=1 DMAH_NUM_CHANNELS=1 DMAH_NUM_HS_INT=0 DMAH_MABRST=0 DMAH_RETURN_ERR_RESP=0 DMAH_INTR_POL=1 DMAH_INTR_IO=2 DMAH_BIG_ENDIAN=0 DMAH_S_HDATA_WIDTH=32 DMAH_M1_HDATA_WIDTH=32 DMAH_M2_HDATA_WIDTH=32 DMAH_M3_HDATA_WIDTH=32 DMAH_M4_HDATA_WIDTH=32 DMAH_M1_AHB_LITE=0 DMAH_M2_AHB_LITE=0 DMAH_M3_AHB_LITE=0 DMAH_M4_AHB_LITE=0 DMAH_ID_NUM=0x02080901 DMAH_CH0_FIFO_DEPTH=8 DMAH_CHx_STAT_DST=0 DMAH_CHx_STAT_SRC=0 DMAH_CH0_MAX_MULT_SIZE=4 DMAH_CH0_MAX_BLK_SIZE=3 DMAH_CHx_FC=0 DMAH_CHx_LOCK_EN=0 DMAH_CHx_LOCK_EN=0 DMAH_CHx_SMS=0 DMAH_CHx_DMS=0 DMAH_CHx_LMS=0 DMAH_CHx_STW=32 DMAH_CHx_DTW=32 DMAH_CH0_SRC_NON_OK=0 DMAH_CH0_DST_NON_OK=0 DMAH_CHx_LLP_NON_OK=0 DMAH_CHx_MULTI_BLK_EN=0 DMAH_CHx_HC_LLP=1 DMAH_CHx_SRC_GAT_EN=0 DMAH_CHx_DST_SCA_EN=0	hclk: 300 MHz	4840 gates	78.2 nW	15.045 uW

Configuration	Operating Frequency	Gate Count	Static Power Consumption	Dynamic Power Consumption
Maximum Configuration: DMAH_NUM_MASTER_INT=4 DMAH_NUM_CHANNELS=8 DMAH_NUM_HS_INT=16 DMAH_MABRST=0 DMAH_RETURN_ERR_RESP=0 DMAH_INTR_POL=1 DMAH_INTR_IO=1 DMAH_BIG_ENDIAN=0 DMAH_S_HDATA_WIDTH=32 DMAH_M1_HDATA_WIDTH=32 DMAH_M2_HDATA_WIDTH=32 DMAH_M3_HDATA_WIDTH=32 DMAH_M4_HDATA_WIDTH=32 DMAH_M1_AHB_LITE=0 DMAH_M2_AHB_LITE=0 DMAH_M3_AHB_LITE=0 DMAH_M4_AHB_LITE=0 DMAH_ID_NUM=0x02080901 DMAH_CHx_FIFO_DEPTH=16 DMAH_CHx_STAT_DST =1 DMAH_CHx_STAT_SRC =1 DMAH_CHx_MAX_MULT_SIZE=256 DMAH_CHx_MAX_BLK_SIZE=4095 DMAH_CHx_FC=3 DMAH_CHx_LOCK_EN=1 DMAH_CHx_SMS=4 DMAH_CHx_DMS=4 DMAH_CHx_LMS=4 DMAH_CHx_STW=0 DMAH_CHx_DTW=0 DMAH_CHx_SRC_NON_OK=1 DMAH_CHx_DST_NON_OK=1 DMAH_CHx_LLP_NON_OK=1 DMAH_CHx_MULTI_BLK_EN=1 DMAH_CHx_HC_LLP=0 DMAH_CHx_SRC_GAT_EN=1 DMAH_CHx_DST_SCA_EN=1 DMAH_CHx_MULTI_BLK_TYPE=8 DMAH_CHx_CTL_WB_EN=1	hclk: 300 MHz	139623 gates	2.23 uW	312 uW

Configuration	Operating Frequency	Gate Count	Static Power Consumption	Dynamic Power Consumption
Maximum Configuration with Low Power Mode: Same configuration as Maximum with the below extra configurations: DMAH_LP_EN=1 DMAH_CH_LP_EN=1	hclk: 300 MHz	142929 gates	2.3 uW	309 uW

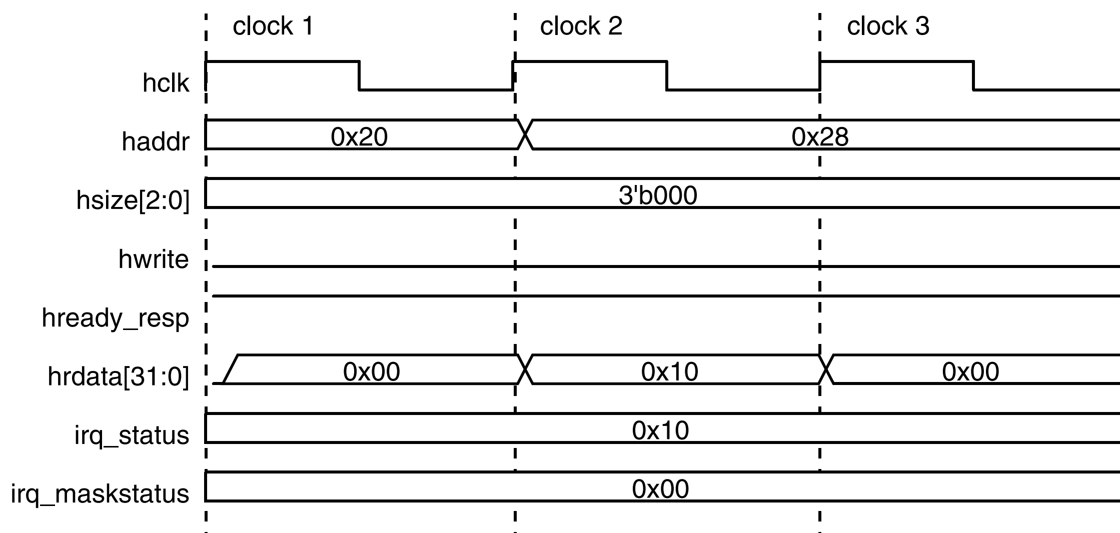
8.2 1KB Boundary Crossing

The AHB protocol requires that no AHB burst cross a 1KB address boundary. DW_ahb_dmac handles this situation automatically. If a DMA transfer is set up by software such that during the transfer one of the AHB transfers could cross a 1KB boundary, DW_ahb_dmac automatically sets up the AHB transfers such that the end of the 1KB boundary completes one AHB transfer and the start of the 1KB boundary starts another AHB transfer.

8.3 Read Accesses

For reads, registers less than the full access width return zeros in the unused upper bits. An AHB read takes two hclk cycles. The two cycles can be thought of as a control and data cycle, respectively. As shown in the following figure, the address and control is driven from clock 1 (control cycle); the read data for this access is driven by the slave interface onto the bus from clock 2 (data cycle) and is sampled by the master on clock 3. The operation of the AHB bus is pipelined, so while the read data from the first access is present on the bus for the master to sample, the control for the next access is present on the bus for the slave to sample.

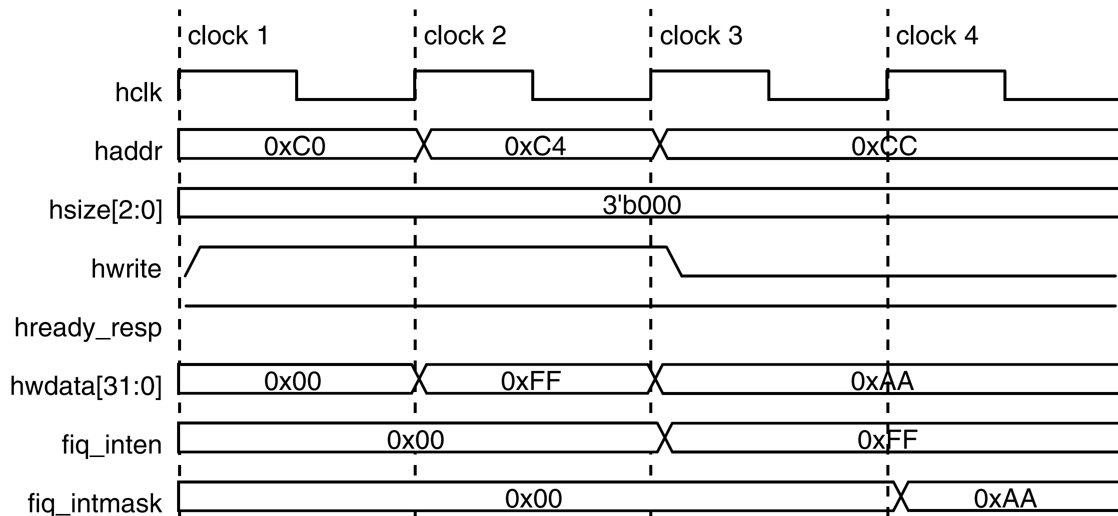
Figure 8-1 AHB Read



8.4 Write Accesses

When writing to a register, bit locations larger than the register width or allocation are ignored. Only pertinent bits are written to the register. Similar to the read case, a write access may be thought of as comprising a control and data cycle. As illustrated in the following figure, the address and control is driven from clock1 (control cycle), and the write data is driven by the bus from clock 2 (data cycle) and sampled by the destination register on clock 3.

Figure 8-2 AHB Write



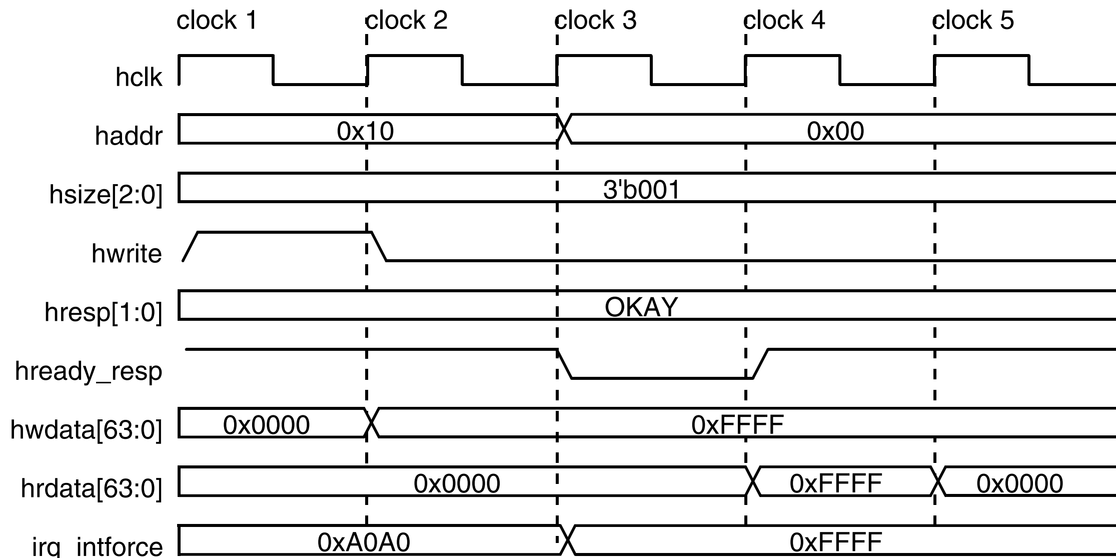
The operation of the AHB bus is pipelined, so while the write data for the first write is present on the bus for the slave to sample, the control for the next write is present on the bus for the slave to sample.

8.5 Consecutive Write-Read

This is a specific case for the AHB slave interface. The AMBA specification says that for a read after a write to the same address, the newly written data must be read back, not the old data. To comply with this, the slave interface in the DW_ahb_ictl inserts a “wait state” when it detects a read immediately after a write to

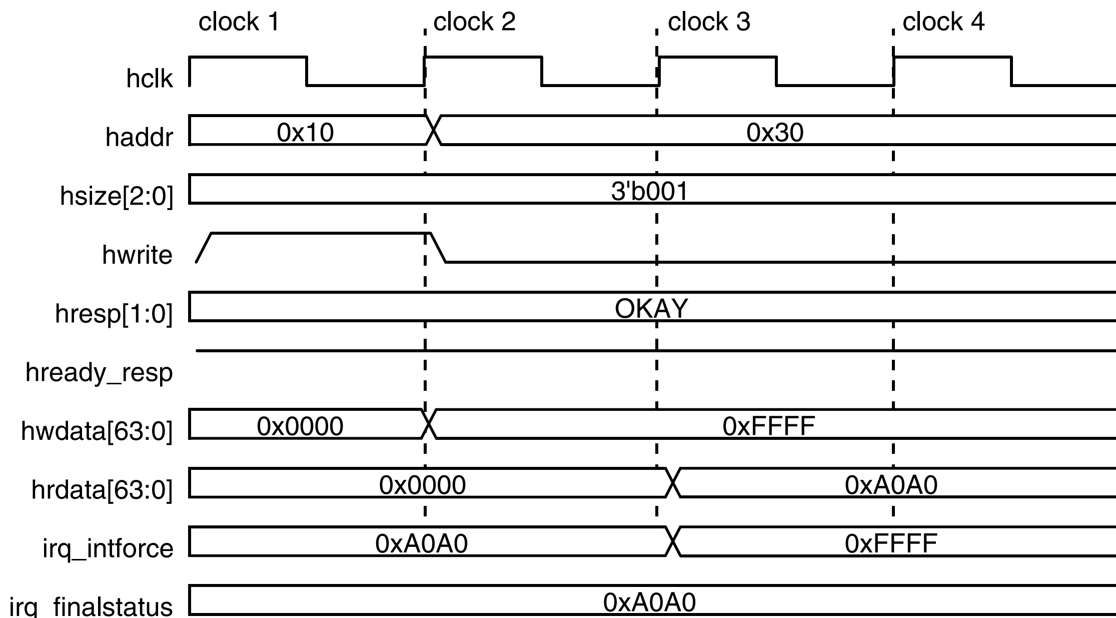
the same address. As shown in the following figure, the control for a write is driven on clock 1, followed by the write data and the control for a read from the same address on clock 2.

Figure 8-3 AHB Wait State Read/Write



Sensing the read after a write to the same address, the slave interface drives hready_resp low from clock 3; hready_resp is driven high on clock 4 when the new write data can be read; and the bus samples hready_resp high on clock 5 and reads the newly written data. The following figure shows a normal consecutive write-read access.

Figure 8-4 AHB Consecutive Read/Write



8.6 Accessing Top-level Constraints

To get SDC constraints out of coreConsultant, you need to first complete the synthesis activity and then use the “write_sdc” command to write out the results:

1. This cC command sets synthesis to write out scripts only, without running DC:

   ```
   set_activity_parameter Synthesize ScriptsOnly 1
   ```

2. This cC command autocompletes the activity:

   ```
   autocomplete_activity Synthesize
   ```

3. Finally, this cC command writes out SDC constraints:

   ```
   write_sdc <filename>
   ```

8.7 Coherency

Coherency is where bits within a register are logically connected. For instance, part of a register is read at time 1 and another part is read at time 2. Being coherent means that the part read at time 2 is at the same value it was when the register was read at time 1. The unread part is stored into a shadow register and this is read at time 2. When there is no coherency, no shadow registers are involved.

A bus master may need to be able to read the contents of a register, regardless of the data bus width, and be guaranteed of the coherency of the value read. A bus master may need to be able to write a register coherently regardless of the data bus width and use that register only when it has been fully programmed. This may need to be the case regardless of the relationship between the clocks.

Coherency enables a value to be read that is an accurate reflection of the state of the counter, independent of the data bus width, the counter width, and even the relationship between the clocks. Additionally, a value written in one domain is transferred to another domain in a seamless and coherent fashion.

Throughout this appendix the following terms are used:

- **Writing**. A bus master programs a configuration register. An example is programming the load value of a counter into a register.
- **Transferring**. The programmed register is in a different clock domain to where it is used, therefore, it needs to be transferred to the other clock domain.
- **Loading**. Once the programmed register is transferred into the correct clock domain, it needs to be loaded or used to perform its function. For example, once the load value is transferred into the counter domain, it gets loaded into the counter.

8.7.1 Writing Coherently

Writing coherently means that all the bits of a register can be written at the same time. A peripheral may have programmable registers that are wider than the width of the connected APB data bus, which prevents all the bits being programmed at the same time unless additional coherency circuitry is provided.

The programmable register could be the load value for a counter that may exist in a different clock domain. Not only does the value to be programmed need to be coherent, it also needs to be transferred to a different clock domain and then loaded into the counter. Depending on the function of the programmable register, a qualifier may need to be generated with the data so that it knows when the new value is currently transferred and when it should be loaded into the counter.

Depending on the system and on the register being programmed, there may be no need for any special coherency circuitry. One example that requires coherency circuitry is a 32-bit timer within an 8-bit APB system. The value is entirely programmed only after four 8-bit wide write transfers. It is safe to transfer or use the register when the last byte is currently written. An example where no coherency is required is a 16-bit wide timer within a 16-bit APB system. The value is entirely programmed after a single 16-bit wide write transfer.

Coherency circuitry enables the value to be loaded into the counter only when fully programmed and crossed over clock domains if the peripheral clock is not synchronous to the processor clock. While the load register is being programmed, the counter has access to the previous load value in case it needs to reload the counter.

Coherency circuitry is only added in cores where it is needed. The coherency circuitry incorporates an upper byte method that requires users to program the load register in LSB to MSB order when the peripheral width is smaller than the register width. When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are being programmed, they need to be stored in shadow registers so that the previous load register is available to the counter if it needs to reload. When the upper byte is programmed, the contents of the shadow registers and the upper byte are loaded into the load register.

The upper byte is the top byte of a register. A register can be transferred and loaded into the counter only when it has been fully programmed. A new value is available to the counter once this upper byte is written into the register. The following table shows the relationship between the register width and the peripheral bus width for the generation of the correct upper byte. The numbers in the table represent bytes, Byte 0 is the LSB and Byte 3 is the MSB. NCR means that no coherency circuitry is required, as the entire register is written with one access.

Table 8-2 Upper Byte Generation

	Upper Byte Bus Width		
Load Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	1	NCR	NCR
17 - 24	2	2	NCR
25 - 32	3	2 (or 3)	NCR

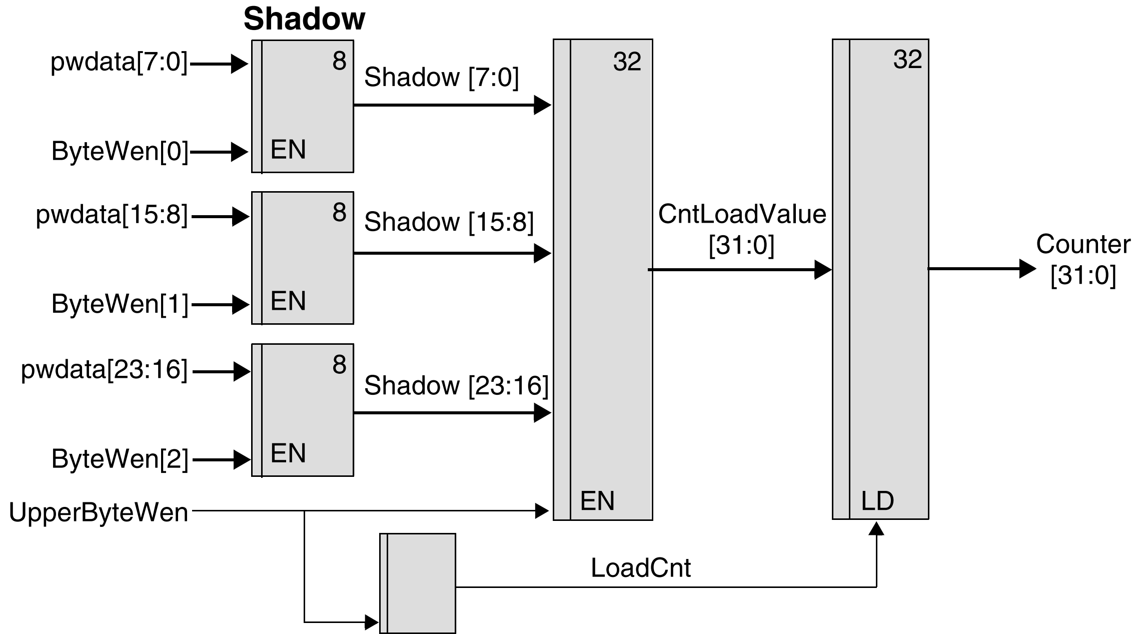
There are three relationship cases to be considered for the processor and peripheral clocks:

- Identical
- Synchronous (phase coherent but of an integer fraction)
- Asynchronous

8.7.1.1 Identical Clocks

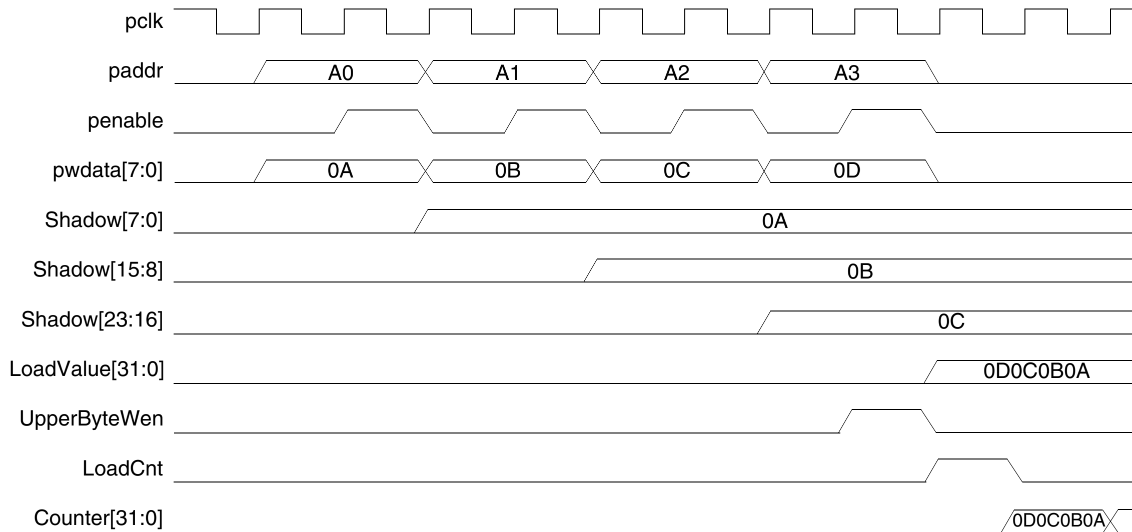
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 8-5 Coherent Loading – Identical Synchronous Clocks



The following figure shows a 32-bit register that is written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal lasts for one cycle and is used to load the counter with CntLoadValue.

Figure 8-6 Coherent Loading – Identical Synchronous Clocks



Each of the bytes that make up the load register are stored into shadow registers until the final byte is written. The shadow register is up to three bytes wide. The contents of the shadow registers and the final byte are transferred into the CntLoadValue register when the final byte is written. The counter uses this register to load/initialize itself. If the counter is operating in a periodic mode, it reloads from this register each time the count expires.

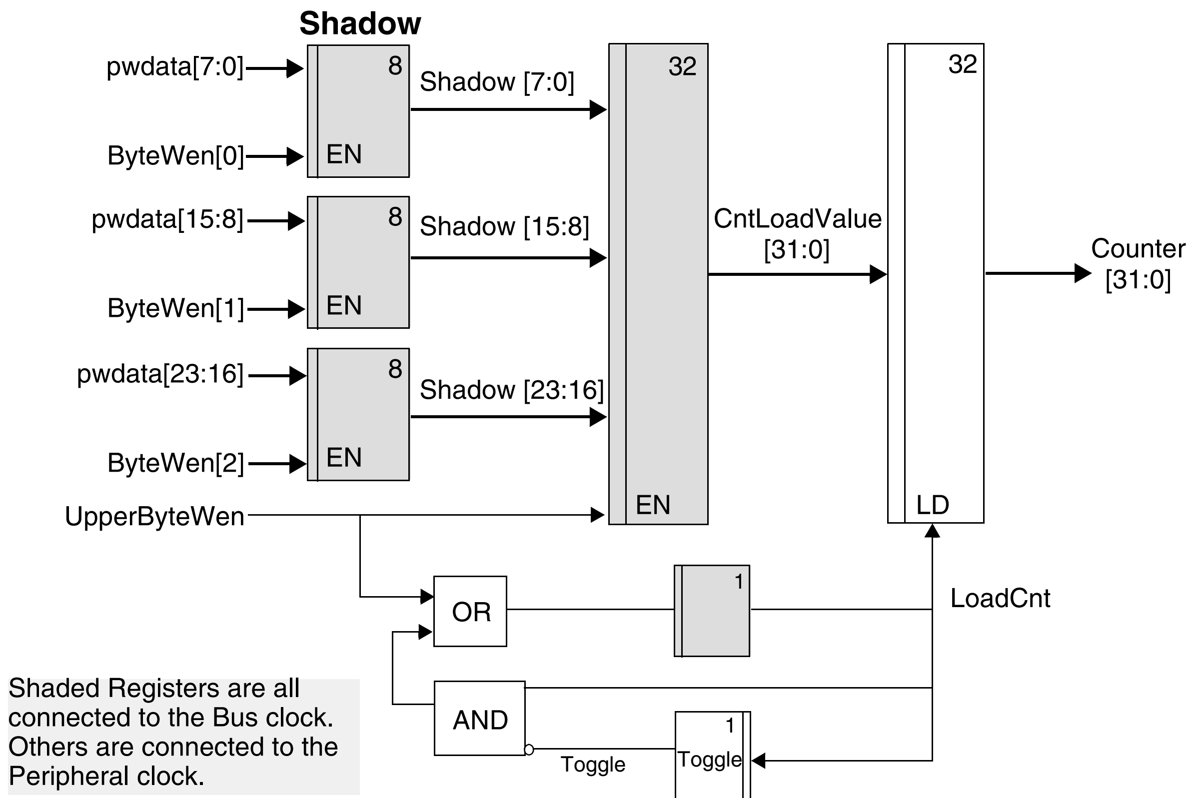
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this is signaled by LoadCnt = 1. After the upper byte is written, the LoadCnt goes to zero.

8.7.1.2 Synchronous Clocks

When the clocks are synchronous but do not have identical periods, the circuitry needs to be extended so that the LoadCnt signal is kept high until a rising edge of the counter clock occurs. This extension is necessary so that the value can be loaded, using LoadCnt, into the counter on the first counter clock edge. At the rising edge of the counter clock if LoadCnt is high, then a register clocked with the counter clock toggles, otherwise it keeps its current value. A circuit detecting the toggling is used to clear the original LoadCnt by looking for edge changes. The value is loaded into the counter when a toggle has been detected. Once it is loaded, the counter should be free to increment or decrement by normal rules.

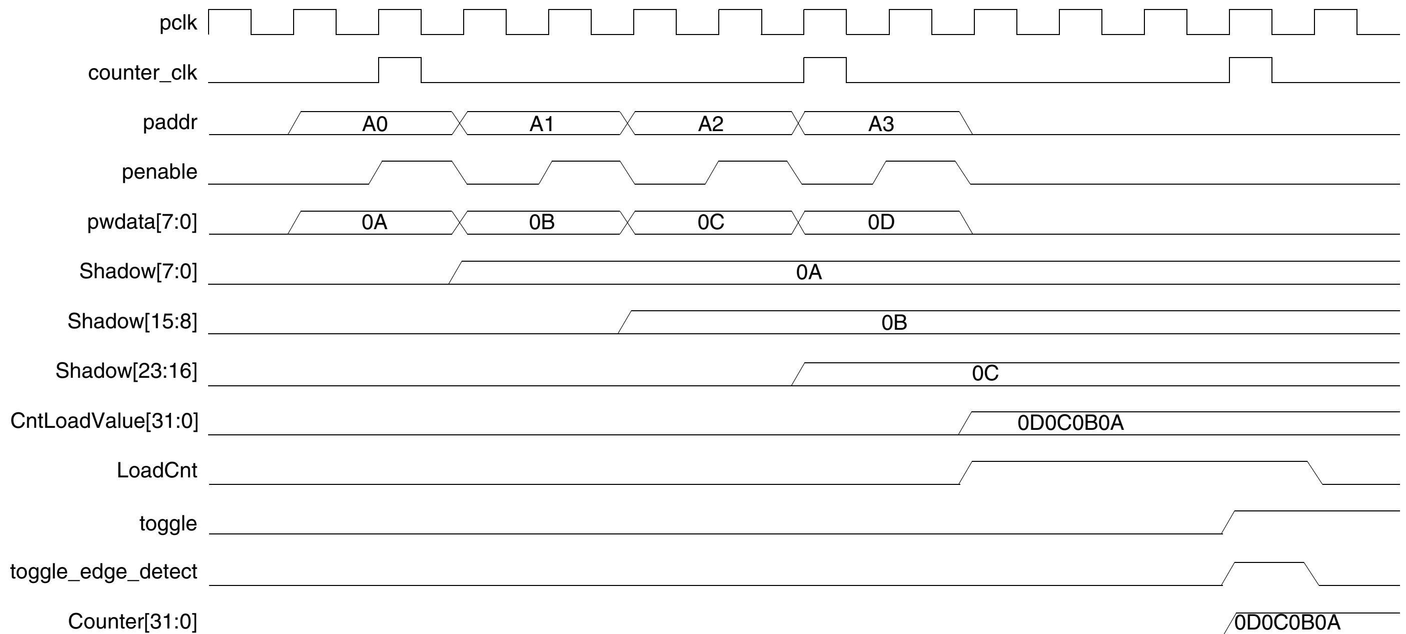
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

Figure 8-7 Coherent Loading – Synchronous Clocks



The following figure shows a 32-bit register being written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal is extended until a change in the toggle is detected and is used to load the counter.

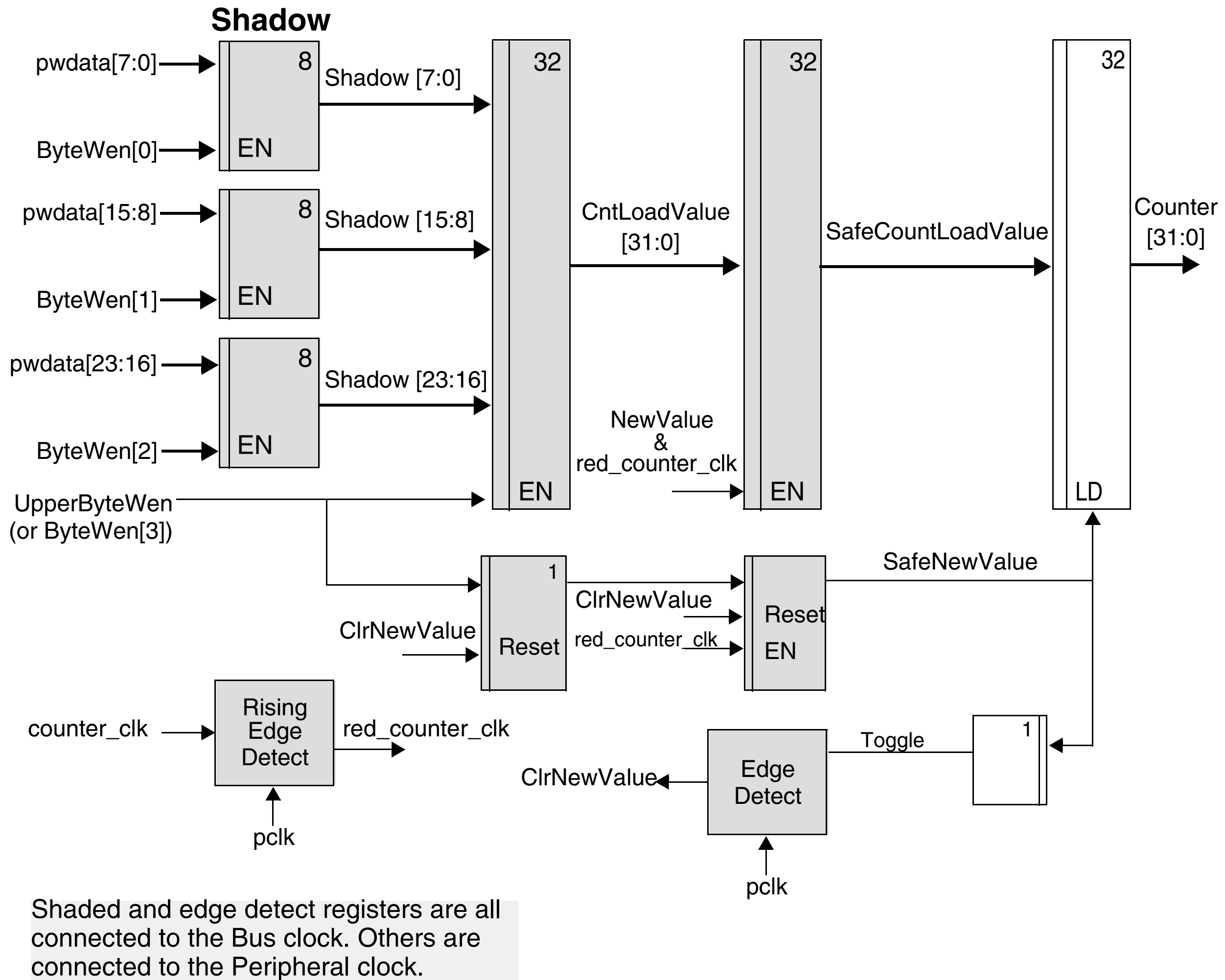
Figure 8-8 Coherent Loading – Synchronous Clocks



8.7.1.3 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three-times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock. The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are asynchronous.

Figure 8-9 Coherent Loading – Asynchronous Clocks



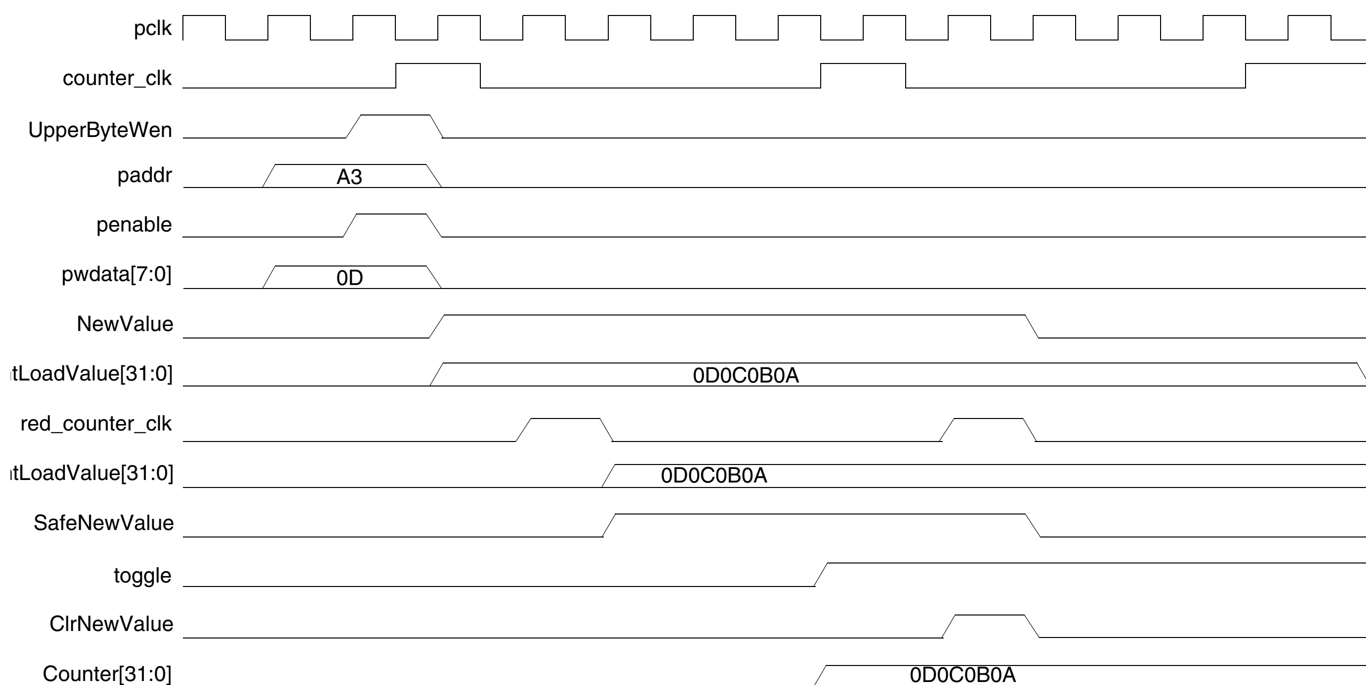
When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock to be used to re-time the peripheral clock. Upon a rising edge of the re-timed clock, the new value signal, NewValue, is transferred into a safe new value signal, SafeNewValue, which happens after the edge of the peripheral clock has occurred.

Every time there is a rising edge of the peripheral clock detected, the CntLoadValue is transferred into a SafeCntLoadValue. This value is used to transfer the load value across the clock domains. The SafeCntLoadValue only changes a number of bus clock cycles after the peripheral clock edge changes. A

counter running on the peripheral clock is able to use this value safely. It could be up to two peripheral clock periods before the value is loaded into the counter. Along with this loaded value, there also is a single bit transferred that is used to qualify the loading of the value into the counter.

The timing diagram depicted in the following figure does not show the shadow registers being loaded. This is identical to the loading for the other clock modes.

Figure 8-10 Coherent Loading – Asynchronous Clocks



The NewValue signal is extended until a change in the toggle is detected and is used to update the safe value. The SafeNewValue is used to load the counter at the rising edge of the peripheral clock. Each time a new value is written the toggle bit is flipped and the edge detection of the toggle is used to remove both the NewValue and the SafeNewValue.

8.7.2 Reading Coherently

For writing to registers, an upper-byte concept is proposed for solving coherency issues. For read transactions, a lower-byte concept is required. The following table provides the relationship between the register width and the bus width for the generation of the correct lower byte.

Table 8-3 Lower Byte Generation

	Lower Byte Bus Width		
Counter Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	0	NCR	NCR

Table 8-3 Lower Byte Generation

	Lower Byte Bus Width		
17 - 24	0	0	NCR
25 - 32	0	0	NCR

Depending on the bus width and the register width, there may be no need to save the upper bits because the entire register is read in one access, in which case there is no problem with coherency. When the lower byte is read, the remaining upper bytes within the counter register are transferred into a holding register. The holding register is the source for the remaining upper bytes. Users must read LSB to MSB for this solution to operate correctly. NCR means that no coherency circuitry is required, as the entire register is read with one access.

There are two cases regarding the relationship between the processor and peripheral clocks to be considered as follows:

- Identical and/or synchronous
- Asynchronous

8.7.2.1 Synchronous Clocks

When the clocks are identical and/or synchronous, the remaining unread bits (if any) need to be saved into a holding register once a read is started. The first read byte must be the lower byte provided in the previous table, which causes the other bits to be moved into the holding register, SafeCntVal, provided that the register cannot be read in one access. The upper bytes of the register are read from the holding register rather than the actual register so that the value read is coherent. This is illustrated in the following figure and in the timing diagram after it.

Figure 8-11 Coherent Registering – Synchronous Clocks

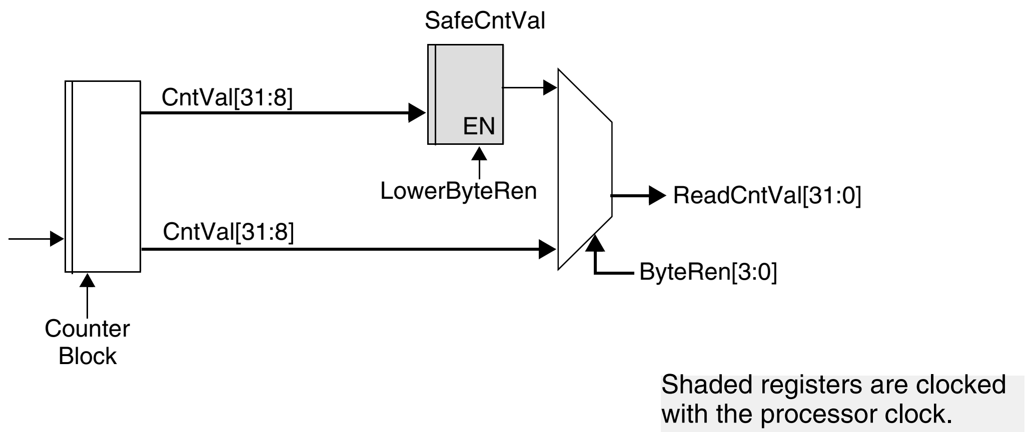
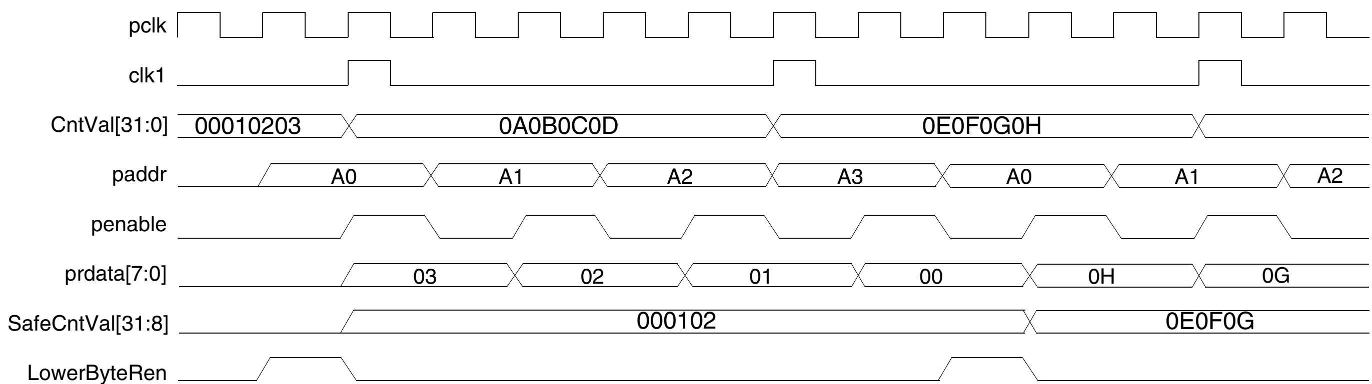


Figure 8-12 Coherent Registering – Synchronous Clocks



8.7.2.2 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock.

To safely transfer a counter value from the counter clock domain to the bus clock domain, the counter clock signal should be transferred to the bus clock domain. When the rising edge detect of this re-timed counter clock signal is detected, it is safe to use the counter value to update a shadow register that holds the current value of the counter.

While reading the counter contents it may take multiple APB transfers to read the value.

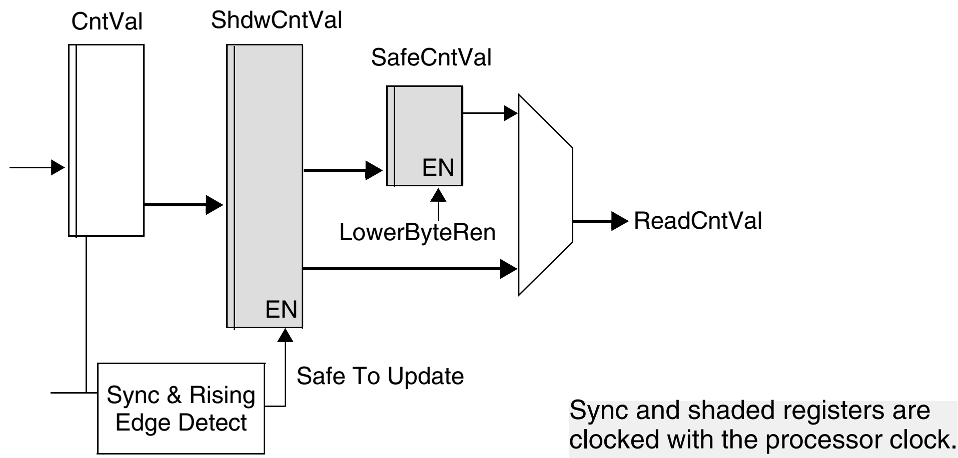
Note You must read LSB to MSB when the bus width is narrower than the counter width.

Once a read transaction has started, the value of the upper register bits need to be stored into a shadow register so that they can be read with subsequent read accesses. Storing these upper bits preserves the coherency of the value that is being read. When the processor reads the current value it actually reads the contents of the shadow register instead of the actual counter value. The holding register is read when the bus width is narrower than the counter width. When the LSB is read, the value comes from the shadow register; when the remaining bytes are read they come from the holding register. If the data bus width is wide enough to read the counter in one access, then the holding registers do not exist.

The counter clock is registered and successively pipelined to sense a rising edge on the counter clock. Having detected the rising edge, the value from the counter is known to be stable and can be transferred into the shadow register. The coherency of the counter value is maintained before it is transferred, because the value is stable.

The following figure illustrates the synchronization of the counter clock and the update of the shadow register.

Figure 8-13 Coherency and Shadow Registering – Asynchronous Clocks



A
Error and Warning Messages

This appendix lists and describes errors and warnings that you may encounter when configuring, verifying, or synthesizing the DesignWare component using the coreConsultant GUI.

A.1 Warnings During Simulation

When the Vera testbench is compiled, multiple copies of warnings similar to the following are generated:

```
Warning: call to potentially undefined super method "run" at
  <line number> in <file name>"
```

These messages can be ignored. The warning occurs because the run() task in the super class is defined as a virtual task, and the super class is an abstract class. This means that run() does not have to be defined. However, the super class always has run() defined, and the compiler should know this.

A.2 Warnings During Synthesis

If you use your own synthesis scripts and perform the ungroup command in Design Compiler (DC) on the design, DC generates warning messages that indicate that some of the DesignWare cells cannot be ungrouped. These messages are generated because few of the DesignWare components used in the design (like W_ram_r_w_s_dff) have no hierarchy and cannot be ungrouped further.

You can ignore these messages.

B
Internal Parameter Descriptions

Provides a description of the internal parameters that might be indirectly referenced in expressions in the Signals, Parameters, or Registers chapters. These parameters are not visible in the coreConsultant GUI and most of them are derived automatically from visible parameters. **You must not set any of these parameters directly.**

Some expressions might refer to TCL functions or procedures (sometimes identified as **function_of**) that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the core in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

Table B-1 Internal Parameters

Parameter Name	Equals To
BLOCK_TS	43:32
CH0_FIFO_DEPTH	{[function_of: DMAH_CH0_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH1_FIFO_DEPTH	{[function_of: DMAH_CH1_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH2_FIFO_DEPTH	{[function_of: DMAH_CH2_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH3_FIFO_DEPTH	{[function_of: DMAH_CH3_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH4_FIFO_DEPTH	{[function_of: DMAH_CH4_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH5_FIFO_DEPTH	{[function_of: DMAH_CH5_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH6_FIFO_DEPTH	{[function_of: DMAH_CH6_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}
CH7_FIFO_DEPTH	{[function_of: DMAH_CH7_FIFO_DEPTH MAX_AHB_HDATA_WIDTH]}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
CH_EN	0
CH_PRIOR	7:5
DEST_MSIZE	13:11
DINC	8:7
DMAH_BIG_ENDIAN_RST	{(DMAH_BIG_ENDIAN)}
DMAH_CH0_CTL_WB_EN_RST	{(DMAH_CH0_CTL_WB_EN)}
DMAH_CH0_DMS_RST	{(DMAH_CH0_DMS == 4) ? 4 : (DMAH_CH0_DMS == 3) ? 3 : (DMAH_CH0_DMS == 2) ? 2 : (DMAH_CH0_DMS == 1) ? 1 : 0}
DMAH_CH0_DST_SCA_EN_RST	{(DMAH_CH0_DST_SCA_EN)}
DMAH_CH0_DST_TR_RST	{(DMAH_CH0_DTW == 256) ? 6 : (DMAH_CH0_DTW == 128) ? 5 : (DMAH_CH0_DTW == 64) ? 4 : (DMAH_CH0_DTW == 32) ? 3 : (DMAH_CH0_DTW == 16) ? 2 : (DMAH_CH0_DTW == 8) ? 1 : 0}
DMAH_CH0_FC_RST	{(DMAH_CH0_FC == 3) ? 3 : (DMAH_CH0_FC == 2) ? 2 : (DMAH_CH0_FC == 1) ? 1 : 0}
DMAH_CH0_FIFO_DEPTH_RST	{(DMAH_CH0_FIFO_DEPTH == 256) ? 5 : (DMAH_CH0_FIFO_DEPTH == 128) ? 4 : (DMAH_CH0_FIFO_DEPTH == 64) ? 3 : (DMAH_CH0_FIFO_DEPTH == 32) ? 2 : (DMAH_CH0_FIFO_DEPTH == 16) ? 1 : 0}
DMAH_CH0_HC_LLP_RST	{(DMAH_CH0_HC_LLP)}
DMAH_CH0_LMS_RST	{(DMAH_CH0_LMS == 4) ? 4 : (DMAH_CH0_LMS == 3) ? 3 : (DMAH_CH0_LMS == 2) ? 2 : (DMAH_CH0_LMS == 1) ? 1 : 0}
DMAH_CH0_LOCK_EN_RST	{(DMAH_CH0_LOCK_EN)}
DMAH_CH0_MAX_BLK_SIZE_INT	{[function_of: DMAH_CH0_MAX_BLK_SIZE]}
DMAH_CH0_MAX_BLK_SIZE_RST	{(DMAH_CH0_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH0_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH0_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH0_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH0_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH0_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH0_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH0_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH0_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH0_MAX_BLK_SIZE == 7) ? 1 : 0}

Table B-1 Internal Parameters (Continued)

<table>
<tr><th>Parameter Name</th><th>Equals To</th></tr>
<tr><td>DMAH_CH0_MAX_MULT_SIZE_RST</td><td>{(DMAH_CH0_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH0_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH0_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH0_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH0_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH0_MAX_MULT_SIZE == 8) ? 1 : 0}</td></tr>
<tr><td>DMAH_CH0_MULTI_BLK_EN_RST</td><td>{(DMAH_CH0_MULTI_BLK_EN)}</td></tr>
<tr><td>DMAH_CH0_MULTI_BLK_TYPE_RST</td><td>{(DMAH_CH0_MULTI_BLK_TYPE)}</td></tr>
<tr><td>DMAH_CH0_RELOAD_DST_HC</td><td>{((DMAH_CH0_MULTI_BLK_EN == 1) && ((DMAH_CH0_MULTI_BLK_TYPE == 0) || (DMAH_CH0_MULTI_BLK_TYPE == 1) || (DMAH_CH0_MULTI_BLK_TYPE == 3) || (DMAH_CH0_MULTI_BLK_TYPE == 7))) ? 0 : 1}</td></tr>
<tr><td>DMAH_CH0_RELOAD_SRC_HC</td><td>{((DMAH_CH0_MULTI_BLK_EN == 1) && ((DMAH_CH0_MULTI_BLK_TYPE == 0) || (DMAH_CH0_MULTI_BLK_TYPE == 2) || (DMAH_CH0_MULTI_BLK_TYPE == 3) || (DMAH_CH0_MULTI_BLK_TYPE == 5))) ? 0 : 1}</td></tr>
<tr><td>DMAH_CH0_SMS_RST</td><td>{(DMAH_CH0_SMS == 4) ? 4 : (DMAH_CH0_SMS == 3) ? 3 : (DMAH_CH0_SMS == 2) ? 2 : (DMAH_CH0_SMS == 1) ? 1 : 0}</td></tr>
<tr><td>DMAH_CH0_SRC_GAT_EN_RST</td><td>{(DMAH_CH0_SRC_GAT_EN)}</td></tr>
<tr><td>DMAH_CH0_SRC_TR_RST</td><td>{(DMAH_CH0_STW == 256) ? 6 : (DMAH_CH0_STW == 128) ? 5 : (DMAH_CH0_STW == 64) ? 4 : (DMAH_CH0_STW == 32) ? 3 : (DMAH_CH0_STW == 16) ? 2 : (DMAH_CH0_STW == 8) ? 1 : 0}</td></tr>
<tr><td>DMAH_CH0_STAT_DST_RST</td><td>{(DMAH_CH0_STAT_DST)}</td></tr>
<tr><td>DMAH_CH0_STAT_SRC_RST</td><td>{(DMAH_CH0_STAT_SRC)}</td></tr>
<tr><td>DMAH_CH1_CTL_WB_EN_RST</td><td>{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_CTL_WB_EN) : 0}</td></tr>
<tr><td>DMAH_CH1_DMS_RST</td><td>{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_DMS == 4) ? 4 : (DMAH_CH1_DMS == 3) ? 3 : (DMAH_CH1_DMS == 2) ? 2 : (DMAH_CH1_DMS == 1) ? 1 : 0) : 0}</td></tr>
<tr><td>DMAH_CH1_DST_SCA_EN_RST</td><td>{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_DST_SCA_EN) : 0}</td></tr>
</table>

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH1_DST_TR_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_DTW == 256) ? 6 : (DMAH_CH1_DTW == 128) ? 5 : (DMAH_CH1_DTW == 64) ? 4 : (DMAH_CH1_DTW == 32) ? 3 : (DMAH_CH1_DTW == 16) ? 2 : (DMAH_CH1_DTW == 8) ? 1 : 0) : 0}
DMAH_CH1_FC_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_FC == 3) ? 3 : (DMAH_CH1_FC == 2) ? 2 : (DMAH_CH1_FC == 1) ? 1 : 0) : 0}
DMAH_CH1_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_FIFO_DEPTH == 256) ? 5: (DMAH_CH1_FIFO_DEPTH == 128) ? 4 : (DMAH_CH1_FIFO_DEPTH == 64) ? 3 : (DMAH_CH1_FIFO_DEPTH == 32) ? 2 : (DMAH_CH1_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH1_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_HC_LLP) : 0}
DMAH_CH1_LMS_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_LMS == 4) ? 4 : (DMAH_CH1_LMS == 3) ? 3 : (DMAH_CH1_LMS == 2) ? 2 : (DMAH_CH1_LMS == 1) ? 1 : 0) : 0}
DMAH_CH1_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_LOCK_EN) : 0}
DMAH_CH1_MAX_BLK_SIZE_RST	{(DMAH_CH1_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH1_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH1_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH1_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH1_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH1_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH1_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH1_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH1_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH1_MAX_BLK_SIZE == 7) ? 1 : 0}
DMAH_CH1_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH1_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH1_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH1_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH1_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH1_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH1_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_MULTI_BLK_EN) : 0}
DMAH_CH1_MULTI_BLK_TYPE_RST	{(DMAH_CH1_MULTI_BLK_TYPE)}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH1_SMS_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_SMS == 4) ? 4 : (DMAH_CH1_SMS == 3) ? 3 : (DMAH_CH1_SMS == 2) ? 2 : (DMAH_CH1_SMS == 1) ? 1 : 0) : 0}
DMAH_CH1_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_SRC_GAT_EN) : 0}
DMAH_CH1_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=2) ? ((DMAH_CH1_STW == 256) ? 6 : (DMAH_CH1_STW == 128) ? 5 : (DMAH_CH1_STW == 64) ? 4 : (DMAH_CH1_STW == 32) ? 3 : (DMAH_CH1_STW == 16) ? 2 : (DMAH_CH1_STW == 8) ? 1 : 0) : 0}
DMAH_CH1_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_STAT_DST) : 0}
DMAH_CH1_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH1_STAT_SRC) : 0}
DMAH_CH2_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_CTL_WB_EN) : 0}
DMAH_CH2_DMS_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_DMS == 4) ? 4 : (DMAH_CH2_DMS == 3) ? 3 : (DMAH_CH2_DMS == 2) ? 2 : (DMAH_CH2_DMS == 1) ? 1 : 0) : 0}
DMAH_CH2_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_DST_SCA_EN) : 0}
DMAH_CH2_DST_TR_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_DTW == 256) ? 6 : (DMAH_CH2_DTW == 128) ? 5 : (DMAH_CH2_DTW == 64) ? 4 : (DMAH_CH2_DTW == 32) ? 3 : (DMAH_CH2_DTW == 16) ? 2 : (DMAH_CH2_DTW == 8) ? 1 : 0) : 0}
DMAH_CH2_FC_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_FC == 3) ? 3 : (DMAH_CH2_FC == 2) ? 2 : (DMAH_CH2_FC == 1) ? 1 : 0) : 0}
DMAH_CH2_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_FIFO_DEPTH == 256) ? 5: (DMAH_CH2_FIFO_DEPTH == 128) ? 4 : (DMAH_CH2_FIFO_DEPTH == 64) ? 3 : (DMAH_CH2_FIFO_DEPTH == 32) ? 2 : (DMAH_CH2_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH2_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_HC_LLP) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH2_LMS_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_LMS == 4) ? 4 : (DMAH_CH2_LMS == 3) ? 3 : (DMAH_CH2_LMS == 2) ? 2 : (DMAH_CH2_LMS == 1) ? 1 : 0) : 0}
DMAH_CH2_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH2_LOCK_EN) : 0}
DMAH_CH2_MAX_BLK_SIZE_RST	{(DMAH_CH2_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH2_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH2_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH2_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH2_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH2_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH2_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH2_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH2_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH2_MAX_BLK_SIZE == 7) ? 1 : 0}
DMAH_CH2_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH2_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH2_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH2_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH2_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH2_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH2_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=2) ? (DMAH_CH2_MULTI_BLK_EN) : 0}
DMAH_CH2_MULTI_BLK_TYPE_RST	{(DMAH_CH2_MULTI_BLK_TYPE)}
DMAH_CH2_SMS_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_SMS == 4) ? 4 : (DMAH_CH2_SMS == 3) ? 3 : (DMAH_CH2_SMS == 2) ? 2 : (DMAH_CH2_SMS == 1) ? 1 : 0) : 0}
DMAH_CH2_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_SRC_GAT_EN) : 0}
DMAH_CH2_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=3) ? ((DMAH_CH2_STW == 256) ? 6 : (DMAH_CH2_STW == 128) ? 5 : (DMAH_CH2_STW == 64) ? 4 : (DMAH_CH2_STW == 32) ? 3 : (DMAH_CH2_STW == 16) ? 2 : (DMAH_CH2_STW == 8) ? 1 : 0) : 0}
DMAH_CH2_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_STAT_DST) : 0}
DMAH_CH2_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=3) ? (DMAH_CH2_STAT_SRC) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH3_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_CTL_WB_EN) : 0}
DMAH_CH3_DMS_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_DMS == 4) ? 4 : (DMAH_CH3_DMS == 3) ? 3 : (DMAH_CH3_DMS == 2) ? 2 : (DMAH_CH3_DMS == 1) ? 1 : 0) : 0}
DMAH_CH3_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_DST_SCA_EN) : 0}
DMAH_CH3_DST_TR_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_DTW == 256) ? 6 : (DMAH_CH3_DTW == 128) ? 5 : (DMAH_CH3_DTW == 64) ? 4 : (DMAH_CH3_DTW == 32) ? 3 : (DMAH_CH3_DTW == 16) ? 2 : (DMAH_CH3_DTW == 8) ? 1 : 0) : 0}
DMAH_CH3_FC_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_FC == 3) ? 3 : (DMAH_CH3_FC == 2) ? 2 : (DMAH_CH3_FC == 1) ? 1 : 0) : 0}
DMAH_CH3_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_FIFO_DEPTH == 256) ? 5: (DMAH_CH3_FIFO_DEPTH == 128) ? 4 : (DMAH_CH3_FIFO_DEPTH == 64) ? 3 : (DMAH_CH3_FIFO_DEPTH == 32) ? 2 : (DMAH_CH3_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH3_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_HC_LLP) : 0}
DMAH_CH3_LMS_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_LMS == 4) ? 4 : (DMAH_CH3_LMS == 3) ? 3 : (DMAH_CH3_LMS == 2) ? 2 : (DMAH_CH3_LMS == 1) ? 1 : 0) : 0}
DMAH_CH3_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_LOCK_EN) : 0}
DMAH_CH3_MAX_BLK_SIZE_RST	{(DMAH_CH3_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH3_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH3_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH3_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH3_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH3_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH3_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH3_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH3_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH3_MAX_BLK_SIZE == 7) ? 1 : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH3_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH3_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH3_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH3_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH3_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH3_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH3_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_MULTI_BLK_EN) : 0}
DMAH_CH3_MULTI_BLK_TYPE_RST	{(DMAH_CH3_MULTI_BLK_TYPE)}
DMAH_CH3_SMS_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_SMS == 4) ? 4 : (DMAH_CH3_SMS == 3) ? 3 : (DMAH_CH3_SMS == 2) ? 2 : (DMAH_CH3_SMS == 1) ? 1 : 0) : 0}
DMAH_CH3_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_SRC_GAT_EN) : 0}
DMAH_CH3_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=4) ? ((DMAH_CH3_STW == 256) ? 6 : (DMAH_CH3_STW == 128) ? 5 : (DMAH_CH3_STW == 64) ? 4 : (DMAH_CH3_STW == 32) ? 3 : (DMAH_CH3_STW == 16) ? 2 : (DMAH_CH3_STW == 8) ? 1 : 0) : 0}
DMAH_CH3_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_STAT_DST) : 0}
DMAH_CH3_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH3_STAT_SRC) : 0}
DMAH_CH4_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_CTL_WB_EN) : 0}
DMAH_CH4_DMS_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_DMS == 4) ? 4 : (DMAH_CH4_DMS == 3) ? 3 : (DMAH_CH4_DMS == 2) ? 2 : (DMAH_CH4_DMS == 1) ? 1 : 0) : 0}
DMAH_CH4_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_DST_SCA_EN) : 0}
DMAH_CH4_DST_TR_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_DTW == 256) ? 6 : (DMAH_CH4_DTW == 128) ? 5 : (DMAH_CH4_DTW == 64) ? 4 : (DMAH_CH4_DTW == 32) ? 3 : (DMAH_CH4_DTW == 16) ? 2 : (DMAH_CH4_DTW == 8) ? 1 : 0) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH4_FC_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_FC == 3) ? 3 : (DMAH_CH4_FC == 2) ? 2 : (DMAH_CH4_FC == 1) ? 1 : 0) : 0}
DMAH_CH4_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_FIFO_DEPTH == 256) ? 5: (DMAH_CH4_FIFO_DEPTH == 128) ? 4 : (DMAH_CH4_FIFO_DEPTH == 64) ? 3 : (DMAH_CH4_FIFO_DEPTH == 32) ? 2 : (DMAH_CH4_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH4_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_HC_LLP) :0}
DMAH_CH4_LMS_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_LMS == 4) ? 4 : (DMAH_CH4_LMS == 3) ? 3 : (DMAH_CH4_LMS == 2) ? 2 : (DMAH_CH4_LMS == 1) ? 1 : 0) : 0}
DMAH_CH4_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH4_LOCK_EN) : 0}
DMAH_CH4_MAX_BLK_SIZE_RST	{(DMAH_CH4_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH4_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH4_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH4_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH4_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH4_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH4_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH4_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH4_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH4_MAX_BLK_SIZE == 7) ? 1 : 0}
DMAH_CH4_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH4_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH4_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH4_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH4_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH4_MAX_MULT_SIZE == 8) ? 1 : 0) :0}
DMAH_CH4_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=4) ? (DMAH_CH4_MULTI_BLK_EN) : 0}
DMAH_CH4_MULTI_BLK_TYPE_RST	{(DMAH_CH4_MULTI_BLK_TYPE)}
DMAH_CH4_SMS_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_SMS == 4) ? 4 : (DMAH_CH4_SMS == 3) ? 3 : (DMAH_CH4_SMS == 2) ? 2 : (DMAH_CH4_SMS == 1) ? 1 : 0) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH4_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_SRC_GAT_EN) : 0}
DMAH_CH4_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=5) ? ((DMAH_CH4_STW == 256) ? 6 : (DMAH_CH4_STW == 128) ? 5 : (DMAH_CH4_STW == 64) ? 4 : (DMAH_CH4_STW == 32) ? 3 : (DMAH_CH4_STW == 16) ? 2 : (DMAH_CH4_STW == 8) ? 1 : 0) : 0}
DMAH_CH4_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_STAT_DST) : 0}
DMAH_CH4_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=5) ? (DMAH_CH4_STAT_SRC) : 0 }
DMAH_CH5_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_CTL_WB_EN) : 0}
DMAH_CH5_DMS_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_DMS == 4) ? 4 : (DMAH_CH5_DMS == 3) ? 3 : (DMAH_CH5_DMS == 2) ? 2 : (DMAH_CH5_DMS == 1) ? 1 : 0) : 0 }
DMAH_CH5_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_DST_SCA_EN) : 0}
DMAH_CH5_DST_TR_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_DTW == 256) ? 6 : (DMAH_CH5_DTW == 128) ? 5 : (DMAH_CH5_DTW == 64) ? 4 : (DMAH_CH5_DTW == 32) ? 3 : (DMAH_CH5_DTW == 16) ? 2 : (DMAH_CH5_DTW == 8) ? 1 : 0) : 0}
DMAH_CH5_FC_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_FC == 3) ? 3 : (DMAH_CH5_FC == 2) ? 2 : (DMAH_CH5_FC == 1) ? 1 : 0) : 0}
DMAH_CH5_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_FIFO_DEPTH == 256) ? 5: (DMAH_CH5_FIFO_DEPTH == 128) ? 4 : (DMAH_CH5_FIFO_DEPTH == 64) ? 3 : (DMAH_CH5_FIFO_DEPTH == 32) ? 2 : (DMAH_CH5_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH5_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_HC_LLP) : 0}
DMAH_CH5_LMS_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_LMS == 4) ? 4 : (DMAH_CH5_LMS == 3) ? 3 : (DMAH_CH5_LMS == 2) ? 2 : (DMAH_CH5_LMS == 1) ? 1 : 0) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH5_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_LOCK_EN) : 0}
DMAH_CH5_MAX_BLK_SIZE_RST	{(DMAH_CH5_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH5_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH5_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH5_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH5_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH5_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH5_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH5_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH5_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH5_MAX_BLK_SIZE == 7) ? 1 : 0}
DMAH_CH5_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH5_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH5_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH5_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH5_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH5_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH5_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_MULTI_BLK_EN) : 0}
DMAH_CH5_MULTI_BLK_TYPE_RST	{(DMAH_CH5_MULTI_BLK_TYPE)}
DMAH_CH5_SMS_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_SMS == 4) ? 4 : (DMAH_CH5_SMS == 3) ? 3 : (DMAH_CH5_SMS == 2) ? 2 : (DMAH_CH5_SMS == 1) ? 1 : 0) : 0 }
DMAH_CH5_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_SRC_GAT_EN) : 0 }
DMAH_CH5_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=6) ? ((DMAH_CH5_STW == 256) ? 6 : (DMAH_CH5_STW == 128) ? 5 : (DMAH_CH5_STW == 64) ? 4 : (DMAH_CH5_STW == 32) ? 3 : (DMAH_CH5_STW == 16) ? 2 : (DMAH_CH5_STW == 8) ? 1 : 0) : 0 }
DMAH_CH5_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_STAT_DST) : 0}
DMAH_CH5_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH5_STAT_SRC) : 0 }
DMAH_CH6_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_CTL_WB_EN) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH6_DMS_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_DMS == 4) ? 4 : (DMAH_CH6_DMS == 3) ? 3 : (DMAH_CH6_DMS == 2) ? 2 : (DMAH_CH6_DMS == 1) ? 1 : 0) : 0}
DMAH_CH6_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_DST_SCA_EN) : 0}
DMAH_CH6_DST_TR_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_DTW == 256) ? 6 : (DMAH_CH6_DTW == 128) ? 5 : (DMAH_CH6_DTW == 64) ? 4 : (DMAH_CH6_DTW == 32) ? 3 : (DMAH_CH6_DTW == 16) ? 2 : (DMAH_CH6_DTW == 8) ? 1 : 0) : 0}
DMAH_CH6_FC_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_FC == 3) ? 3 : (DMAH_CH6_FC == 2) ? 2 : (DMAH_CH6_FC == 1) ? 1 : 0) : 0}
DMAH_CH6_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_FIFO_DEPTH == 256) ? 5: (DMAH_CH6_FIFO_DEPTH == 128) ? 4 : (DMAH_CH6_FIFO_DEPTH == 64) ? 3 : (DMAH_CH6_FIFO_DEPTH == 32) ? 2 : (DMAH_CH6_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH6_HC_LLP_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_HC_LLP) : 0}
DMAH_CH6_LMS_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_LMS == 4) ? 4 : (DMAH_CH6_LMS == 3) ? 3 : (DMAH_CH6_LMS == 2) ? 2 : (DMAH_CH6_LMS == 1) ? 1 : 0) : 0}
DMAH_CH6_LOCK_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH6_LOCK_EN) : 0}
DMAH_CH6_MAX_BLK_SIZE_RST	{(DMAH_CH6_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH6_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH6_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH6_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH6_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH6_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH6_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH6_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH6_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH6_MAX_BLK_SIZE == 7) ? 1 : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH6_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH6_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH6_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH6_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH6_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH6_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH6_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS>=6) ? (DMAH_CH6_MULTI_BLK_EN) : 0}
DMAH_CH6_MULTI_BLK_TYPE_RST	{(DMAH_CH6_MULTI_BLK_TYPE)}
DMAH_CH6_SMS_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_SMS == 4) ? 4 : (DMAH_CH6_SMS == 3) ? 3 : (DMAH_CH6_SMS == 2) ? 2 : (DMAH_CH6_SMS == 1) ? 1 : 0) : 0}
DMAH_CH6_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_SRC_GAT_EN) : 0}
DMAH_CH6_SRC_TR_RST	{(DMAH_NUM_CHANNELS>=7) ? ((DMAH_CH6_STW == 256) ? 6 : (DMAH_CH6_STW == 128) ? 5 : (DMAH_CH6_STW == 64) ? 4 : (DMAH_CH6_STW == 32) ? 3 : (DMAH_CH6_STW == 16) ? 2 : (DMAH_CH6_STW == 8) ? 1 : 0) : 0}
DMAH_CH6_STAT_DST_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_STAT_DST) : 0}
DMAH_CH6_STAT_SRC_RST	{(DMAH_NUM_CHANNELS>=7) ? (DMAH_CH6_STAT_SRC):0}
DMAH_CH7_CTL_WB_EN_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_CTL_WB_EN) : 0}
DMAH_CH7_DMS_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_DMS == 4) ? 4 : (DMAH_CH7_DMS == 3) ? 3 : (DMAH_CH7_DMS == 2) ? 2 : (DMAH_CH7_DMS == 1) ? 1 : 0) : 0}
DMAH_CH7_DST_SCA_EN_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_DST_SCA_EN) : 0}
DMAH_CH7_DST_TR_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_DTW == 256) ? 6 : (DMAH_CH7_DTW == 128) ? 5 : (DMAH_CH7_DTW == 64) ? 4 : (DMAH_CH7_DTW == 32) ? 3 : (DMAH_CH7_DTW == 16) ? 2 : (DMAH_CH7_DTW == 8) ? 1 : 0) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH7_FC_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_FC == 3) ? 3 : (DMAH_CH7_FC == 2) ? 2 : (DMAH_CH7_FC == 1) ? 1 : 0) : 0}
DMAH_CH7_FIFO_DEPTH_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_FIFO_DEPTH == 256) ? 5: (DMAH_CH7_FIFO_DEPTH == 128) ? 4 : (DMAH_CH7_FIFO_DEPTH == 64) ? 3 : (DMAH_CH7_FIFO_DEPTH == 32) ? 2 : (DMAH_CH7_FIFO_DEPTH == 16) ? 1 : 0) : 0}
DMAH_CH7_HC_LLP_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_HC_LLP) : 0}
DMAH_CH7_LMS_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_LMS == 4) ? 4 : (DMAH_CH7_LMS == 3) ? 3 : (DMAH_CH7_LMS == 2) ? 2 : (DMAH_CH7_LMS == 1) ? 1 : 0) : 0}
DMAH_CH7_LOCK_EN_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_LOCK_EN) : 0}
DMAH_CH7_MAX_BLK_SIZE_RST	{(DMAH_CH7_MAX_BLK_SIZE == 4095) ? 10 : (DMAH_CH7_MAX_BLK_SIZE == 2047) ? 9 : (DMAH_CH7_MAX_BLK_SIZE == 1023) ? 8 : (DMAH_CH7_MAX_BLK_SIZE == 511) ? 7 : (DMAH_CH7_MAX_BLK_SIZE == 255) ? 6 : (DMAH_CH7_MAX_BLK_SIZE == 127) ? 5 : (DMAH_CH7_MAX_BLK_SIZE == 63) ? 4 : (DMAH_CH7_MAX_BLK_SIZE == 31) ? 3 : (DMAH_CH7_MAX_BLK_SIZE == 15) ? 2 : (DMAH_CH7_MAX_BLK_SIZE == 7) ? 1 : 0}
DMAH_CH7_MAX_MULT_SIZE_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_MAX_MULT_SIZE == 256) ? 6 : (DMAH_CH7_MAX_MULT_SIZE == 128) ? 5 : (DMAH_CH7_MAX_MULT_SIZE == 64) ? 4 : (DMAH_CH7_MAX_MULT_SIZE == 32) ? 3 : (DMAH_CH7_MAX_MULT_SIZE == 16) ? 2 : (DMAH_CH7_MAX_MULT_SIZE == 8) ? 1 : 0) : 0}
DMAH_CH7_MULTI_BLK_EN_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_MULTI_BLK_EN) : 0}
DMAH_CH7_MULTI_BLK_TYPE_RST	{(DMAH_CH7_MULTI_BLK_TYPE)}
DMAH_CH7_SMS_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_SMS == 4) ? 4 : (DMAH_CH7_SMS == 3) ? 3 : (DMAH_CH7_SMS == 2) ? 2 : (DMAH_CH7_SMS == 1) ? 1 : 0) : 0}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_CH7_SRC_GAT_EN_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_SRC_GAT_EN) : 0}
DMAH_CH7_SRC_TR_RST	{(DMAH_NUM_CHANNELS ==8) ? ((DMAH_CH7_STW == 256) ? 6 : (DMAH_CH7_STW == 128) ? 5 : (DMAH_CH7_STW == 64) ? 4 : (DMAH_CH7_STW == 32) ? 3 : (DMAH_CH7_STW == 16) ? 2 : (DMAH_CH7_STW == 8) ? 1 : 0) : 0}
DMAH_CH7_STAT_DST_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_STAT_DST) : 0}
DMAH_CH7_STAT_SRC_RST	{(DMAH_NUM_CHANNELS ==8) ? (DMAH_CH7_STAT_SRC) :0 }
DMAH_COMP_ID	32'h44571110
DMAH_DEBUG_BUS	1
DMAH_HADDR_WIDTH	32
DMAH_INTR_IO_RST	{(DMAH_INTR_IO == 0) ? 0 : ((DMAH_INTR_IO == 1) ? 1 : 2)}
DMAH_M1_HDATA_WIDTH_RST	{(DMAH_M1_HDATA_WIDTH == 256) ? 3 : (DMAH_M1_HDATA_WIDTH == 128) ? 2 : (DMAH_M1_HDATA_WIDTH == 64) ? 1 : 0}
DMAH_M2_HDATA_WIDTH_RST	{(DMAH_M2_HDATA_WIDTH == 256) ? 3 : (DMAH_M2_HDATA_WIDTH == 128) ? 2 : (DMAH_M2_HDATA_WIDTH == 64) ? 1 : 0}
DMAH_M3_HDATA_WIDTH_RST	{(DMAH_M3_HDATA_WIDTH == 256) ? 3 : (DMAH_M3_HDATA_WIDTH == 128) ? 2 : (DMAH_M3_HDATA_WIDTH == 64) ? 1 : 0}
DMAH_M4_HDATA_WIDTH_RST	{(DMAH_M4_HDATA_WIDTH == 256) ? 3 : (DMAH_M4_HDATA_WIDTH == 128) ? 2 : (DMAH_M4_HDATA_WIDTH == 64) ? 1 : 0}
DMAH_MABRST_RST	{(DMAH_MABRST)}
DMAH_NUM_CHANNELS_RST	{(DMAH_NUM_CHANNELS - 1)}
DMAH_NUM_HS_INT_NZ	{[function_of: DMAH_NUM_HS_INT]}
DMAH_NUM_HS_INT_RST	{(DMAH_NUM_HS_INT)}
DMAH_NUM_MASTER_INT_RST	{(DMAH_NUM_MASTER_INT - 1)}
DMAH_NUM_PER	{[function_of: DMAH_NUM_CHANNELS]}
DMAH_REG_HS_IF	{DMAH_REMOVE_PIPELINING ? 0 : 1}

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
DMAH_S_HDATA_WIDTH_RST	{(DMAH_S_HDATA_WIDTH == 256) ? 3 : (DMAH_S_HDATA_WIDTH == 128) ? 2 : (DMAH_S_HDATA_WIDTH == 64) ? 1 : 0}
DMAH_STATIC_ENDIAN_SELECT_RST	{(DMAH_STATIC_ENDIAN_SELECT)}
DMAH_VERSION_ID	32'h3232322a
DMS	24:23
DST_SCATTER_EN	18
DS_UPD_EN	37
FCMODE	32
FIFO_EMPTY	9
FIFO_MODE	33
HS_SEL_DST	10
HS_SEL_SRC	11
INT_EN	0
LLP_DST_EN	27
LLP_SRC_EN	28
LMS	1:0
LOC	(DMAH_HADDR_WIDTH-1):2
LOCK_B	17
LOCK_B_L	15:14
LOCK_CH	16
LOCK_CH_L	13:12
LOG2_DMAH_NUM_HS_INT	{[function_of: DMAH_NUM_HS_INT]}
LOG2_DMAH_NUM_PER	{[function_of: DMAH_NUM_PER]}
MAX_AHB_HDATA_WIDTH	{[function_of:]}
MAX_LOG2_FIFO_DEPTH_BYTES	{[function_of:]}
RELOAD_DST	31
RELOAD_SRC	30
SINC	10:9

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
SMS	26:25
SRC_GATHER_EN	17
SRC_MSIZE	16:14
SRC_PER	42:39
SRC_TR_WIDTH	6:4
SS_UPD_EN	38
TT_FC	22:20

C

Channel Locking and Deadlock

This appendix explains how locking can cause deadlock. Deadlock situations can occur where multiple channels are enabled concurrently and no channel can proceed with its DMA transfer. This only occurs for configurations where DMAH_NUM_MASTER_INT > 1 and DMAH_NUM_CHANNELS > 1.

The methods used to avoid deadlock are:

- Hardware detects when a potential deadlock occurs between two channels and deal with it. Refer to "Deadlock Prevention by Hardware" on page 412.
- Programming restrictions to avoid potential deadlock situations. Refer to "Programming Restrictions to Avoid Deadlock" on page 412.

Note

Refer to "Channel Locking" on page 96 for details of the actual value of channel locking enable and channel locking level used in the DMA transfer. These may differ from the programmed values, *CFGx_LOCK_CH* and *CFGx.LOCK_CH_L*. All references to channel locking enable and channel locking level in this section refer to the actual values used and not the programmed values.

C.1 Hardware Detection of Deadlock

Three cases are outlined that can potentially cause a deadlock situation.

C.1.1 Case 1

Consider a channel programmed as follows:

C.1.1.1 Channel 0

CTL0.SMS = 0	Source on AHB Layer 0
CTL0.DMS = 1	Destination on AHB Layer 1
CFG0.LOCK_CH_L = 0	Locking over complete DMA transfer
CFG0.LOCK_CH = 1	Channel Locking enabled

C.1.1.2 Channel 1

CTL1.SMS = 1	Source on AHB Layer 1
CTL1.DMS = 0	Destination on AHB Layer 0
CFG1.LOCK_CH_L = 0	Locking over complete DMA transfer
CFG1.LOCK_CH = 1	Channel Locking enabled

Consider the following scenario. Channel 0 and Channel 1 are both enabled. The source of Channel 0 is granted ownership of Layer 0 and then locks out all other channel access to Layer 0 until it completes the DMA transfer. Before the Channel 0 destination requests Layer 1, the source of Channel 1 requests and is granted Layer 1. Channel 1 now locks all other channel accesses to Layer 1.

This is a deadlock situation whereby neither Channel 1 nor Channel 0 can complete because neither channel destination can gain access to its respective AHB layer.

C.1.1.3 Case 1 potential deadlock

Hardware flags a potential deadlock situation between two channels (channels *x* and *y*) if:

- Both channels are enabled
- CTL*x*.SMS = CTL*y*.DMS
- CTL*y*.SMS = CTL*x*.DMS
- Both channels have channel locking enabled

C.1.2 Case 2

This case occurs only when channel locking is enabled at the DMA transaction level.

C.1.2.1 Case 2 potential deadlock

Hardware flags a potential deadlock situation between two channels if:

1. Both channels are enabled
2. CTLv.SMS = CTL*y*.SMS
3. CTL*x*.DMS = CTL*y*.DMS
4. CTL*x*.SMS != CTL*x*.DMS and CTL*y*.SMS != CTL*y*.DMS
5. Both channels have channel locking enabled
6. Either channel *x* or channel *y* has channel locking enabled at the transaction level

C.1.3 Case 3

Another potential deadlock occurs when DMAH_NUM_MASTER_INT > 2 &&
DMAH_NUM_CHANNELS > 2. The following shows two examples when channel locking is enabled at any level and where a potential deadlock situation can occur:

C.1.3.1 Example 1

CTL0.SMS = 0	Channel 0 Source on AHB Layer 0
CTL0.DMS = 1	Channel 0 Destination on AHB Layer 1
CTL1.SMS = 1	Channel 1 Source on AHB Layer 1
CTL1.DMS = 2	Channel 1 Destination on AHB Layer 2
CTL2.SMS = 2	Channel 2 Source on AHB Layer 2
CTL2.DMS = 0	Channel 2 Destination on AHB Layer 0

For example, suppose all three channels employ channel locking at the block transfer level. If the source of all three channels start before any of the destinations start, then a deadlock situation arises.

1. Layer 0 locked by the source of channel 0
2. Layer 1 locked by the source of channel 1
3. Layer 2 locked by the source of channel 2

Then:

1. The destination of channel 0 cannot proceed as Layer 1 is locked by channel 1
2. The destination of channel 1 cannot proceed as Layer 2 is locked by channel 2
3. The destination of channel 2 cannot proceed as Layer 0 is locked by channel 0

This is a deadlocked situation.

C.1.3.2 Example 2

CTL0.SMS = 0	Channel 0 Source on AHB Layer 0
CTL0.DMS = 1	Channel 0 Destination on AHB Layer 1
CTL1.SMS = 1	Channel 1 Source on AHB Layer 1
CTL1.DMS = 2	Channel 1 Destination on AHB Layer 2
CTL2.SMS = 2	Channel 2 Source on AHB Layer 2
CTL2.DMS = 3	Channel 2 Destination on AHB Layer 3
CTL3.SMS = 3	Channel 3 Source on AHB Layer 3
CTL3.DMS = 3	Channel 3 Destination on AHB Layer 3

This example is similar to "Example 1" on page 411. Suppose all four channels employ channel locking at the block transfer level. If the source of all four channels start before any of the destinations start, then a deadlock situation arises.

C.1.3.3 Case 3 potential deadlock

Hardware flags a potential deadlock situation between two channels if:

1. DMAH_NUM_MASTER_INT > 2 && DMAH_NUM_CHANNELS > 2
2. Both channels are enabled
3. Both channels have channel locking enabled at any level
4. The channel source and destination are on different layers
5. The source of one channel is on the same layer as the destination of the other channel

C.1.4 Deadlock Prevention by Hardware

Hardware detects a potential deadlock situation between two channels, as given in Case 1 potential deadlock, Case 2 potential deadlock, and Case 3 potential deadlock. It then stalls one channel until the enable bit of the other channel in the Channel Enable Register, ChEnReg.Ch_En, is cleared by hardware. The channel that is first enabled by software is allowed to complete its DMA transfer before the other channel can begin its DMA transfer. In this case, a lower priority channel can be given precedence over a higher one if it is enabled first over the AHB slave interface; for example, the programmed channel priorities are overridden.

Note This preventative method of avoiding deadlock assumes that no two channels are enabled by software at the same time. To avoid enabling two or more channels at the same time, writes to ChEnReg.ChEn should enable only one channel per write. Also, when enabling the DMAC with a write to DmaCfgReg.DMA_EN, there should be no more than one ChEnReg.ChEn bit already enabled.

C.2 Programming Restrictions to Avoid Deadlock

The previous section details scenarios where hardware detects potential deadlock situations and takes preventative action. This section details scenarios where hardware does not detect potential deadlock situations, but it is left up to software to avoid programming the DW_ahb_dmac in a way that may cause deadlock. Failure to comply with these programming restrictions may lead to deadlock.

1. If
   a. Channel locking is enabled at the DMA transfer level
   b. The LLP*x*.LOC field is non-zero and
   c. Block chaining is enabled on the source or destination side.

In equation form, this is:

((CFG*x*.LOCK_B == 1 && CFG.LOCK_B_L == 2'b00) | |
(CFG*x*.LOCK_CH == 1 && CFG.LOCK_CH_L == 2'b00)) &&
(LLP*x*.LOC != 0) &&
((CTL*x*.LLP_SRC_EN == 1) | | ((CTL*x*.LLP_DST_EN == 1))

then the following condition must be complied with:

(LLP*x*.LMS == CTL*x*.SMS) | | (LLP*x*.LMS == CTL*x*.DMS)

2. If:

   CTL*x*.SMS != CTL*x*.DMS && DMAH_NUM_MASTER_INT > 2 && DMAH_NUM_CHANNELS > 2 then channel transaction level locking is prohibited. Software must ensure that if channel locking is enabled, then the locking level used for channel locking is not at the transaction level.

3. If the DMA block size in bytes is less than or equal to the FIFO depth in bytes (the source block can complete before the destination block starts) and if the source and destination are on different layers, only transaction-level locking is allowed for channel locking.

D DW_ahb_dmac Application Notes

This appendix provides application notes for the DW_ahb_dmac.

D.1 Interoperability Between DW_ahb_dmac and PrimeCell Hardware Handshaking Interface

You may have already designed a peripheral with a handshaking interface that conforms to the ARM PrimeCell handshaking interface for use with the ARM PrimeCell DMA. This handshaking interface can be mapped to conform with the DW_ahb_dmac handshaking interface with glue logic, as shown in Figure D-1 through Figure D-3. Table D-1 lists PrimeCell handshaking signals.

Table D-1 PrimeCell Handshaking Signals

ARM PrimeCell DMA Handshaking Signal	Direction Relative to Peripheral	Description
dmacclr	Input	DMA request acknowledge clear.
dmactc	Input	DMA terminal count. Indicates the transaction is complete and the block of data is transferred.
dmacbreq	Output	DMA burst transfer request.
dmacsreq	Output	DMA single transfer request.
dmaclbreq	Output	DMA last burst transfer request.
dmaclsreq	Output	DMA last single transfer request.

Figure D-1 Mapping Between ARM PrimeCell and DW_ahb_dmac Peripheral Handshaking – Peripheral (source or destination) is Flow Controller

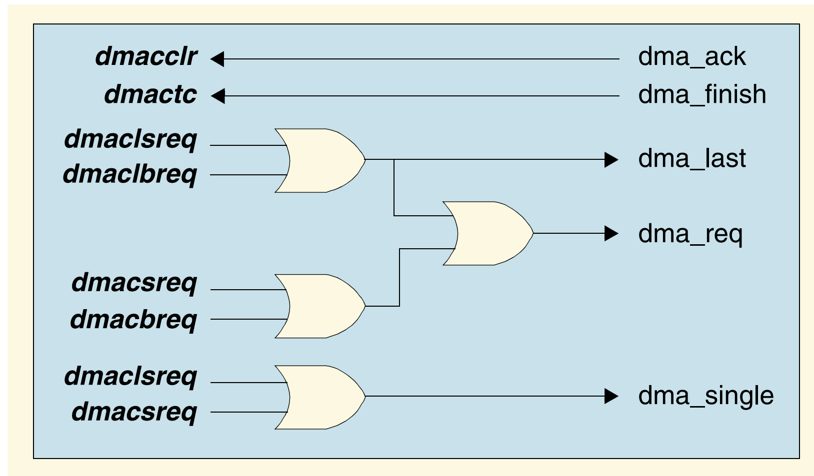


Figure D-2 Mapping Between ARM PrimeCell and DW_ahb_dmac Source Peripheral Handshaking – Source Peripheral Not Flow Controller

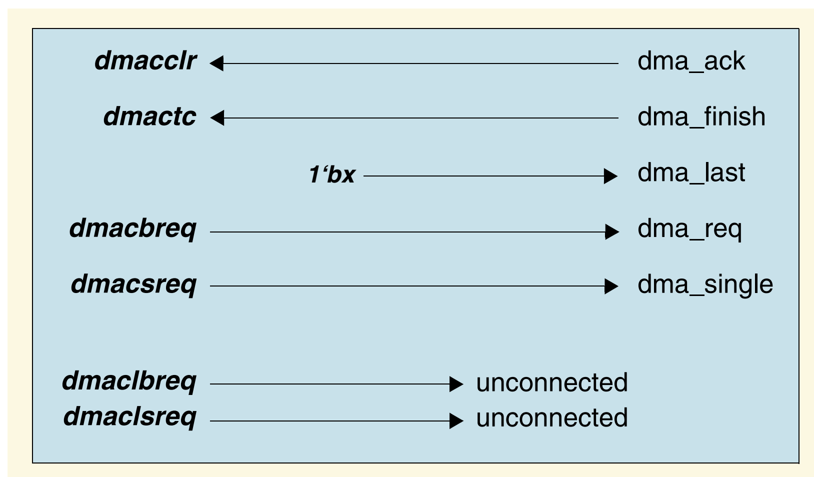
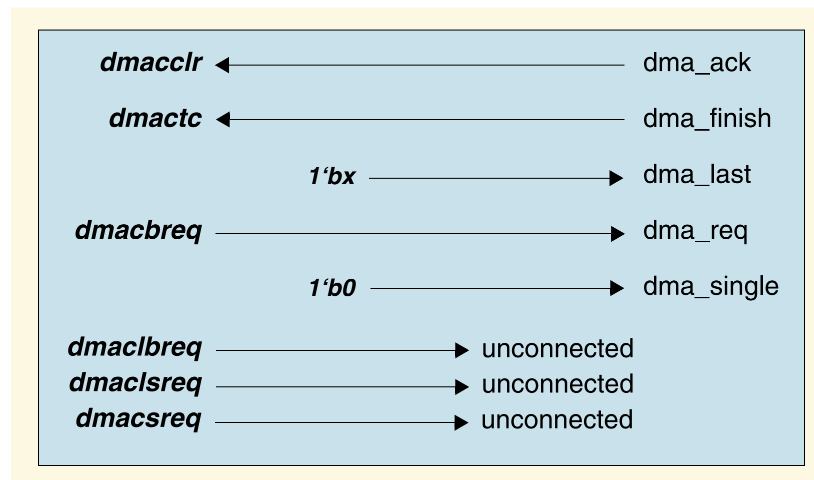
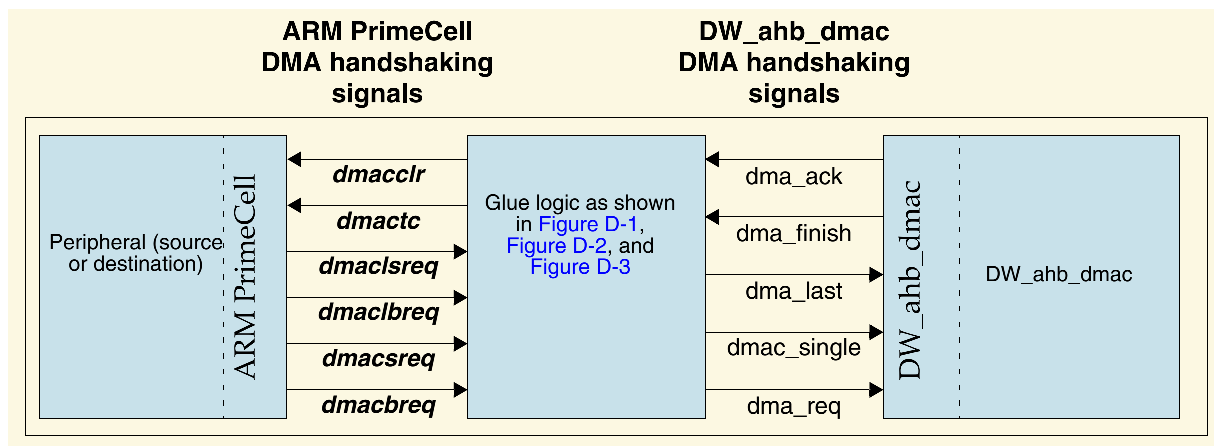


Figure D-3 Mapping Between ARM PrimeCell and DW_ahb_dmac Destination Peripheral Handshaking – Destination Peripheral Not Flow Controller



This glue logic may be used to allow a peripheral designed with the ARM PrimeCell handshaking interface to operate with DW_ahb_dmac. This logic is shown in the Figure D-4.

Figure D-4 DMA Operation Using ARM PrimeCell Handshaking Interface with DW_ahb_dmac



D.2 Mapping of PrimeCell Software Handshaking Registers to DW_ahb_dmac

This application note describes the mapping of the PrimeCell software handshaking registers to the equivalent DW_ahb_dmac software handshaking interface registers. Software that is used to control the PrimeCell DMA Controller software interface registers can be changed to conform to the DW_ahb_dmac software interface by the mapping described in the following sections.

D.2.1 PrimeCell Software Handshaking Registers

The ARM PrimeCell DMA Controller (PL080 and PL081) has four software registers that are used to control the DMA handshaking. These registers allow DMA requests to be generated by software.

- DMACSoftBReq (16 bits) – Generates DMA burst requests
- DMACSoftSReq (16 bits) – Generates DMA single requests

- DMACSoftLBReq (16 bits) – Generates DMA last burst requests
- DMACSoftLSReq (16 bits) – Generates DMA last single requests

A DMA request can be generated for a peripheral (Source or Destination) by writing a 1 to the corresponding register bit of the previous registers.

D.2.2 DW_ahb_dmac Software Handshaking Registers

DW_ahb_dmac has six registers allocated for software handshaking:

- ReqSrcReg (64 bits) – Generates DMA requests for source peripheral
- ReqDstReg (64 bits) – Generates DMA requests for destination peripheral
- SglReqSrcReg (64 bits) – Generates DMA single requests for source peripheral
- SglReqDstReg (64 bits) – Generates DMA single requests for destination peripheral
- LstSrcReg (64 bits) – Generates DMA last requests for source peripheral
- LstDstReg (64 bits) – Generates DMA last requests for destination peripheral

A DMA request can be generated for a source or destination peripheral by writing 1 to the "request" and "request write enable" bits of the respective channel bits of the previous registers. For the exact register bit encoding of these software handshaking interface registers, refer to "Software Handshaking Registers" in this databook and to the PrimeCell DMAC databook for the PrimeCell registers.

D.2.3 Register Interface Mapping

This section provides the mapping of the ARM PrimeCell DMAC handshaking registers to DW_ahb_dmac software handshaking registers when the peripheral (source or destination is flow controller) is not the flow controller and when the peripheral is the flow controller.

D.2.3.1 Peripheral is Not Flow Controller

Table D-2 describes the mapping of the DW_ahb_dmac software handshaking registers with respect to the PrimeCell DMAC handshaking registers when the peripheral is not the flow controller. The mapping is true for the source peripheral when it is not the flow controller, and the destination peripheral when it is not the flow controller.

Table D-2 Mapping of PrimeCell Handshaking Registers to DW_ahb_dmac DMA is Flow Controller

D.2.3.2 Peripheral is Flow Controller

Burst transaction outside Single Transaction Region	Write to DMACSoftBReq	SglReqSrcReg/ SglReqDstReg followed by ReqSrcReg/ ReqDstReg OR ReqSrcReg/ ReqDstReg followed by SglReqSrcReg/ SglReqDstReg	When converting from PrimeCell programming for writes to DMACSoftBReq, a corresponding write is required to ReqSrcReg or ReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. Writing a 1 to the ReqSrcReg[x]/ReqDstReg[x] register is always interpreted as a burst transaction request, where *x* is the channel number. However, in order for a burst transaction request to start, software must write a 1 to the SglReqSrcReg[x]/SglReqDstReg[x] register.
Single transaction Inside the Single Transaction Region	Write to DMACSoftSReq	Write to SglReqSrcReg/ SglReqDstReg	When converting from PrimeCell programming for writes to DMACSoftSReq, a corresponding write is required on SglReqSrcReg or SglReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. This is only true when a block transfer to the peripheral is in the Single Transaction Region.

provides the mapping of the DW_ahb_dmac software handshaking registers with respect to the PrimeCell DMAC handshaking registers when (source or destination) peripheral is the flow controller.

Table D-3 Mapping of PrimeCell Handshaking Registers to DW_ahb_dmac Peripheral is Flow Controller

Operation	ARM PrimeCell	DW_ahb_dmac	Notes
Burst transaction	DMACSoftBReq	SglReqSrcReg/ SglReqDstReg followed by ReqSrcReg/ ReqDstReg	When converting from PrimeCell programming for writes to DMACSoftBReq, a corresponding write is required on ReqSrcReg or ReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. This initiates a transaction. The type of transaction, single or burst, depends on the value in the corresponding SglReqSrcReg/ SglReqDstReg register.If it is 0, it is a burst transaction.
Single transaction	DMACSoftSReq	SglReqSrcReg/ SglReqDstReg followed by ReqSrcReg/ ReqDstReg	When converting from PrimeCell programming for writes to DMACSoftSReq, a corresponding write is required on ReqSrcReg or ReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. This initiates a transaction. The type of transaction, single or burst, depends on the value in the corresponding SglReqSrcReg/SglReqDstReg register. If it is 1, it is a single transaction.

Table D-3 Mapping of PrimeCell Handshaking Registers to DW_ahb_dmac Peripheral is Flow Controller

(Continued)

Operation	ARM PrimeCell	DW_ahb_dmac	Notes
Last burst transaction	DMACSoftLBReq	LstSrcReg/LstDstReg followed by SglReqSrcReg/ SglReqDstReg followed by ReqSrcReg/ ReqDstReg[a]	When converting from PrimeCell programming for writes to DMACSoftLBReq, a corresponding write is required on ReqSrcReg or ReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. This initiates a transaction. The type of transaction, single or burst, depends on the value in the corresponding SglReqSrcReg/SglReqDstReg register. If it is 0, it is a burst transaction. Futhermore, the transaction is the last transaction in the block if the corresponding bit in the LstSrcReg/LstDstReg register is 1.
Last single transaction	DMACSoftLSReq	LstSrcReg/LstDstReg followed by SglReqSrcReg/ SglReqDstReg followed by ReqSrcReg/ReqDstReg[a]	When converting from PrimeCell programming for writes to DMACSoftLSReq, a corresponding write is required on ReqSrcReg or ReqDstReg in DW_ahb_dmac, depending on the bit location (source or destination peripheral) in the DMACSoftBReq register. This initiates a transaction. The type of transaction, single or burst, depends on the value in the corresponding SglReqSrcReg/SglReqDstReg register. If it is 1, it is a single transaction. Futhermore, the transaction is the last transaction in the block if the corresponding bit in the LstSrcReg/LstDstReg register is 1.

a. The order before the write to the ReqSrcReg/ReqDstReg register is not important.

E
Configuring DW_ahb_dmac to Match Arm PrimeCell PL080/PL081

This appendix provides the configuration parameter settings for DW_ahb_dmac so that it matches or is equivalent to the Arm PrimeCell PL080 and PL081 devices.

E.1 ARM PL080 Equivalent

The ARM PL080 equivalent is a two-master, eight-channel device. The channel parameter settings are the same for all channels, so only one set of parameters is listed for channel *x*, where *x* = 0 to 7. The configuration parameters for the PL080 equivalent are described in Table E-1.

E.2 ARM PL081 Equivalent

The PL081 Arm equivalent is a one-master, two-channel device. The parameter settings are the same as the PL080, except for the following two parameters:

- DMAH_NUM_MASTER_INT = 1
- DMAH_NUM_CHANNELS = 2

Again, the channel parameters are the same for all channels, so only one set of parameters is listed for channel *x*, where *x* = 0 to 2. For the configuration settings for the PL081 equivalent (except for number of masters and number of channels), refer to Table E-1.

For more information about the configuration parameters for DW_ahb_dmac, refer to “Parameter Descriptions” on page 117.

Table E-1 ARM PL080 Equivalent Configuration Settings

DW_ahb_dmac Configuration Parameter	Setting
DMAH_NUM_MASTER_INT	2
DMAH_NUM_CHANNELS	8
DMAH_NUM_HS_INT	16
DMAH_MABRST	0

DW_ahb_dmac Configuration Parameter	Setting
DMAH_RETURN_ERR_RESP	0
DMAH_INTR_POL	1
DMAH_INTR_IO	1
DMAH_BIG_ENDIAN	0
DMAH_S_HDATA_WIDTH	32
DMAH_M1_HDATA_WIDTH	32
DMAH_M2_HDATA_WIDTH	32
DMAH_M1_AHB_LITE	0
DMAH_M2_AHB_LITE	0
DMAH_ID_NUM	0x02080901
Channel Register (x = 0 to 7)	
DMAH_CHx_FIFO_DEPTH	16
DMAH_CHx_STAT_DST	0
DMAH_CHx_STAT_SRC	0
DMAH_CHx_MAX_MULT_SIZE	256
DMAH_CHx_MAX_BLK_SIZE	4095
DMAH_CHx_FC	3
DMAH_CHx_LOCK_EN	0
DMAH_CHx_SMS	4
DMAH_CHx_DMS	4
DMAH_CHx_LMS	4
DMAH_CHx_STW	0
DMAH_CHx_DTW	0
DMAH_CHx_SRC_NON_OK	1
DMAH_CHx_DST_NON_OK	1
DMAH_CHx_LLP_NON_OK	1
DMAH_CHx_MULTI_BLK_EN	1
DMAH_CHx_HC_LLP	0
DMAH_CHx_SRC_GAT_EN	0

DW_ahb_dmac Configuration Parameter	Setting
DMAH_CHx_DST_SCA_EN	0
DMAH_CHx_MULTI_BLK_TYPE	8
DMAH_CHx_CTL_WB_EN	0

F
Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
activity	A set of functions in coreConsultant that step you through configuration, verification, and synthesis of a selected core.
AHB	Advanced High-performance Bus — high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces (Arm® Limited specification).
AMBA	Advanced Microcontroller Bus Architecture — a trademarked name by Arm® Limited that defines an on-chip communication standard for high speed microcontrollers.
APB	Advanced Peripheral Bus — optimized for minimal power consumption and reduced interface complexity to support peripheral functions (Arm® Limited specification).
APB bridge	DW_apb submodule that converts protocol between the AHB bus and APB bus.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
arbiter	AMBA bus submodule that arbitrates bus activity between masters and slaves.
BFM	Bus-Functional Model — A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. See also Full-Functional Model.
big-endian	Data format in which most significant byte comes first; normal order of bytes in a word.
blocked command stream	A command stream that is blocked due to a blocking command issued to that stream; see also command stream, blocking command, and non-blocking command.
blocking command	A command that prevents a testbench from advancing to next testbench statement until this command executes in model. Blocking commands typically return data to the testbench from the model.

bus bridge	Logic that handles the interface and transactions between two bus standards, such as AHB and APB. See APB bridge.
command channel	Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.
command stream	The communication channel between the testbench and the model.
component	A generic term that can refer to any synthesizable IP or verification IP in the DesignWare Library. In the context of synthesizable IP, this is a configurable block that can be instantiated as a single entity (VHDL) or module (Verilog) in a design.
configuration	The act of specifying parameters for a core prior to synthesis; can also be used in the context of VIP.
configuration intent	Range of values allowed for each parameter associated with a reusable core.
core	Any configurable block of synthesizable IP that can be instantiated as a single entity (VHDL) or module (Verilog) in a design. Core is the preferred term for a big piece of IIP. Anything that requires coreConsultant for configuration, as well as anything in the DesignWare Cores library, is a core.
core developer	Person or company who creates or packages a reusable core. All the cores in the DesignWare Library are developed by Synopsys.
core integrator	Person who uses coreConsultant or coreAssembler to incorporate reusable cores into a system-level design.
coreAssembler	Synopsys product that enables automatic connection of a group of cores into a subsystem. Generates RTL and gate-level views of the entire subsystem.
coreConsultant	A Synopsys product that lets you configure a core and generate the design views and synthesis views you need to integrate the core into your design. Can also synthesize the core and run the unit-level testbench supplied with the core.
coreKit	An unconfigured core and associated files, including the core itself, a specified synthesis methodology, interfaces definitions, and optional items such as verification environment files and core-specific documentation.
cycle command	A command that executes and causes HDL simulation time to advance.
decoder	Software or hardware subsystem that translates from and “encoded” format back to standard format.
design context	Aspects of a component or subsystem target environment that affect the synthesis of the component or subsystem.
design creation	The process of capturing a design as parameterized RTL.
Design View	A simulation model for a core generated by coreConsultant.
DesignWare Synthesizable Components	The Synopsys name for the collection of AMBA-compliant coreKits and verification models delivered with DesignWare and used with coreConsultant or coreAssembler to quickly build DesignWare Synthesizable Component designs.

DesignWare cores	A specific collection of synthesizable cores that are licensed individually. For more information, refer to www.synopsys.com/designware.
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the DesignWare Synthesizable Components.
dual role device	Device having the capabilities of function and host (limited).
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Mode	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
GPIO	General Purpose Input Output.
GTECH	A generic technology view used for RTL simulation of encrypted source code by non-Synopsys simulators.
hard IP	Non-synthesizable implementation IP.
HDL	Hardware Description Language – examples include Verilog and VHDL.
IIP	Implementation Intellectual Property — A generic term for synthesizable HDL and non-synthesizable “hard” IP in all of its forms (coreKit, component, core, MacroCell, and so on).
implementation view	The RTL for a core. You can simulate, synthesize, and implement this view of a core in a real chip.
instantiate	The act of placing a core or model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant byte comes first.
MacroCell	Bigger IP blocks (6811, 8051, memory controller) available in the DesignWare Library and delivered with coreConsultant.
master	Device or model that initiates and controls another device or peripheral.
model	A Verification IP component or a Design View of a core.
monitor	A device or model that gathers performance statistics of a system.
non-blocking command	A testbench command that advances to the next testbench statement without waiting for the command to complete.
peripheral	Generally refers to a small core that has a bus connection, specifically an APB interface.

RTL	Register Transfer Level. A higher level of abstraction that implies a certain gate-level structure. Synthesis of RTL code yields a gate-level design.
SDRAM	Synchronous Dynamic Random Access Memory; high-speed DRAM adds a separate clock signal to control signals.
SDRAM controller	A memory controller with specific connections for SDRAMs.
slave	Device or model that is controlled by and responds to a master.
SoC	System on a chip.
soft IP	Any implementation IP that is configurable. Generally referred to as synthesizable IP.
static controller	Memory controller with specific connections for Static memories such as asynchronous SRAMs, Flash memory, and ROMs.
subsystem	In relation to coreAssembler, highest level of RTL that is automatically generated.
synthesis intent	Attributes that a core developer applies to a top-level design, ports, and core.
synthesizable IP	A type of Implementation IP that can be mapped to a target technology through synthesis. Sometimes referred to as Soft IP.
technology-independent	Design that allows the technology (that is, the library that implements the gate and via widths for gates) to be specified later during synthesis.
Testsuite Regression Environment (TRE)	A collection of files for stand-alone verification of the configured component. The files, tests, and functionality vary from component to component.
VIP	Verification Intellectual Property — A generic term for a simulation model in any form, including a Design View.
workspace	A network location that contains a personal copy of a component or subsystem. After you configure the component or subsystem (using coreConsultant or coreAssembler), the workspace contains the configured component/subsystem and generated views needed for integration of the component/subsystem at the top level.
wrap, wrapper	Code, usually VHDL or Verilog, that surrounds a design or model, allowing easier interfacing. Usually requires an extra, sometimes automated, step to create the wrapper.
zero-cycle command	A command that executes without HDL simulation time advancing.

Index

A

Accessing registers 332
active command queue
    definition 427
activity
    definition 427
AHB
    definition 427
AHB master interface
    arbitration for 98
AMBA
    definition 427
APB
    definition 427
APB bridge
    definition 427
application design
    definition 427
arbiter
    definition 427
Arbitration, for AHB master interface 98
Auto-reloading, of channel registers 337

B

BFM
    definition 427
big-endian
    definition 427
Block
    chaining, about 333
    size 35
Block diagram, of DW_ahb_dmac 21
blocked command stream
    definition 427
blocking command
    definition 427
bus bridge
    definition 428
Bus locking, about 96

C

Channel
    locking
        about 96
        and deadlock 409
        levels of 97
    registers
        auto-reloading of 337
Coherency
    about 379
    read 385
    write 379
command channel
    definition 428
command stream
    definition 428
component
    definition 428
configuration
    definition 428
configuration intent
    definition 428
Configuration parameters
    ARM PrimeCell equivalents 423
core
    definition 428
core developer
    definition 428
core integrator
    definition 428
coreAssembler
    definition 428
coreConsultant
    definition 428
coreKit

definition 428
Customer Support 16
cycle command
definition 428

D

Deadlock, and channel locking 409
decoder
definition 428
design context
definition 428
design creation
definition 428
Design View
definition 428
DesignWare cores
definition 429
DesignWare Library
definition 429
DesignWare Synthesizable Components
definition 428
Destination
burst transaction size 35
peripheral
and transaction requests 96
single transaction size 35
Disabling DMA channel
prior to transfer completion 367
DMA transfers
hierarchy of 25
locked 96
dual role device
definition 429
DW_ahb_dmac
block diagram of 21
features of 22, 23
functional description of 21, 33
programming of 331
testbench
overview of 370
overview of tests 369

E

Early-terminated burst transaction 39
endian
definition 429
Environment, licenses 31

F

FIFO, readiness of 94
Flow control
about 33
configurations of 88
Full-Functional Mode
definition 429
Functional description 21

G

Generating
hardware handshaking signals 45
requests 94
GPIO
definition 429
GTECH
definition 429

H

Handshaking interface
overview 34
hard IP
definition 429
Hardware handshaking interface
generating signals for 45
mapping DW_ahb_dmac to PrimeCell 415
peripheral is flow controller 49
peripheral not flow controller 40
HDL
definition 429

I

IIP
definition 429
implementation view
definition 429
instantiate
definition 429
interface
definition 429
IP
definition 429

L

Licenses 31
little-endian
definition 429

M

MacroCell
  definition 429
master
  definition 429
Memory peripherals 36
model
  definition 429
monitor
  definition 429

N

non-blocking command
  definition 429

P

peripheral
  definition 429
Peripheral burst transaction request, about 90
Peripheral interrupt request interface
  about 87
Peripheral interrupt request interface, about 87
PrimeCell
  equivalent configuration parameters 423
  mapping hardware handshaking interface to DW_ahb_dmac 415
  mapping software handshaking registers to DW_ahb_dmac 417

R

Read coherency
  about 385
  and asynchronous clocks 387
  and synchronous clocks 386
Register access 332
Requesting, DMA transfers 34
RTL
  definition 430

S

SDRAM
  definition 430
SDRAM controller
  definition 430
Setting up, transfers 52
Single transactions
  peripheral is flow controller 52
  peripheral not flow controller 47
slave
  definition 430
SoC
  definition 430
SoC Platform
  AHB contained in 19
  APB, contained in 19
  defined 19
soft IP
  definition 430
Software handshaking interface
  about 36
  mapping PrimeCell to DW_ahb_dmac 417
  peripheral as flow controller 51
  peripheral is flow controller 51
  peripheral not flow controller 46
  registers 37
Source
  burst transaction size 35
  single transaction size 35
static controller
  definition 430
subsystem
  definition 430
synthesis intent
  definition 430
synthesizable IP
  definition 430

T

technology-independent
  definition 430
Testsuite Regression Environment (TRE)
  definition 430
Transaction requests
  from destination peripheral 96
  generating 94
Transactions
  early-terminated burst 39
  single
    peripheral is flow controller 52
    peripheral not flow controller 47
Transfers
  operation of 53
  setting up 52
TRE
  definition 430

V

Vera, overview of tests 369
Verification
  and Vera tests 369
VIP
  definition 430

W

workspace
  definition 430
wrap
  definition 430
wrapper
  definition 430
Write coherency
  about 379
  and asynchronous clocks 384
  and identical clocks 381
  and synchronous clocks 382

Z

zero-cycle command
  definition 430