DesignWare Cores DesignWare DW_axi_dmac Databook

DW_axi_dmac – A415-0

Version 1.00a
October 2014

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

www.synopsys.com

Contents

Revision History

This section tracks the significant documentation changes that occur from release-to-release and during a release from version 1.00a-ea01 onward.

Date	Version	Description
October 2014	1.00a	■ 2014.10a GA release ■ Included these sections in the "Verification" chapter: - "Overview of SV-UVM Tests" - "Overview of DW_axi_dmac Testbench"
August 2014	1.00a-ea04	Added the Clock Parameters block in Table 4-1
July 2014	1.00a-ea03	■ Modified the Integration Considerations chapter ■ Modified description for the DMAX_LLI_ENDIAN_SELECTION_PIN_EN parameter
June 2014	1.00a-ea02	■ Modified address range for the Undefined Register Space in Table 6-1. ■ Added two configuration parameters: - DMAX_MSTIF1_OSR_LMT - DMAX_MSTIF2_OSR_LMT ■ Modified reset values for the following registers: - DMAC_CommonReg_IntSignal_EnableReg - DMAC_CommonReg_IntStatus_EnableReg - CHx_IntStatus_EnableReg - CHx_IntSignal_EnableReg
April 2014	1.00a-ea01	■ Changing the Behaviour of Abnormal Channel Abort ■ Channel Abort feature now supported, removing all the Notes related to that. ■ Added a new section in Debug Interface ■ Added Description in section CHx_IntStatusReg ■ Removing theNote from Software Handshaking chapter
March 2014	1.00a-ea00	Initial release

Preface

This databook provides information that you need to interface the DesignWare AXI Central Direct Memory Access (DMA) Controller, referred to as DW_axi_dmac throughout the remainder of this databook. DW_axi_dmac is a highly configurable, highly programmable, high-performance multi-master multi-channel DMA Controller with AXI as the bus interface for data transfer. This component conforms to the AMBA Specification, Revision 2.0 and AMBA AXI Protocol Specification, Version 3.0 and 4.0 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, "Building and Verifying a Component or Subsystem" introduces you to using the DW_axi_dmac within the coreAssembler and coreConsultant tools.
- Chapter 3, "Functional Description" describes the functional operation of the DW_axi_dmac.
- Chapter 4, "Parameters" identifies the configurable parameters supported by the DW_axi_dmac.
- Chapter 5, "Signals" provides a list and description of the DW_axi_dmac signals.
- Chapter 6, "Registers" describes the programmable registers of the DW_axi_dmac.
- Chapter 7, "Programming the DW_axi_dmac" provides information needed to program the configured DW_axi_dmac.
- Chapter 8, "Verification" provides information on verifying the configured DW_axi_dmac.
- Chapter 9, "Integration Considerations" includes information you need to integrate the configured DW_axi_dmac into your design.
- Appendix A, "Glossary" provides a glossary of general terms.

Related Documentation

- *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 above information, using one of the following methods:
  - *For fastest response*, use the SolvNet website. If you fill in your information as explained below, 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 on the link 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_axi_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 above) 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_axi_dmac, which is a highly configurable, highly programmable, high performance, multimaster multichannel DMA Controller with AXI as the bus interface for data transfer.

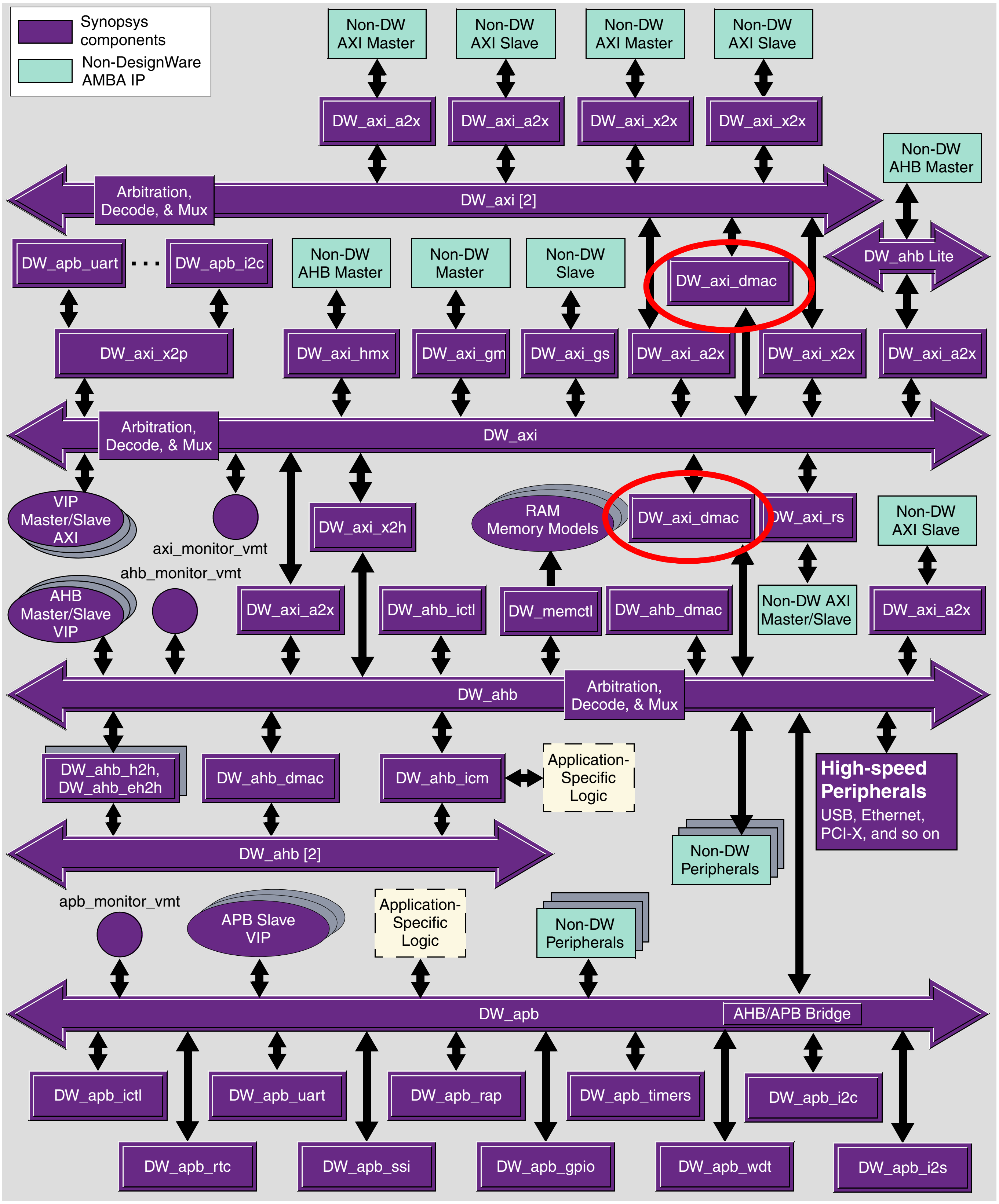
1.1 DesignWare System Overview

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 and 4.0-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.

Attention Links resolve only if you are viewing this databook from your $DESIGNWARE_HOME tree, and to only those components that are installed in the tree.

Figure 1-1 Example of DW_axi_dmac in a Complete System



You can connect, configure, synthesize, and verify the DW_axi_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_axi_dmac component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

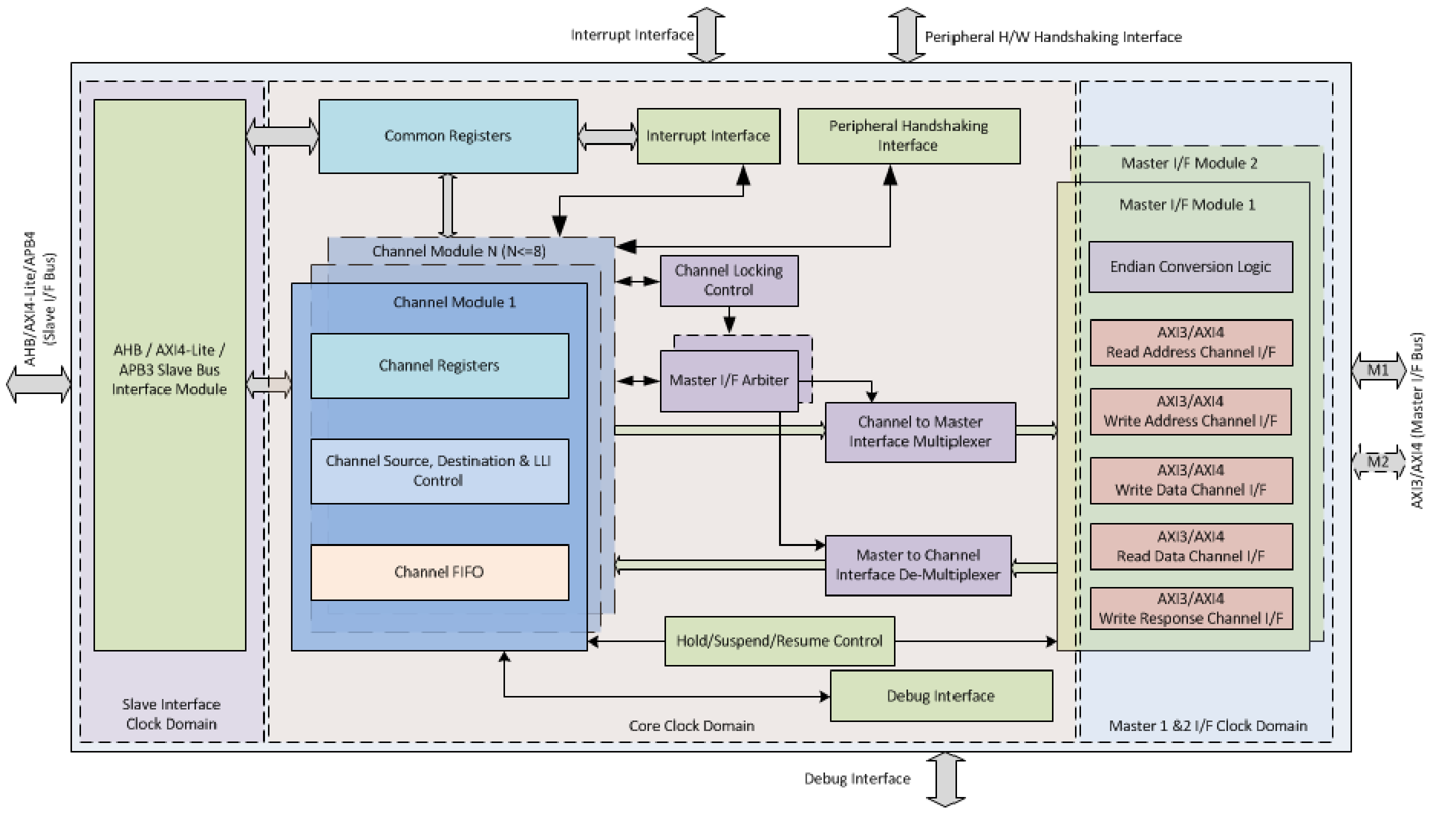
1.2 General Product Description

The Synopsys DW_axi_dmac conforms to the AMBA Specification, Revision 2.0 and the AMBA AXI Protocol Specification, Version 2.0 from ARM.

1.2.1 DW_axi_dmac Block Diagram

Figure 1-2 shows the top-level architecture of the DW_axi_dmac.

Figure 1-2


Interrupt Interface
Peripheral H/W Handshaking Interface
AHB/AXI4-Lite/APB4 (Slave I/F Bus)
AHB / AXI4-Lite / APB3 Slave Bus Interface Module
Slave Interface Clock Domain
Common Registers
Interrupt Interface
Peripheral Handshaking Interface
Channel Module N (N<=8)
Channel Module 1
Channel Registers
Channel Source, Destination & LLI Control
Channel FIFO
Channel Locking Control
Master I/F Arbiter
Channel to Master Interface Multiplexer
Master to Channel Interface De-Multiplexer
Hold/Suspend/Resume Control
Debug Interface
Core Clock Domain
Master I/F Module 2
Master I/F Module 1
Endian Conversion Logic
AXI3/AXI4 Read Address Channel I/F
AXI3/AXI4 Write Address Channel I/F
AXI3/AXI4 Write Data Channel I/F
AXI3/AXI4 Read Data Channel I/F
AXI3/AXI4 Write Response Channel I/F
Master 1 &2 I/F Clock Domain
M1
M2
AXI3/AXI4 (Master I/F Bus)
Debug Interface

1.3 Features

The DW_axi_dmac supports the following features:

1.3.1 General Features

- Independent core, slave interface and master interface clocks
- Shutting down the slave interface clock
  - Master can shut down the slave interface clock when the slvif_busy output is de-asserted.
  - Master must restart the clock before trying to access the slave interface again.
- Individually shutting down the master interface clocks when no peripheral is active
- Up to eight channels, one per source and destination pair, configurable in coreConsultant
- Data transfers in one direction only (each channel is unidirectional)
- Up to two AXI master interfaces, configurable in coreConsultant
  - Two master interfaces for multilayer support
  - Multiple AXI masters increase bus performance by allowing direct connection of peripherals on different AXI interconnects
  - Support for different ACLK on different AMBA layers
- Memory-to-memory, memory-to-peripheral, peripheral-to-memory, and peripheral-to-peripheral DMA transfers
- AMBA 3 AXI/AMBA 4 AXI-compliant master interface
- AHB/AXI4-Lite/APB3 slave interface for programming the DMA controller
  - Only AHB supported in this version
  - AHB slave interface supports only SINGLE transfers (hburst = 3'b000).
- AXI master data bus width up to 512 bits (for both AXI master interfaces), configurable in coreConsultant
- Endian mode can be selected statically or dynamically for AXI master interfaces, configurable in coreConsultant
- Input pin to dynamically select endian scheme
- Independent control for endian scheme of linked list access on master interfaces, configurable in coreConsultant
- Optional identification register, configurable in coreConsultant
- Channel locking support

  Supports locking of the internal channel arbitration for the master bus interface at different transfer hierarchy
- DMAC status indication outputs

  Idle/busy indication

- DMA hold function
- Output pin indicates the last write transfer at DMA transaction level
- Multiple levels of DMA transfer hierarchy

  DMA transfer split into transaction, block, and complete DMA transfer levels

1.3.2 Channel Buffering

- Single FIFO per channel
- FIFO depth, configurable in coreConsultant
- Automatic packing/unpacking of data to fit FIFO width

1.3.3 Channel Control

- Programmable transfer type for each channel (memory-to-memory, memory-to-peripheral, peripheral-to-memory and peripheral-to-peripheral)
- Single or multiple DMA transactions
- Programmable multiple transaction size for each channel
- Programmable maximum AMBA burst transfer size for each channel, configurable in coreConsultant
- Channel disabling without data loss
- Channel suspend and resume
- Programmable channel priority
- Locking of internal channel arbitration for master bus interface at different transfer hierarchy
- Programmable multiblock transfer using linked list, contiguous address, auto reload, and shadow register methods
- Dynamic extension of linked list
- Independent configuration of SRC/DST multiblock transfer type
- Multiple state machines, one for each channel SRC and DST
- Separate state machines for data and LLI access
- Control signals, such as cache and protection, programmable per DMA block
- Programmable transfer length (block length)
- Error status register to ease debugging during error events

1.3.4 Flow Control

- Programmable flow control at DMA transfer level
  - If the size of the block transfer is known prior to DMA initialization, the DMA controller is a flow controller at the DMA block transfer level.
  - If the size of the DMA block transfer is unknown prior to the DMA initialization Peripheral, either source or destination is the flow controller for undefined length (demand mode) DMA block transfers.

1.3.5 Handshaking Interface

- Programmable software and hardware handshaking interfaces for non-memory peripherals
- Up to 16 hardware handshaking interfaces/peripherals, configurable in coreConsultant
- Enabling/disabling of individual handshake interfaces
- Programmable mapping between peripherals and channels; many-to-one mapping with only one peripheral active at a time
- Memory mapped registers to control DMA transfer in software handshaking mode

1.3.6 Interrupt Outputs

- Combined and separate interrupt outputs
- Interrupt generation on
  - DMA transfer completion
  - Block transfer completion
  - Single or multiple transaction completion
  - Error condition
  - Channel suspend or disable
- Interrupt enabling and masking

1.3.7 Bus Interface

- AMBA 3 AXI and AMBA 4 AXI protocols for master interface and AHB, AXI4-Lite, and APB 3 protocols for slave interface
- Data bus width up to 512 bits for master interface, configurable in coreConsultant
- Outstanding transactions on master interface
- Setting outstanding transaction limit per channel on the master interface
- Configurable AXI transfer width
- Out-of-order transaction support for different channels connected on same master interface

  Transactions of a particular channel are always initiated in order.

- Increment and fixed address transfers on master interface
- Source and destination data transfer addresses; must be aligned to respective transfer widths
- Data bus width of 32/64 bits for slave interface, configurable in coreConsultant
- Transfer size (width) used for slave interface; must be same as data bus width

1.3.8 Unsupported Features

The following features are not supported in this release:

- AXI 3 locked transfers (bus locking)
- Bus locking on master interface in AXI 3 mode
- Different FIFO modes to reduce latency and increase bus utilization
- Wrap address transfers
- AXI unaligned transfers
- Narrow transfers

1.4 Terminology

The following terms are concise definitions of the DMA concepts used throughout this document.

- **Source peripheral** – Device on an AXI layer from which the DW_axi_dmac reads data. The DW_axi_dmac then stores the data in the channel FIFO. The source peripheral teams up with a destination peripheral to form a channel.
- **Destination peripheral** – Device to which the DW_axi_dmac writes the stored data from the FIFO (data is previously read from the source peripheral).
- **Memory** – Source or destination that is always ready for a DMA transfer and does not require a handshaking interface to interact with the DW_axi_dmac.
- **Channel** – Read/write data path between a source peripheral on one configured AXI layer and a destination peripheral on the same or a different AXI 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_axi_dmac is a master on the AXI bus, reading data from the source and writing it to the destination over the AXI bus. It is possible to have up to two master interfaces, so that up to two independent source and destination channels can operate simultaneously. Each channel has to arbitrate for the master interface. If the source and destination peripherals reside on different AXI layers, there must be multiple master interfaces.
- **Slave interface** – The AHB/AXI4-Lite/APB3 interface over which the DW_axi_dmac is programmed. The slave interface 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 to perform a handshake between the DW_axi_dmac and the source or destination peripheral. A handshake helps to control the transfer of a single or burst transaction between the DW_axi_dmac and the peripheral. This interface is used to request, acknowledge, and control a DW_axi_dmac transaction. A channel can receive a request through one of two types of handshaking interface: hardware or software.
  - **Hardware handshaking interface** – Uses hardware signals to control the transfer of a single or burst transaction between the DW_axi_dmac and the source or destination peripheral. The simple use of the hardware handshaking interface is when the interrupt line from the peripheral is tied to the "dma_req" input of the hardware handshaking interface; other interface signals are ignored.
  - **Software handshaking interface** – Uses software registers to control transferring a single or burst transaction between the DW_axi_dmac and the source or destination peripheral. In this mode, no special DW_axi_dmac handshaking signals are needed on the I/O of the peripheral. This mode is useful for interfacing an existing peripheral to the DW_axi_dmac without modifying it.
- **Flow controller** – Device (either the DW_axi_dmac, or a source or destination peripheral) that determines the length of a DMA block transfer and terminates it. If the length of a block is known before enabling the channel, then the DW_axi_dmac should be programmed as the flow controller. If the length of a block is not known prior to enabling the channel, the source or destination peripheral should terminate the block transfer. In this mode, the peripheral (source/destination) is the flow controller.
- **Transfer hierarchy** – Transfers are split into a maximum of four levels: DMA transfer level, block transfer level, transaction level, and AXI transfer level. This is done to minimize the effect of the scenario where the channel is granted to a particular set of peripherals, but a peripheral doesn't have enough data to transfer continuously. In such a scenario, the channel can't be given to any other peripheral, which reduces the performance.

  Figure 1-3 illustrates the hierarchy between DMA transfers, block transfers, transactions (single or burst), and AMBA AXI transfers (single or burst) for non-memory peripherals.

  Figure 1-4 illustrates the hierarchy between DMA transfers, block transfers, and AMBA AXI transfers (single or burst) for memory peripherals.

Note There is no transaction level for memory peripherals, as a memory is assumed to be always ready for data transfer.

Figure 1-3 DMA Transfer Hierarchy for Non-Memory Peripherals

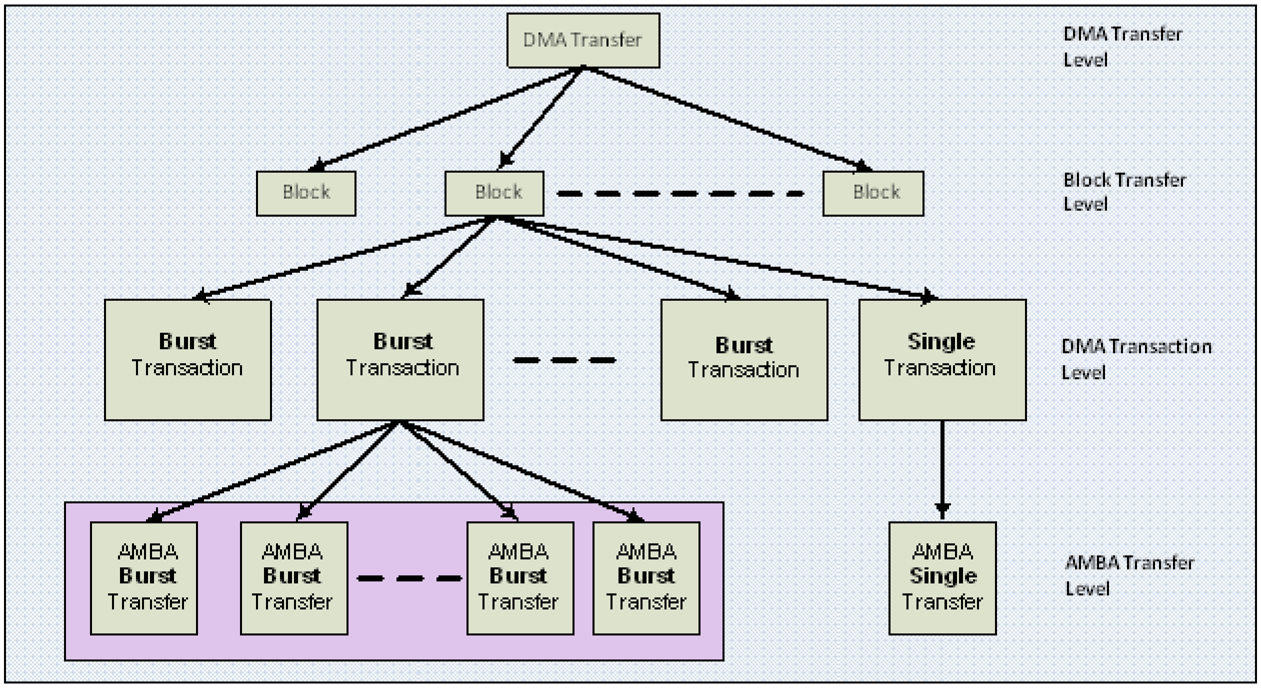
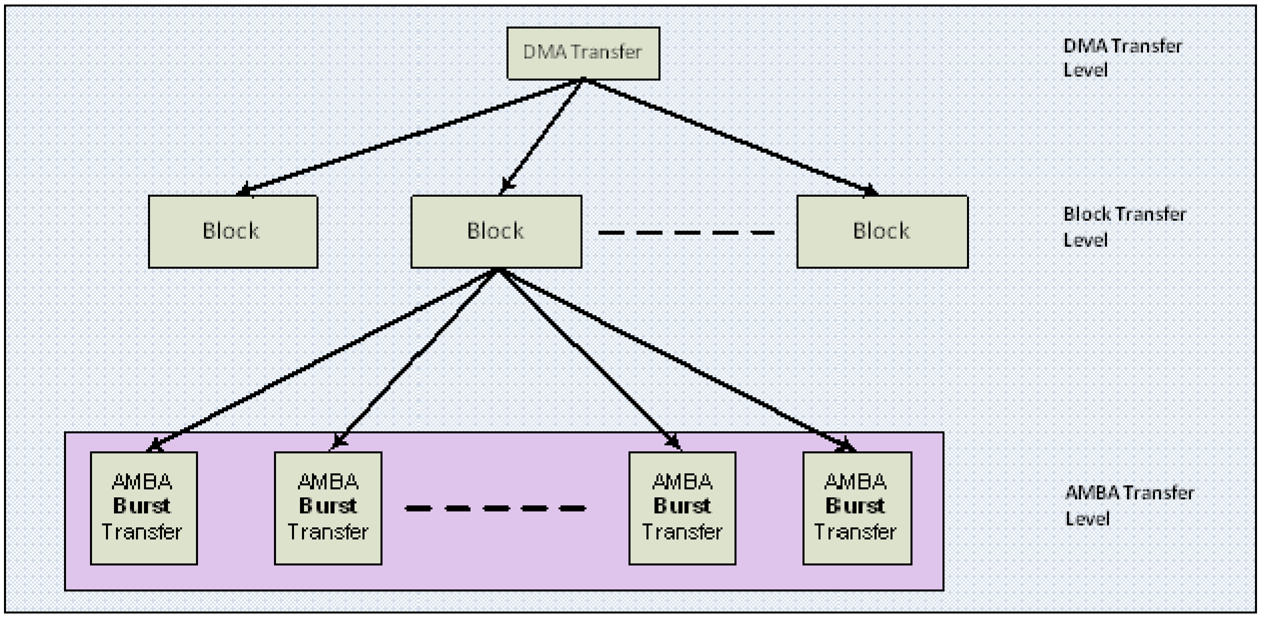


Figure 1-4 DMA Transfer Hierarchy for Memory Peripherals



- **Transaction** – Basic unit of a DW_axi_dmac transfer, as determined by either the hardware or the software handshaking interface. A transaction is relevant only for transfers between the DW_axi_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 it is converted to an INCR AXI transfer of burst length 1.
    - **Burst transaction** – Length of a burst transaction is programmed into the DW_axi_dmac. A burst transaction is converted into a sequence of AXI burst transfers. The burst transaction length is under program control and normally bears some relationship to the FIFO sizes in the DW_axi_dmac and in the source and destination peripherals.
- **Block** – A block of DW_axi_dmac data, the amount of which is the block length and is determined by the flow controller. For transfers between the DW_axi_dmac and memory, a block is broken directly into a sequence of burst transfers. For transfers between the DW_axi_dmac and a non-memory peripheral, a block is broken into a sequence of DW_axi_dmac transactions. These are in turn broken into a sequence of AXI transfers.
- **DMA transfer** – Software controls the number of blocks in a DW_axi_dmac transfer. Once the DMA transfer has completed, the hardware within the DW_axi_dmac disables the channel and can generate an interrupt to signal the DMA transfer completion. The channel can then be reprogrammed for a new DMA transfer.
    - **Single-block DMA transfer** – Consists of a single block.
    - **Multi-block DMA transfer** – A DMA transfer may consist of multiple DMA blocks. Multi-block DMA transfers are supported through block chaining (linked list pointers), auto-reloading of channel registers, shadow registers, and contiguous blocks. The source and destination can independently select which method to use.
        - **Linked lists (block chaining)** – A 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_axi_dmac fetches the LLI at the beginning of every block when block chaining is enabled. LLI access always uses the burst size (arsize/awsize) that is the same as the data bus width and cannot be changed or programmed to anything other than this. Burst length (awlen/arlen) is chosen based on the data bus width so that the access does not cross one complete LLI structure of 64 bytes. DW_axi_dmac fetches the entire LLI (40 bytes) in one AXI burst, if the burst length is not limited by other settings.
        - **Auto-reloading** – DW_axi_dmac automatically reloads the channel registers at the end of each block to the value that was set when the channel was first enabled.
        - **Contiguous blocks** – Address between successive blocks is selected to be a continuation from the end of the previous block.
        - **Shadow register** - DW_axi_dmac automatically loads the channel registers from the contents of shadow register set at the end of each block. Software can program the shadow registers with the values corresponding to the next block transfer when the current block transfer is in progress.
- **Channel locking** – Software can program a channel to keep the AXI master interface by locking arbitration of the master bus interface for the duration of a DMA transfer, block transfer, or transaction (single or burst).

1.5 Standards Compliance

The Synopsys DW_axi_dmac conforms to the AMBA Specification, Revision 2.0 and the AMBA AXI Protocol Specification, Version 2.0 from ARM. Readers are assumed to be familiar with this specification.

1.6 Verification Environment Overview

The DW_axi_dmac must support an extensive verification environment, which sets up and invokes the selected simulation tool to execute tests that verify the functionality of the configured component and allows you to analyze the results of the simulation.

"Verification" on page 201 discusses the specific procedures for verifying the DW_axi_dmac.

1.7 Licenses

Before you begin using the DW_axi_dmac, you must have a valid license. For more information, refer to "Licenses" in the *DesignWare Synthesizable Components for AMBA 2, AMBA 3 AXI, and AMBA 4 AXI Installation Guide.*

1.8 Where To Go From Here

At this point, you may want to get started working with the DW_axi_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, refer to *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_axi_dmac component, refer to "Overview of the coreConsultant Configuration and Integration Process" on page 26.

For more information about implementing your DW_axi_dmac component within a DesignWare subsystem using coreAssembler, refer to "Overview of the coreAssembler Configuration and Integration Process" on page 29.

2 Building and Verifying a Component or Subsystem

DesignWare Synthesizable IP (SIP) components for AMBA 2, AMBA 3 AXI, and AMBA 4 AXI are packaged using Synopsys coreTools, which enable you to configure, synthesize, and run simulations on a single SIP title, or to build a configured AMBA subsystem. You do this by generating a workspace view using one of the following coreTools applications:

- coreConsultant – Used for configuration, RTL generation, synthesis, and execution of packaged verification for a single SIP title. The *coreConsultant User Guide* provides complete information on using coreConsultant.
- coreAssembler – Used for building and configuration of a subsystem that connects multiple SIP titles, RTL generation, synthesis, and creation of a template subsystem testbench. The *coreAssembler User Guide* provides complete information on using coreAssembler.

A workspace is your working version of a DesignWare SIP component or subsystem. In fact, you can create several workspaces to experiment with different design alternatives.

Hint If you are unfamiliar with coreTools—which is comprised of the coreAssembler, coreConsultant, and coreBuilder tools—you can go to *Using DesignWare Library IP in coreAssembler* to "get started" learning how to work with DesignWare SIP components.

2.1 Setting up Your Environment

The DW_axi_dmac is included in a release of DesignWare SIP components. It is assumed that you have already downloaded and installed the release. If you have not, you can download and install the latest versions of required tools using the *DesignWare Synthesizable Components for AMBA 2, AMBA 3 AXI, and AMBA 4 AXI Installation Guide*.

You also need to set up your environment correctly using specific environment variables, such as DESIGNWARE_HOME, VERA_HOME, PATH, and SYNOPSYS. If you are not familiar with these requirements and the necessary licenses, refer to "Setting up Your Environment" in the *DesignWare Synthesizable Components for AMBA 2, AMBA 3 AXI, and AMBA 4 AXI Installation Guide.*

2.2 Overview of the coreConsultant Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on the DW_axi_dmac using coreConsultant.

2.2.1 coreConsultant Usage

Figure 2-1 illustrates some general directories and files in a coreConsultant workspace.

Figure 2-1 coreConsultant Usage Flow

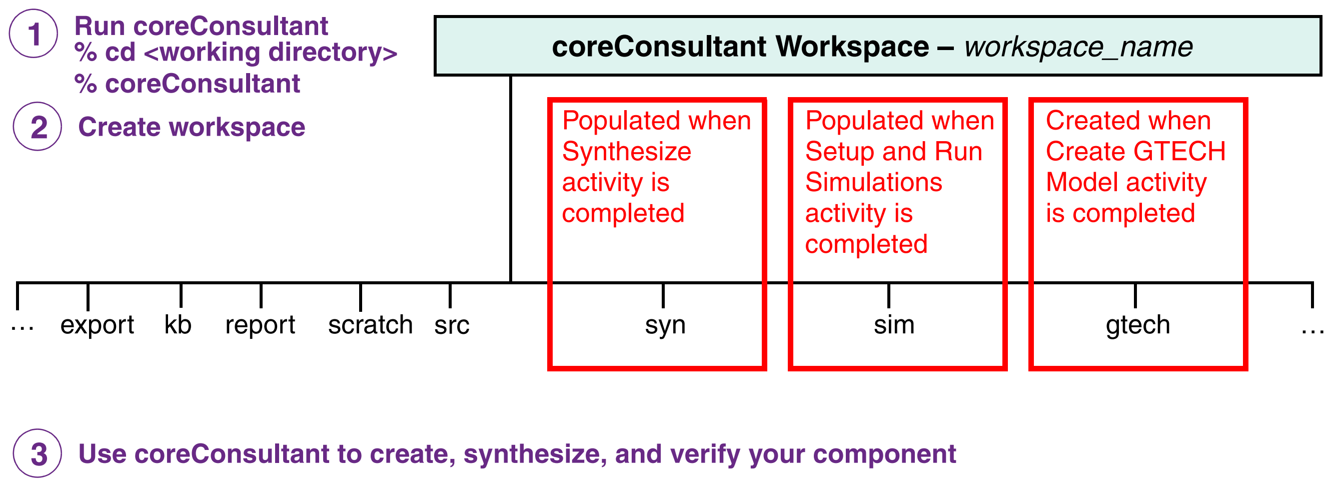

1 Run coreConsultant
% cd <working directory>
% coreConsultant
2 Create workspace
coreConsultant Workspace – workspace_name
Populated when Synthesize activity is completed
Populated when Setup and Run Simulations activity is completed
Created when Create GTECH Model activity is completed
…
export
kb
report
scratch
src
syn
sim
gtech
…
3 Use coreConsultant to create, synthesize, and verify your component

Table 2-1 provides a description of the implementation workspace directory and subdirectories.

Table 2-1 coreConsultant Implementation Workspace Directory Contents

Directory/Subdirectory	Description
auxiliary	Scripts and text files used by coreConsultant. Generated upon first creating workspace.
doc	Contains local copies of component-specific databooks. Generated upon first creating workspace.
export	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreConsultant). Generated upon first creating workspace; populated during Specify Configuration activity.
gtech	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Generate GTECH Model activity.
kb	Contains knowledge base information used by coreConsultant. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.

Table 2-1 coreConsultant Implementation Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
leda	Contains Leda configuration files for the component. Generated upon first creating workspace; updated during Run Leda Coding Checker activity.
pkg	Contains RTL preprocessor scripts. Generated during Specify Configuration activity.
report	Contains all of the reports created by coreConsultant during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
scratch	Contains temp files used during the coreConsultant processes. Generated upon first creating workspace; populated and updated throughout activities.
sim	Contains test stimulus and output files. Generated upon first creating workspace; updated during Setup and Run Simulations activity.
src	Includes the top-level RTL file, *design_name*.v. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated during Specify Configuration activity.
syn	Contains synthesis files for the component. Generated upon first creating workspace; updated during Synthesis activity and Formal Verification activity.
tcl	Contains synthesis intent scripts. Generated upon first creating workspace.

For details on some key files created during coreConsultant activities, refer to "Database Files" on page 32.

For information on using coreConsultant, refer to the *coreConsultant User Guide*.

2.2.2 Configuring the DW_axi_dmac within coreConsultant

"Parameters" on page 81 describes the DW_axi_dmac hardware configuration parameters that you configure using the coreConsultant GUI.

The "Creating the RTL View of a Core" chapter in the *coreConsultant User Guide* discusses how to specify a configuration for an individual component like the DW_axi_dmac.

2.2.3 Creating Gate-Level Netlists within coreConsultant

The "Creating the Gate-Level Netlist for a Core" chapter in the *coreConsultant User Guide* discusses how to create a translation of the RTL view into a technology-specific netlist for an individual component like the DW_axi_dmac.

2.2.4 Verifying the DW_axi_dmac within coreConsultant

"Verification" on page 201 provides an overview of the testbench available for DW_axi_dmac verification using the coreConsultant GUI.

The "Verifying Your Implementation" chapter in the *coreConsultant User Guide* discusses how to simulate an individual component like the DW_axi_dmac.

2.2.5 Running Leda on Generated Code with coreConsultant

When you select **Verify Component > Run Leda Coding Checker** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.3 Overview of the coreAssembler Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on your DesignWare subsystem with coreAssembler.

2.3.1 coreAssembler Usage

Figure 2-2 illustrates some general directories and files in a coreAssembler workspace.

Figure 2-2 coreAssembler Usage Flow

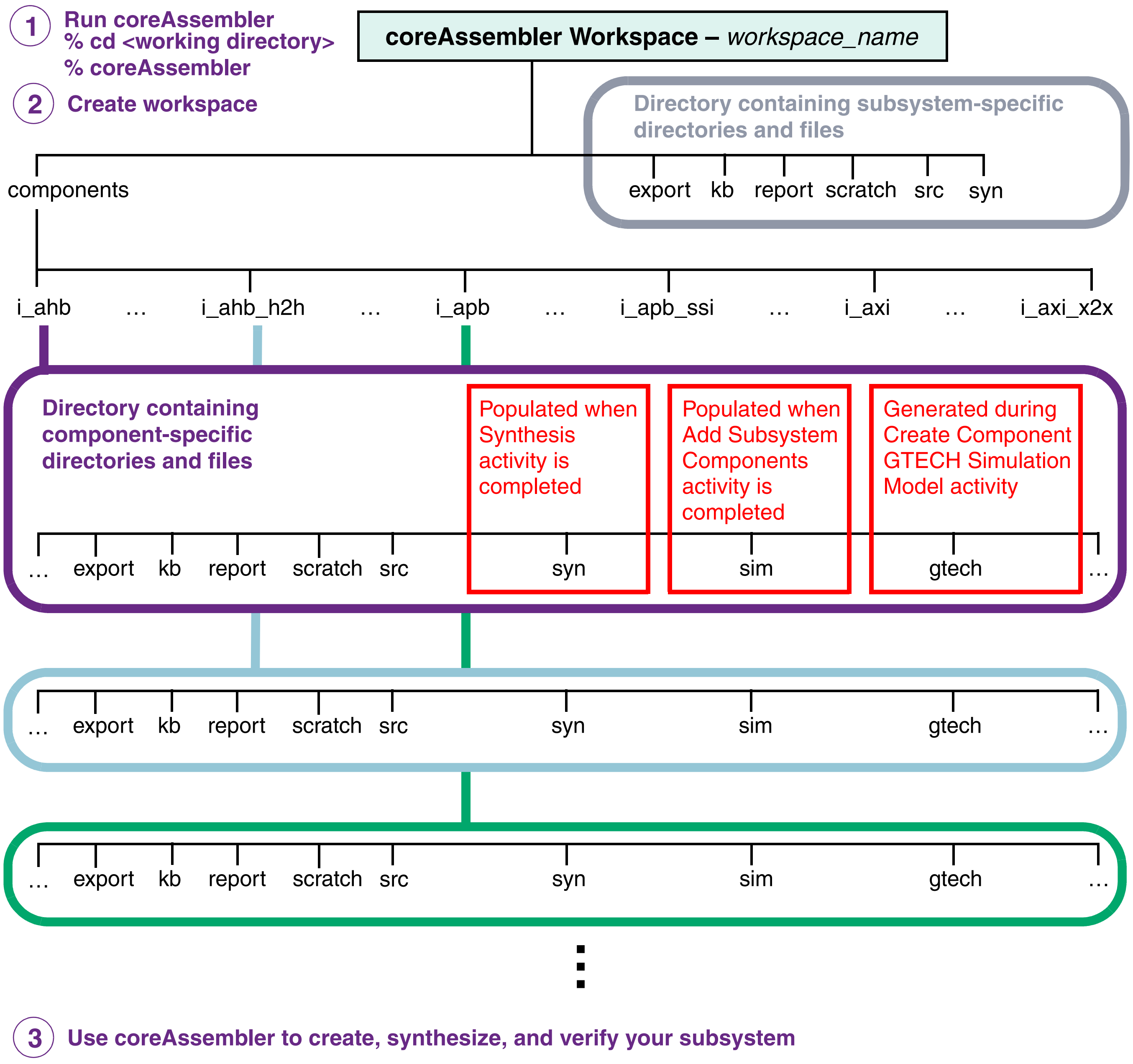

1 Run coreAssembler
% cd <working directory>
% coreAssembler
2 Create workspace
coreAssembler Workspace – workspace_name
Directory containing subsystem-specific directories and files
export kb report scratch src syn
components
i_ahb … i_ahb_h2h … i_apb … i_apb_ssi … i_axi … i_axi_x2x
Directory containing component-specific directories and files
Populated when Synthesis activity is completed
Populated when Add Subsystem Components activity is completed
Generated during Create Component GTECH Simulation Model activity
… export kb report scratch src syn sim gtech …
3 Use coreAssembler to create, synthesize, and verify your subsystem

Table 2-2 provides a description of the implementation workspace directory and subdirectories.

Table 2-2 coreAssembler Implementation Workspace Directory Contents

Directory/Subdirectory	Description
components	Contains a directory for each IP component instance connected in the subsystem. Generated and populated with separate component directories upon first adding components; populated and updated throughout activities.
i_*component*/auxiliary	Scripts and text files used by coreAssembler. Generated during Add Subsystem Components activity.
i_*component*/doc	Contains local copies of component-specific databooks. Generated during Add Subsystem Components activity.
i_*component*/export	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated during Add Subsystem Components activity; populated during Configure Components activity.
i_*component*/gtech	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Create Component GTECH Simulation Model activity.
i_*component*/kb	Contains knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/leda	Contains Leda configuration files for the component. Generated during Add Subsystem Components activity; populated during Run Leda Coding Checker (for /i_*component*) activity.
i_*component*/pkg	Contains RTL preprocessor scripts. Generated during Configure Components activity.
i_*component*/report	Contains all of the reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/scratch	Contains temp files used during the coreAssembler processes. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/sim	Contains test stimulus and output files. Generated during Add Subsystem Components activity; updated during Setup and Run Simulations (for /i_*component*) activity.

Table 2-2 coreAssembler Implementation Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
i_*component*/src	Includes the top-level RTL file, *design_name*.v. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated during Add Subsystem Components activity; populated during Specify Configuration activity.
i_*component*/syn	Contains synthesis files for the component. Generated during Add Subsystem Components activity; updated during Synthesis activity.
i_*component*/tcl	Contains synthesis intent scripts. Generated during Add Subsystem Components activity.
export	Contains subsystem files used to integrate the results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated upon first creating workspace; populated starting with Memory Map Specification activity.
kb	Contains subsystem knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.
report	Contains subsystem reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
scratch	Contains subsystem temp files used during the coreAssembler processes. Generated upon first creating workspace; populated and updated throughout activities.
src	Includes the RTL related to the subsystem. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated starting with Generate Subsystem RTL activity.
syn	Contains synthesis files for the subsystem. Generated upon first creating workspace; updated during Synthesize activity and Formal Verification activity.

For details on some key files created during coreAssembler activities, refer to "Database Files" on page 32.

For information on using coreAssembler, refer to the *coreAssembler User Guide*. For information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI/AMBA 4 AXI components within coreTools, refer to *Using DesignWare Library IP in coreAssembler*.

2.3.2 Configuring the DW_axi_dmac within a Subsystem

"Parameters" on page 81 describes the DW_axi_dmac hardware configuration parameters that you configure using the coreAssembler GUI. Corresponding databooks for the other components in a subsystem contain "Parameters" chapters that describe their respective configuration parameters.

The "Creating the RTL View of a Subsystem" chapter in the *coreAssembler User Guide* discusses how to configure subsystem components and automatically connect them using the coreAssembler GUI.

2.3.3 Creating Gate-Level Netlists within coreAssembler

The "Creating the Gate-Level Netlist for a Subsystem" chapter in the *coreAssembler User Guide* discusses how to create a translation of the RTL view into a technology-specific netlist for a subsystem.

2.3.4 Verifying the DW_axi_dmac within coreAssembler

"Verification" on page 201 provides an overview of the testbench available for DW_axi_dmac verification using the coreAssembler GUI.

The "Verifying Subsystems and Components" chapter in the *coreAssembler User Guide* discusses how to simulate a subsystem.

Note To verify subsystems that contain DW_axi_dmac in the coreAssembler, you must manually connect dmac_core_clock and dmac_core_resetn signals in the subsystem testbench.

2.3.5 Running Leda on Generated Code with coreAssembler

When you select **Verify Component > Run Leda Coding Checker for /i_*component*)** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.4 Database Files

The following subsections describe some key files created in coreConsultant and coreAssembler activities.

2.4.1 Design/HDL Files

The following sections describe the design and HDL files that are produced by coreConsultant and coreAssembler when configuring and verifying a DesignWare Synthesizable Component. The following files are created in different directories by coreConsultant and coreAssembler:

- coreConsultant – *workspace*/ directory
- coreAssembler – *workspace*/components/i_*component*/ directory

2.4.1.1 RTL-Level Files

The following table describes the RTL files that are generated by the Create RTL activity. They are encrypted except where otherwise noted. Any Synopsys synthesis tool or simulator can read encrypted RTL files.

Table 2-3 RTL-Level Files

Files	Encrypted?	Purpose
./src/*component*_cc_constants.v	No	Includes definitions and values of all configuration parameters that you have specified for the component.

./src/*component*.v	No	Top-level HDL file. Include the DesignWare libraries by using the following options in your simulator invocation:
		`+libext+.v+.V`
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`

./src/*component_submodule*.v	Yes	Sub-modules of component
./src/*component*_constants.v	No	Includes the constants used internally in the design.
./src/*component*_undef.v		Includes an undef for each of the definitions found in the *component*_cc_constants.v file; compiled in after the last file listed in ./src/*components*.lst when compiling multiple instances of the same IP.
./src/*component*.lst	No	Lists the order in which the RTL files should be read into tools, such as simulators or dc_shell. For example, use the following option to read the design into VCS: `vcs +v2k -f component.lst`

2.4.1.2 Simulation Model Files

The following table includes files generated for the component during the Generate GTECH Simulation activity. These files are needed when you are using a non-Synopsys simulator (when you can not use the encrypted RTL).

Table 2-4 Simulation Model Files

Files	Encrypted?	Purpose

./gtech/final/db/*component*.v	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist; VHDL and Verilog versions are generated. Include the DesignWare libraries by using the following options in your simulator invocation:
		`+libext+.v+.V`
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`

2.4.2 Synthesis Files

The following table includes files generated after the Create Gate-Level Netlist activity is performed on a component.

Table 2-5 Synthesis Files

Files	Encrypted?	Purpose
./syn/auxScripts	No	Auxiliary files for synthesis.
./syn/final/db/*component*.db	Binary format	Synopsys .db files (gate level) that can be read into dc_shell for further synthesis, if desired.
./syn/final/db/*component*.v	No	Gate-level netlist that is mapped to technology libraries that you specify.
./syn/constrain/script/*.*	No	Constraint files for the components.
./syn/final/report/*.*	No	Synthesis result files.

2.4.3 Verification Reference Files

Files described in the following table include information pertaining to the component's operation so that you can verify installation and configuration of the component has been successful. These files are not for re-use during system-level verification.

Table 2-6 Verification Reference Files

Files	Encrypted?	Purpose
./sim/runtest	No	Perl script that runs the Setup and Run Simulations activity from the command line.
./sim/runtest.log	No	The overall result of simulation, including pass/fail results.
./sim/test_*testname*/test.result	No	Pass/fail of individual test.
./sim/test_*testname*/test.log	No	Log file for individual test.

3 Functional Description

This chapter describes the functional details of the DW_axi_dmac component. DW_axi_dmac is a highly configurable, highly programmable, high-performance multi-master multi-channel DMA Controller with AXI as the bus interface for data transfer.

This chapter included the following topics:

3.1 Data Flow

Figure 3-1 on page 37 shows the flow of data in the DW_axi_dmac in a scenario when the source and destination peripherals are on the same AXI layer, while Figure 3-2 on page 38 shows the flow of data when the source and destination peripherals are on different AXI layers. In both scenarios, the source peripheral uses a hardware handshaking interface and the destination peripheral uses a software handshaking interface. If these peripherals are memory, handshaking interfaces are not used.

Figure 3-1 DW_axi_dmac Flow Diagram when Source and Destination Peripherals on Same AXI Layer

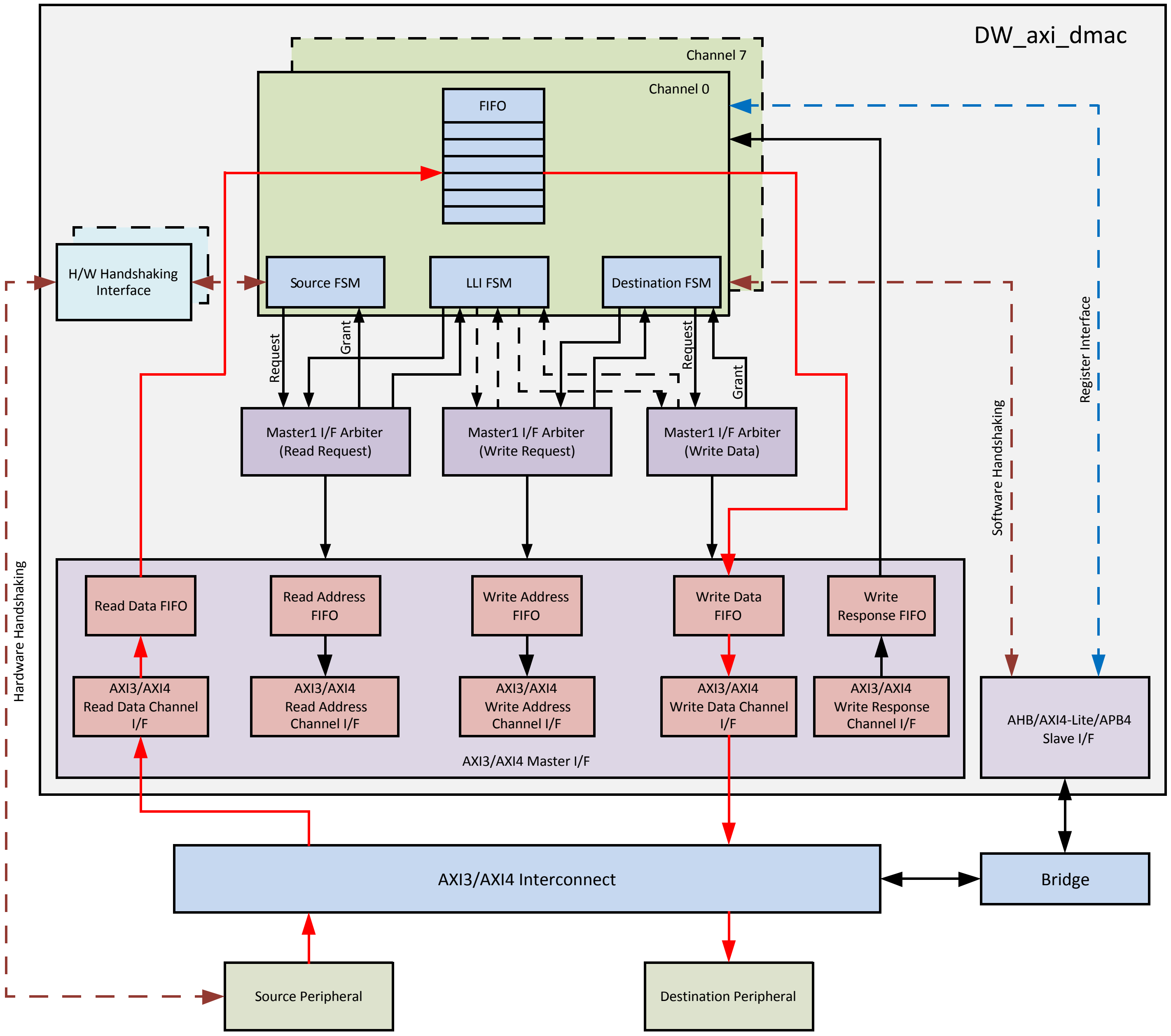
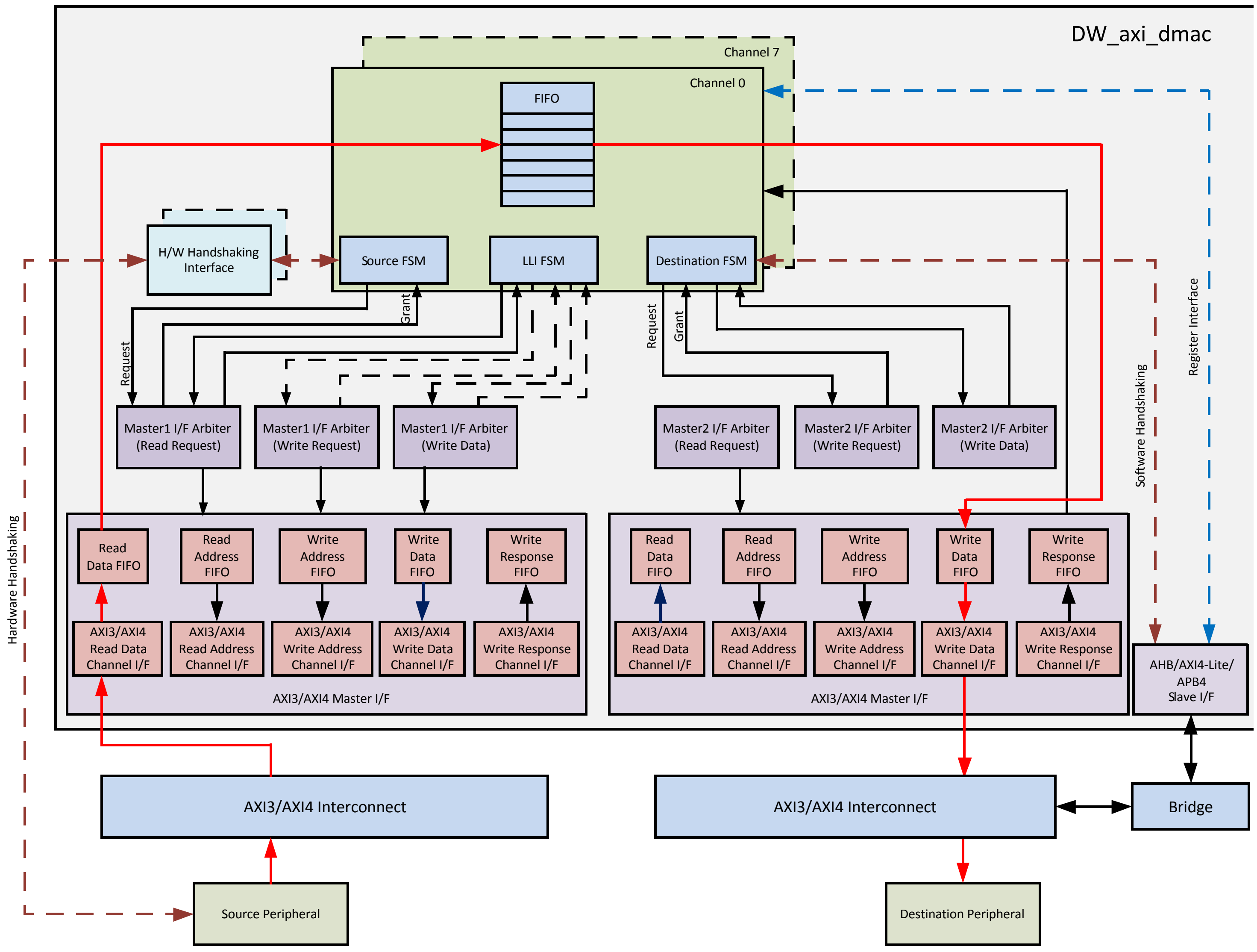

DW_axi_dmac
Channel 7
Channel 0
FIFO
H/W Handshaking Interface
Source FSM
LLI FSM
Destination FSM
Request
Grant
Request
Grant
Master1 I/F Arbiter (Read Request)
Master1 I/F Arbiter (Write Request)
Master1 I/F Arbiter (Write Data)
Software Handshaking
Register Interface
Hardware Handshaking
Read Data FIFO
Read Address FIFO
Write Address FIFO
Write Data FIFO
Write Response FIFO
AXI3/AXI4 Read Data Channel I/F
AXI3/AXI4 Read Address Channel I/F
AXI3/AXI4 Write Address Channel I/F
AXI3/AXI4 Write Data Channel I/F
AXI3/AXI4 Write Response Channel I/F
AXI3/AXI4 Master I/F
AHB/AXI4-Lite/APB4 Slave I/F
AXI3/AXI4 Interconnect
Bridge
Source Peripheral
Destination Peripheral

Figure 3-2 DW_axi_dmac Flow Diagram when Source and Destination Peripherals on Different AXI Layers


DW_axi_dmac
Channel 7
Channel 0
FIFO
H/W Handshaking Interface
Source FSM
LLI FSM
Destination FSM
Request
Grant
Request
Grant
Register Interface
Software Handshaking
Hardware Handshaking
Master1 I/F Arbiter (Read Request)
Master1 I/F Arbiter (Write Request)
Master1 I/F Arbiter (Write Data)
Master2 I/F Arbiter (Read Request)
Master2 I/F Arbiter (Write Request)
Master2 I/F Arbiter (Write Data)
Read Data FIFO
Read Address FIFO
Write Address FIFO
Write Data FIFO
Write Response FIFO
AXI3/AXI4 Read Data Channel I/F
AXI3/AXI4 Read Address Channel I/F
AXI3/AXI4 Write Address Channel I/F
AXI3/AXI4 Write Data Channel I/F
AXI3/AXI4 Write Response Channel I/F
AXI3/AXI4 Master I/F
Read Data FIFO
Read Address FIFO
Write Address FIFO
Write Data FIFO
Write Response FIFO
AXI3/AXI4 Read Data Channel I/F
AXI3/AXI4 Read Address Channel I/F
AXI3/AXI4 Write Address Channel I/F
AXI3/AXI4 Write Data Channel I/F
AXI3/AXI4 Write Response Channel I/F
AXI3/AXI4 Master I/F
AHB/AXI4-Lite/ APB4 Slave I/F
AXI3/AXI4 Interconnect
AXI3/AXI4 Interconnect
Bridge
Source Peripheral
Destination Peripheral

3.2 Clocks and Resets

DW_axi_dmac design supports different modes of clocking based on the value of DMAX_SLVIF_CLOCK_MODE, DMAX_MSTIF1_CLOCK_MODE, and DMAX_MSTIF2_CLOCK_MODE coreConsultant parameters. There can be a maximum of four clocks in the design; DW_axi_dmac internally takes care of the clock domain crossing requirements.

There can be one to four reset inputs to DW_axi_dmac:

- Slave interface – one reset
- Master 1 and 2 interfaces – one reset each
- One core reset input based on the values of the following coreConsultant parameters:
    - DMAX_SLVIF_CLOCK_MODE,
    - DMAX_MSTIF1_CLOCK_MODE, AND
    - DMAX_MSTIF2_CLOCK_MODE

It is assumed that all reset inputs are asserted and de-asserted at the same time with de-assertion synchronous to the corresponding clocks. De-assertion may not be exactly at the same time as there may be delays associated with respect to the synchronization to each clock domain. All resets can be derived from a single reset input.

Note DW_axi_dmac does not support asserting only of a few resets (not all) or asserting resets at different times (with a delay between each); doing so results in unpredictable behavior.

DW_axi_dmac supports a soft reset, which is set using the DMAC_RST field in the DMAC_ResetReg register. The soft reset procedure is as follows.

1. Software writes 1 to the DMAC_ResetReg.DMAC_RST bit to reset the DW_axi_dmac

   Software polls the DMAC_ResetReg.DMAC_RST bit until it is seen as 0, which confirms the reset.
2. DW_axi_dmac resets all the modules in different clock domains except for the slave bus interface module.

   The Slave bus interface module is not reset because software is polling the DMAC_ResetReg.DMAC_RST field.
3. DW_axi_dmac clears the DMAC_ResetReg.DMAC_RST bit to 0.

Note Software is not allowed to write 0 to the DMAC_ResetReg.DMAC_RST field. A reset does not guarantee the completion of the data transfer for ongoing or posted requests. If a reset is asserted in between transfers, it might result in AXI protocol violations.

3.3 Slave Bus Interface

Slave Bus Interface Module implements the logic to access the internal registers of DW_axi_dmac by an external AHB/AXI4-Lite/APB3 Master. This module supports only the little endian scheme for data transfer. The Slave Bus Interface module supports a 32- or 64-bit data bus width (the APB3 interface supports only 32 bits).

The Slave Bus Interface can be configured to operate on a different clock than the DW_axi_dmac core clock (dmac_core_clock), which can be configured by setting of the DMAX_SLVIF_CLOCK_MODE parameter in coreConsultant. If DMAX_SLVIF_CLOCK_MODE is set to 1, the Slave Bus Interface module operates on either hclk, aclk, or pclk depending on the protocol used for this interface. The DW_axi_dmac design takes care of the clock domain crossing in this situation.

For more details about configuration and programming of this module, refer to Chapter 4, “Parameters”, Chapter 5, “Signals”, and Chapter 6, “Registers”.

Note The current version of DW_axi_dmac supports only the AHB protocol for the slave bus interface; AXI4-Lite and APB3 support is deferred to future versions.

3.4 Master Interface

The master interface implements the transfer of data on the AXI bus. The AXI protocol can be either AXI3 or AXI4, which is configurable in coreConsultant. DW_axi_dmac supports a maximum of two master interfaces, which operate independently to transfer data between peripherals and memories. Both master interfaces must use the same AXI protocol. Address and Data bus widths of master interfaces are configurable in coreConsultant, but both master interfaces use the same configuration.

The master interface implements all five channels specified in the AXI protocol:

- Write address
- Write data
- Write response
- Read address
- Read data

The master interface implements different FIFOs for temporary storage of data transferred between external memories or peripherals through different channels in DW_axi_dmac core.

The Master interface can be configured to operate on a different clock than the DW_axi_dmac core clock (dmac_core_clock), which can be configured by setting DMAX_MSTIFN_CLOCK_MODE parameter in coreConsultant. If DMAX_MSTIFN_CLOCK_MODE is set to 1, the Master Interface module operates on aclk_m*N*. The DW_axi_dmac design takes care of the clock domain crossing in this case.

The master interface implements logic to convert between big endian and little endian format. The endian scheme used for the big endian format is the Byte Invariant (BE-8) method, which is the big endian format supported by the AXI protocol. DW_axi_dmac enables you to use the little endian scheme for LLI fetch and LLI write-back, irrespective of the endian format used for data transfer on that particular master interface.

For more information about configuring or programming the Master interface, refer to Chapter 4, “Parameters”, Chapter 5, “Signals”, and Chapter 6, “Registers”.

DW_axi_dmac supports a maximum of eight channels, which can be assigned to either of the master interfaces, based on a well-defined, programmable arbitration scheme for accessing the master interface. Apart from channel source and destination peripherals, LLI fetch and LLI write-back operations also need access to the master interface. DW_axi_dmac implements a programmable arbitration scheme which is explained in the following section.

3.5 Arbitration Scheme

DMA transfers can be split into the following transfer hierarchy levels:

- Transaction level (only applicable for non-memory peripherals)
- Block level
- Complete DMA transfer level
- AXI transfer level

DW_axi_dmac implements a dynamic priority and fair-among-equals arbitration scheme, with options for channel locking at different DMA transfer hierarchy levels.

The number of priority levels available is the same as the number of enabled channels and the priority value is programmable for each channel in the channel configuration register, CHx_CFG. Multiple channels can be given the same priority level, however, some priority levels remain unused. A priority of 7 is the highest priority, and 0 is the lowest.

Dynamic priority and fair-among-equals arbitration is a two-tier arbitration scheme which works as follows:

- The first-tier arbitration uses the priority inputs of the requests and decides which one of the requesting clients is issued the grant signal. The request with the highest priority is granted access to the AXI channel on the master interface.
- If two or more requests to the arbiter have the same programmed priority value, the second-tier arbitration, based on the fair-among-equals scheme, is used. In this scheme, the grant is issued fairly among actively requesting clients with the same priority level.

Locks and Grants

The lock input of the arbiter enables a request, despite other requests, to be given an exclusive grant for the duration of the corresponding lock input. After a client receives the grant, it can lock out other clients from the arbitration process by setting the corresponding lock input.

DW_axi_dmac design allows a channel to be locked to the arbiter at the transaction (only applicable for non-memory peripherals), block, or complete DMA transfer level. By default, DW_axi_dmac locks the arbiter for one complete AXI transfer. If channel locking is enabled at a higher level—for a transaction, block, or complete DMA transfer—the channel that locked the arbiter gets access to the master interface until the end of locking period, and requests from other channels during this time are ignored.

Priority for Different Accesses

If the source, destination, and linked list peripherals of different channels with the same priority value are connected on the same master interface, the priority for different accesses are as follows:

- For AXI Read Channel

  Linked List fetch > Source Data Transfer/Source Status fetch/Destination Status fetch

- For AXI Write Channel

  Destination Data Transfer > Linked List Write-back (CHx_LLP_STATUS, Source and Destination Status)

Same Channel Transfer, Fetch, and Access

Source data transfer, destination data transfer, linked list fetch, and linked list status write back access for a particular channel generally happens sequentially as follows:

Linked list fetch > Source data transfer and destination data transfer > Source status fetch and destination status fetch > linked list write back

There is no need for arbitration between different accesses in this case.

3.6 Channel Locking

DMA transfers can be split into the following transfer hierarchy levels:

- Complete transfer level
- Block level
- Transaction level (only applicable for non-memory peripherals)
- AXI transfer level

DW_axi_dmac allows a channel to be locked to the arbiter during memory-to-memory transfers at block and complete DMA transfer levels. If channel locking is enabled on a block or at the complete DMA transfer level, the channel that locked the arbiter gets access to the master interface until the end of the locking period, and requests from other channels during this time are ignored.

3.6.1 Enabling Channel Locking

Channel locking can be implemented in the following scenarios.

- If channel locking is enabled at a particular transfer hierarchy and it is a memory-to-memory transfer, the locking of the corresponding channel to the AXI channels of the master interface is established.
- Transfer is a memory-to-memory transfer.

Note Hardware does not check for the validity of the channel locking setting, therefore, the software must enable the channel locking only for memory-to-memory transfers at block or DMA transfer levels. Illegal programming of channel locking might result in unpredictable behavior.

3.6.2 Clearing Channel Locking

Channel locking is cleared when DW_axi_dmac suspends, disables, or aborts the channel upon request from software, or DW_axi_dmac disables the channel upon receiving an error response on the master interface.

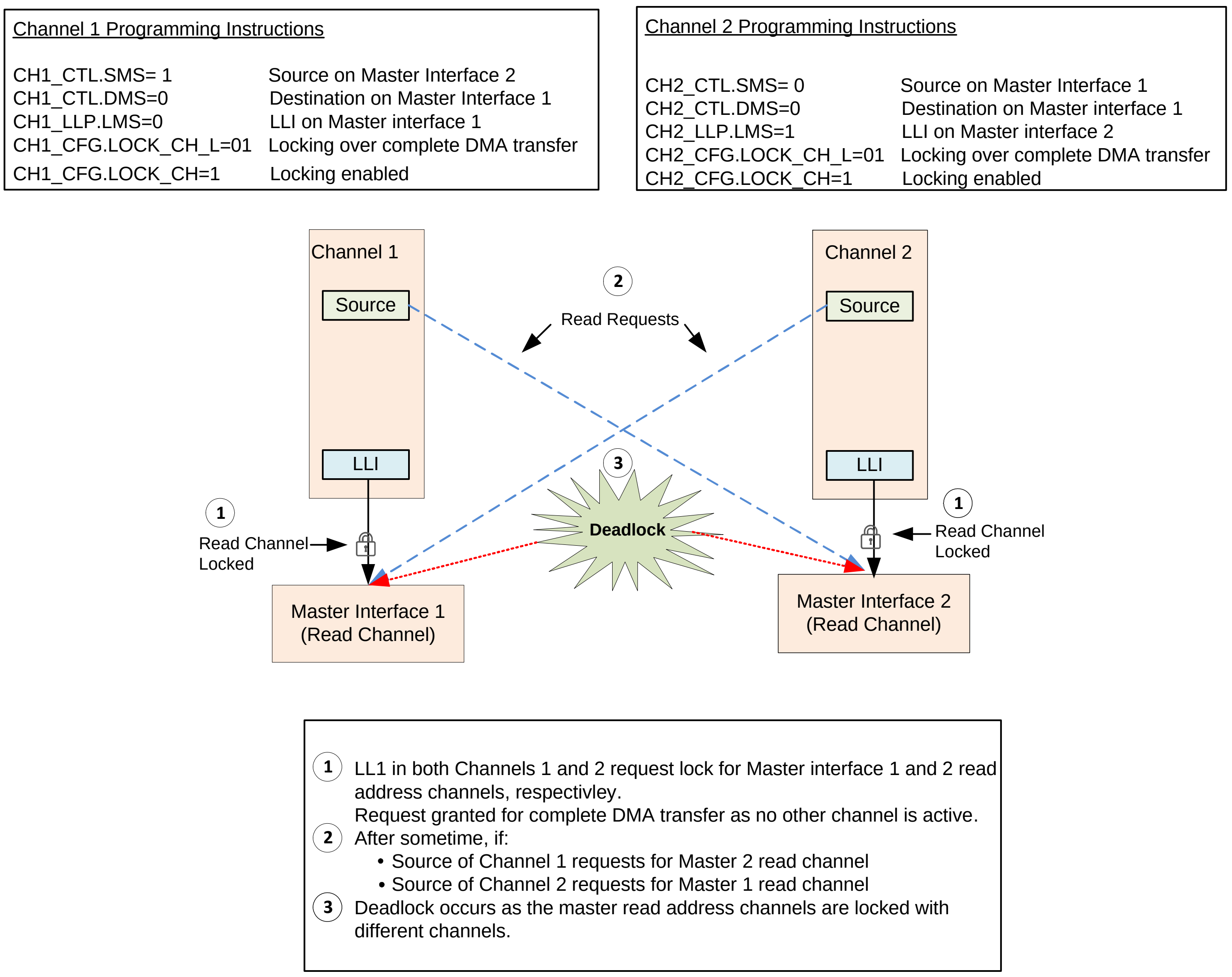
The ChLock_Cleared_IntStat bit in the CHx_IntStatusReg register is set to 1 if DW_axi_dmac clears the channel locking due to an error response received on the master interface, or if software suspends, disables, or aborts the channel. Additionally, the ChLock_Cleared_IntStat is cleared at the end of each block or DMA transfer if channel locking is enabled on DMA block level or transfer level, respectively.

3.6.3 Possible Deadlock Conditions

A deadlock condition can occur when multiple channels are enabled concurrently on a master interface and one of these channels has obtained a read lock on the master interface and the remaining channels are locked out from proceeding with a DMA transfer. While programming, user must ensure that sure deadlock do not occur. A deadlock can occur only for configurations where the number of masters are equal to 2 (DMAC_NUM_MASTER_IF = 2) and the number of channels are more than 1 (DMAC_NUM_CHANNELS > 1).

Figure 3-3 illustrates a scenario in which the read channel is locked out for subsequent read requests.

Figure 3-3 Scenario 1: Deadlock Condition



Channel 1 Programming Instructions

CH1_CTL.SMS= 1	Source on Master Interface 2
CH1_CTL.DMS=0	Destination on Master Interface 1
CH1_LLP.LMS=0	LLI on Master interface 1
CH1_CFG.LOCK_CH_L=01	Locking over complete DMA transfer
CH1_CFG.LOCK_CH=1	Locking enabled

Channel 2 Programming Instructions

CH2_CTL.SMS= 0	Source on Master Interface 1
CH2_CTL.DMS=0	Destination on Master interface 1
CH2_LLP.LMS=1	LLI on Master interface 2
CH2_CFG.LOCK_CH_L=01	Locking over complete DMA transfer
CH2_CFG.LOCK_CH=1	Locking enabled

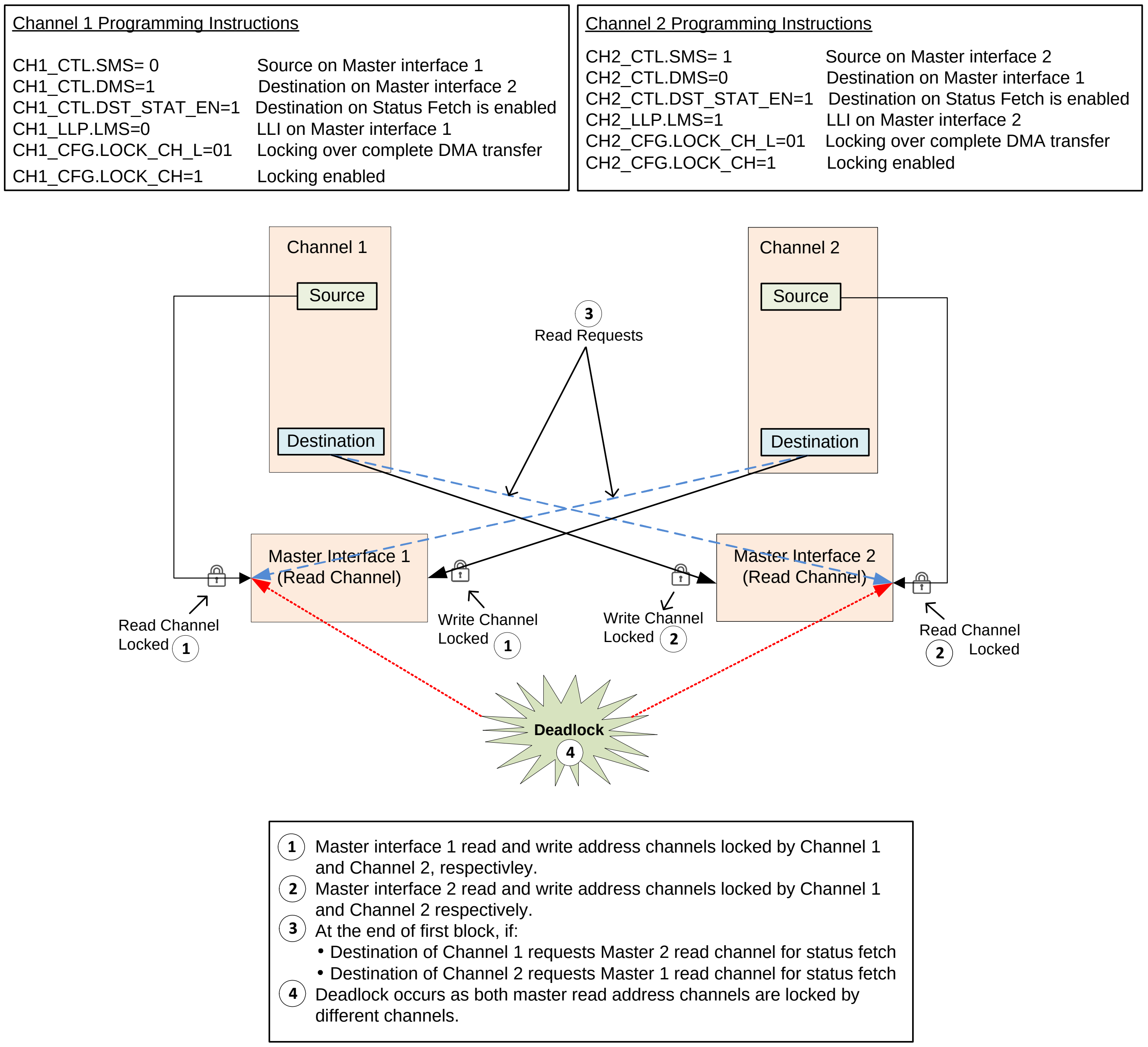
1 LL1 in both Channels 1 and 2 request lock for Master interface 1 and 2 read address channels, respectivley.
Request granted for complete DMA transfer as no other channel is active.

2 After sometime, if:
- Source of Channel 1 requests for Master 2 read channel
- Source of Channel 2 requests for Master 1 read channel

3 Deadlock occurs as the master read address channels are locked with different channels.

Figure 3-4 illustrates another scenario in which the read channel is locked out for subsequent read requests.

Figure 3-4 Scenario 2: Deadlock Condition



Channel 1 Programming Instructions

CH1_CTL.SMS= 0	Source on Master interface 1
CH1_CTL.DMS=1	Destination on Master interface 2
CH1_CTL.DST_STAT_EN=1	Destination on Status Fetch is enabled
CH1_LLP.LMS=0	LLI on Master interface 1
CH1_CFG.LOCK_CH_L=01	Locking over complete DMA transfer
CH1_CFG.LOCK_CH=1	Locking enabled

Channel 2 Programming Instructions

CH2_CTL.SMS= 1	Source on Master interface 2
CH2_CTL.DMS=0	Destination on Master interface 1
CH2_CTL.DST_STAT_EN=1	Destination on Status Fetch is enabled
CH2_LLP.LMS=1	LLI on Master interface 2
CH2_CFG.LOCK_CH_L=01	Locking over complete DMA transfer
CH2_CFG.LOCK_CH=1	Locking enabled

1 Master interface 1 read and write address channels locked by Channel 1 and Channel 2, respectivley.
2 Master interface 2 read and write address channels locked by Channel 1 and Channel 2 respectively.
3 At the end of first block, if:
- Destination of Channel 1 requests Master 2 read channel for status fetch
- Destination of Channel 2 requests Master 1 read channel for status fetch

4 Deadlock occurs as both master read address channels are locked by different channels.

3.7 Endian Scheme

DW_axi_dmac supports little endian and big endian format for data access on the AXI master interface. The endian scheme used for the big endian format is the Byte Invariant (BE-8) method, which is the format supported by the AXI protocol. If the AXI master interface is configured for the big endian format, big endian BE-8-to-little endian conversion is done for AXI read data and little endian-to-big endian BE-8 conversion is done for AXI write data.

The following figures show the conversion process between big endian BE-8 and little endian data appearing on the AXI master bus for different data bus width and transfer size combinations.

Figure 3-5 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 32-Bit Data Bus (Byte Access)

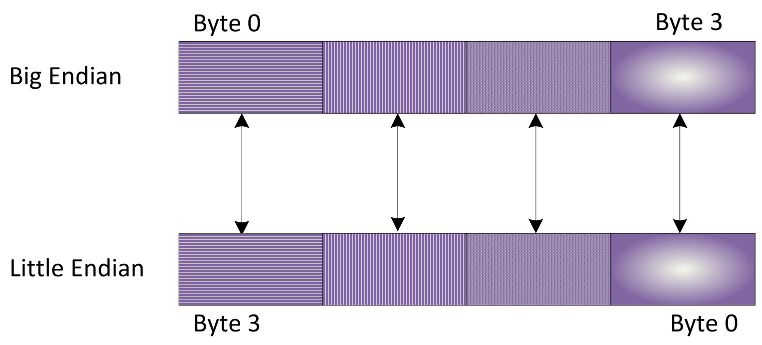


Figure 3-6 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 32-Bit Data Bus (Half Word Access)

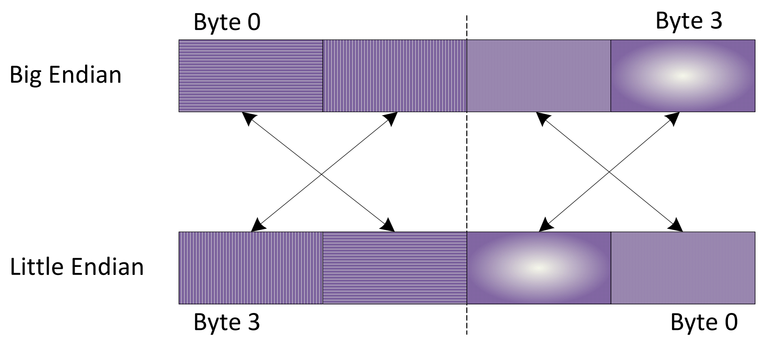


Figure 3-7 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 32-Bit Data Bus (Word Access)

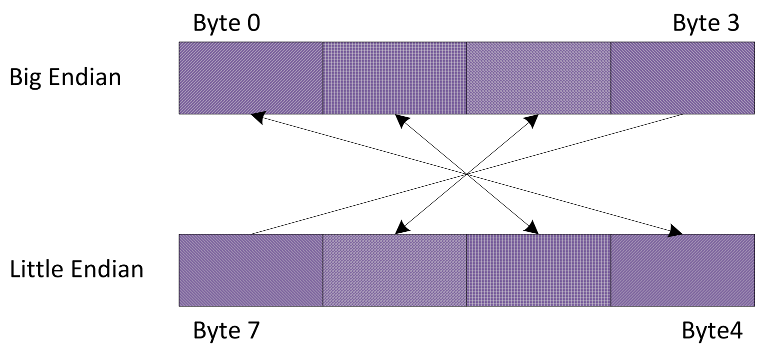


Figure 3-8 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 64-Bit Data Bus (Byte Access)

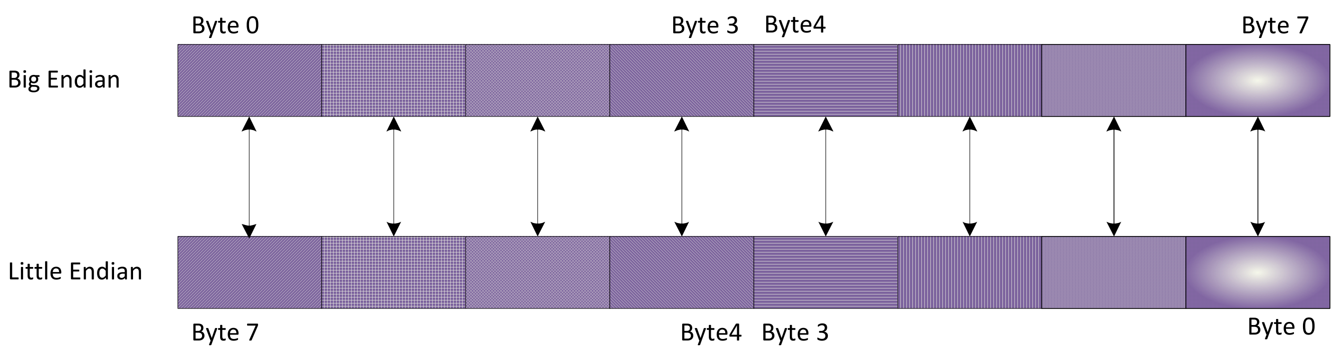


Figure 3-9 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 64-Bit Data Bus (Half Word Access)

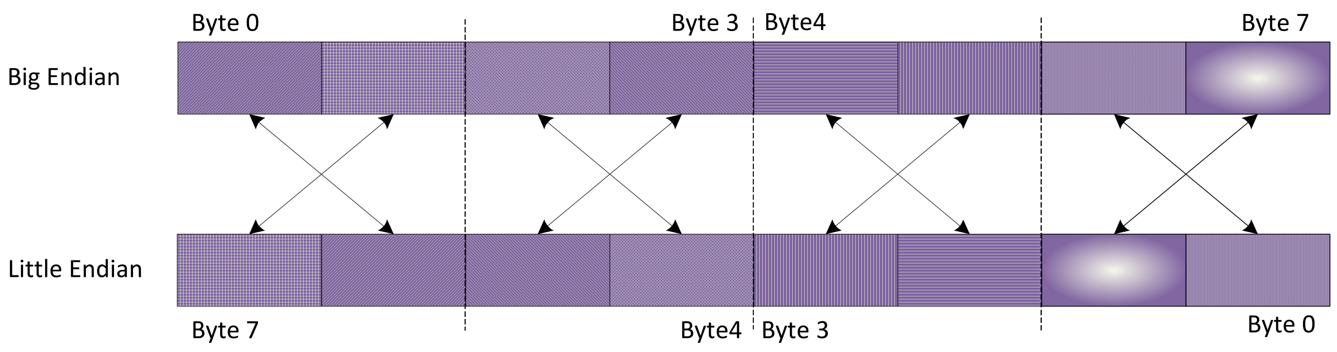


Figure 3-10 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 64-Bit Data Bus (Word Access)

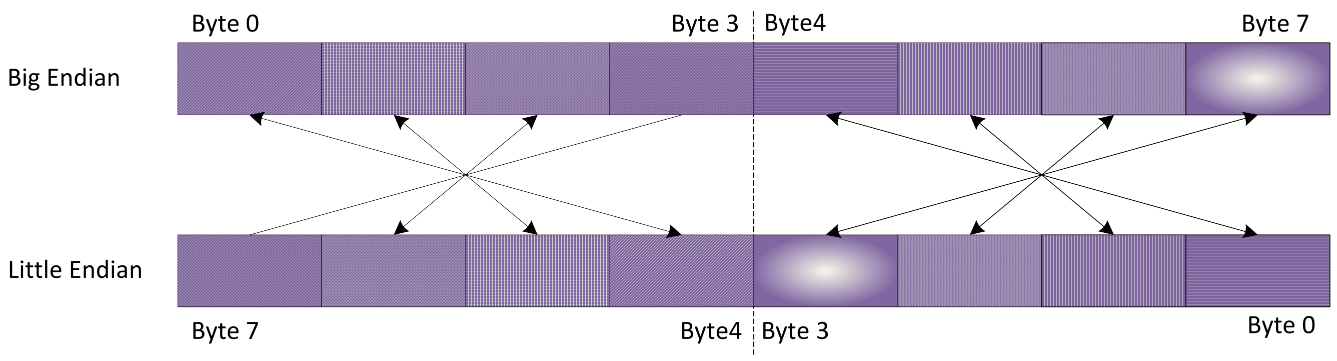


Figure 3-11 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 64-Bit Data Bus (Double Word Access)

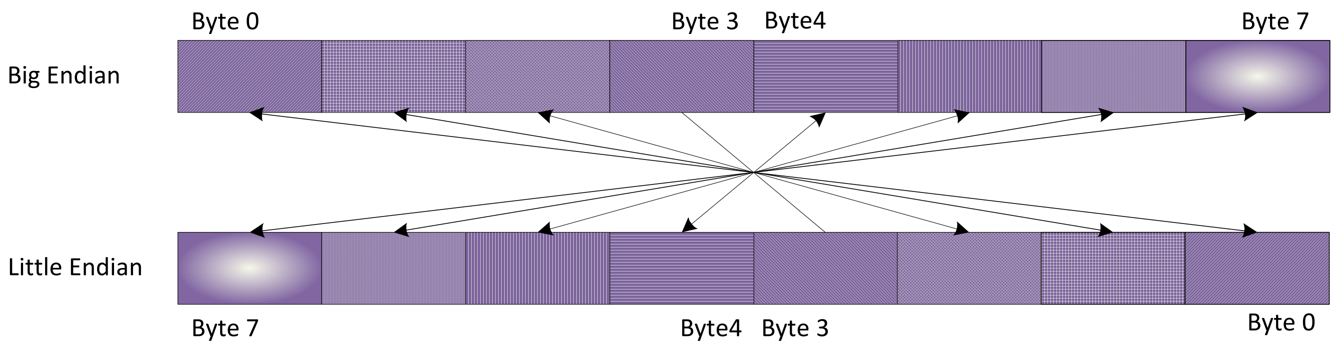


Figure 3-12 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 128-Bit Data Bus (Byte Access)

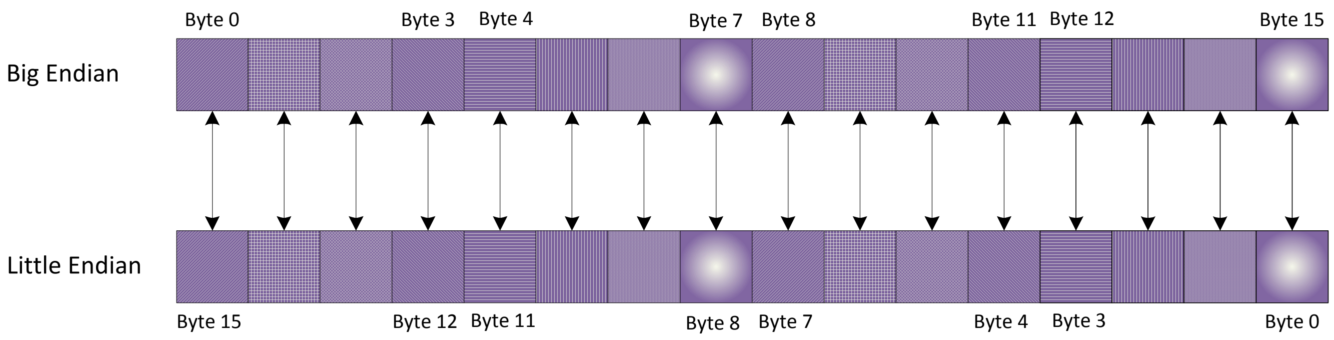


Figure 3-13 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 128-Bit Data Bus (Half Word Access)

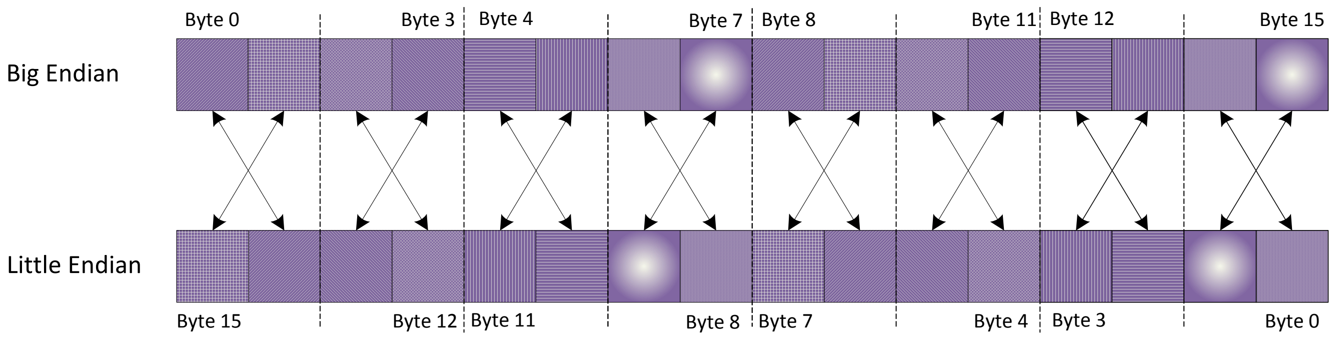


Figure 3-14 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 128-Bit Data Bus (Word Access)

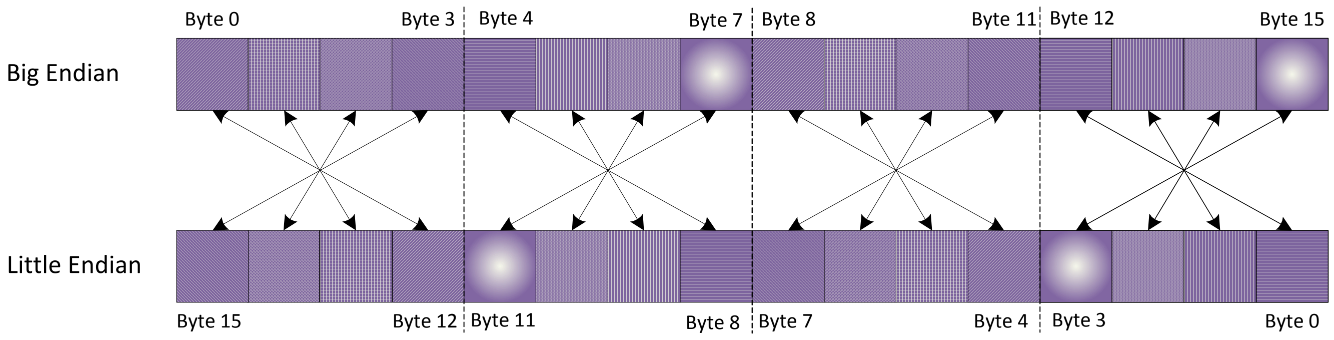


Figure 3-15 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 128-Bit Data Bus (Double Word Access)

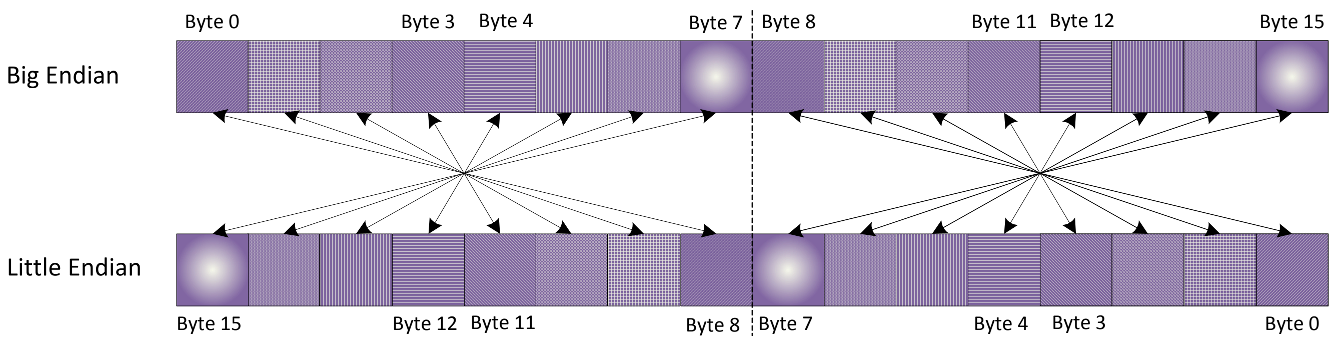
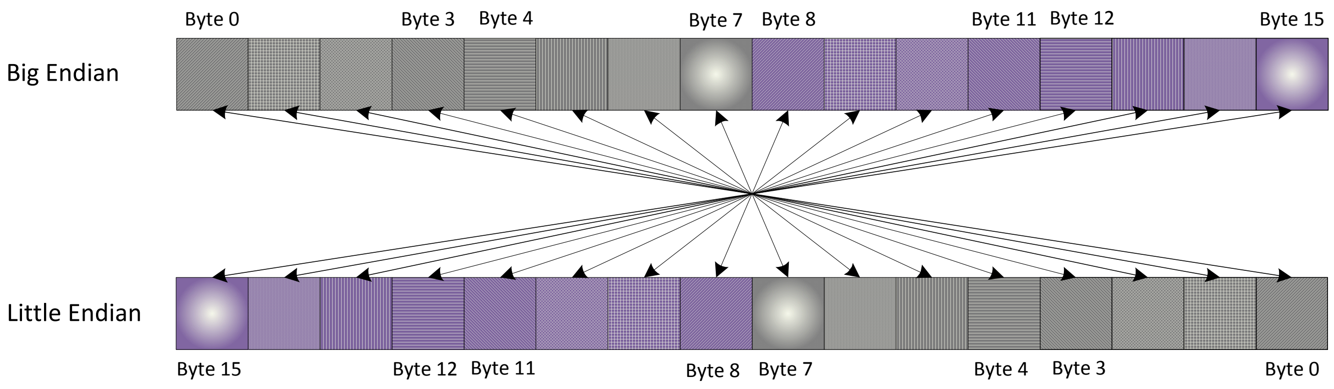


Figure 3-16 BE-LE Conversion Using BE-8 (Byte Invariant) Method for 128-Bit Data Bus (128-Bit Access)



Note A similar scheme is used for other transfer sizes as well. In general, the data bus is grouped in terms of transfer size and bytes within each group are swapped such that the lower byte becomes the higher byte, and the higher byte becomes the lower byte.

The endian scheme used for the source and destination data transfer, LLI fetch, and status and control write-back access on the master interfaces is decided by the value of the respective coreConsultant parameters and the I/O signals. Table 3-1 on page 51 shows all possible combinations of the endian scheme on the master interface.

Table 3-1 Endian Formats Supported on Master Interface

DMAX_STATIC_ENDIAN_SELECT_MSTIF	DMAX_ENDIAN_FORMAT_MSTIF	dmac_endian_format_mstif1/2	Endian Scheme Used for SRC/DST Data Access on M1/M2 Master Interface	DMAX_LLI_ENDIAN_SELECTION_PIN_EN	dmac_le_select_lli_mstif1/2	Endian Scheme Used for LLI Fetch/ Status and Control Write Back Access on M1/M2 Master Interface
1	0	X	Little Endian	X	X	Little Endian
1	1	X	Big Endian BE-8	0	X	Big Endian BE-8
				1	0	Big Endian BE-8
					1	Little Endian
0	X	0	Little Endian	X	X	Little Endian
0	X	1	Big Endian BE-8	0	X	Big Endian BE-8
				1	0	Big Endian BE-8
					1	Little Endian

3.8 Interrupt Interface

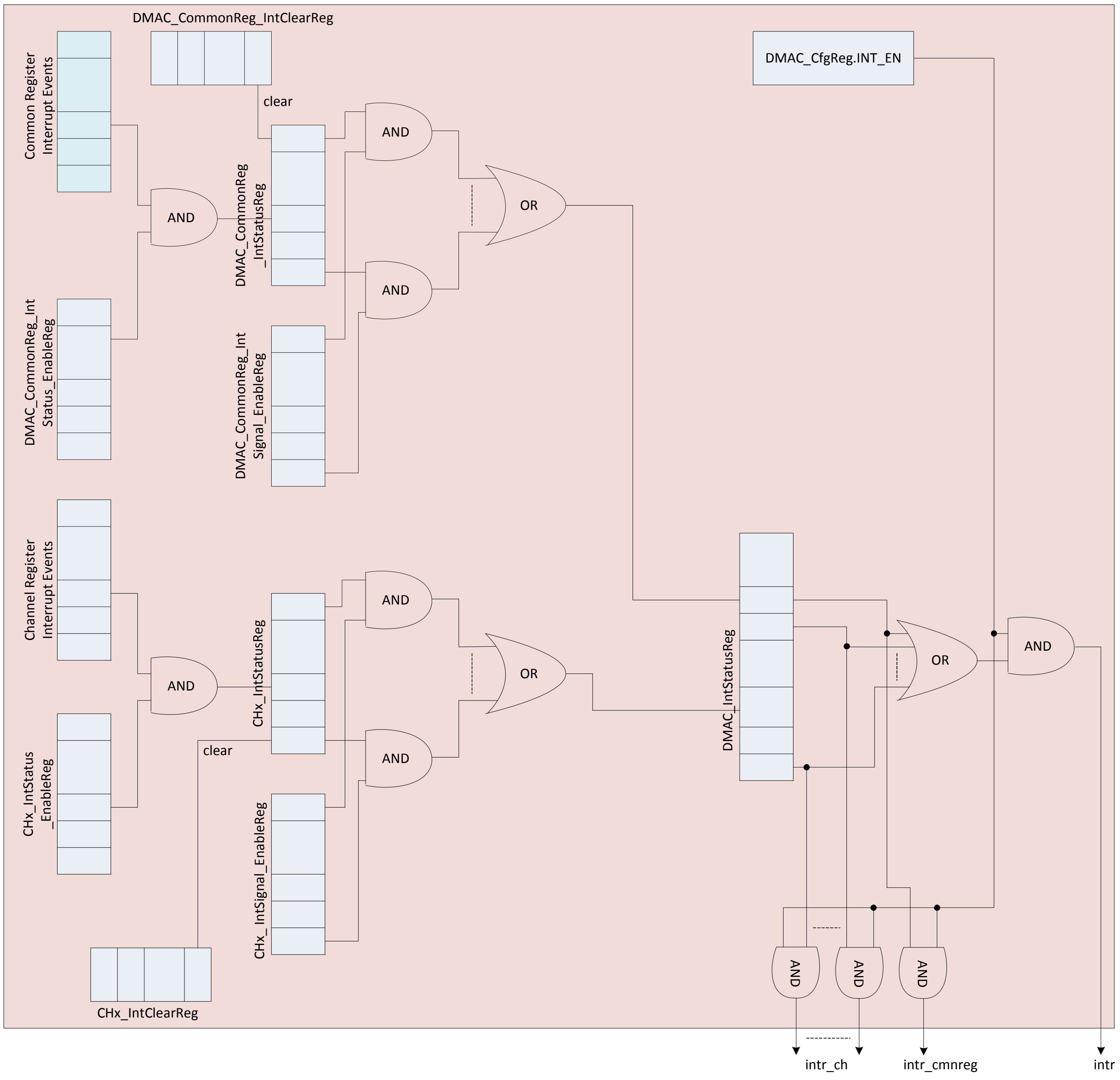
DW_axi_dmac supports combined and individual interrupt outputs configurable in coreConsultant. The polarity of the interrupt (Active High/Active Low) can be configured in coreConsultant. Interrupt outputs, by default, are synchronous to the core clock (dmac_core_clock), but they can be synchronized to the slave interface clock using the DMAX_INTR_SYNC2SLVCLK parameter in coreConsultant. If this parameter is set to 1, interrupt outputs are synchronous to hclk/aclk/pclk based on the protocol used for the slave interface. DW_axi_dmac manages the clock domain synchronization requirements in this configuration.

DW_axi_dmac has different registers to support the interrupt interface. Software can read these registers to understand the source of the interrupt and take appropriate actions.

Note DW_axi_dmac supports synchronizing the interrupt outputs to the Slave Interface clock domain. The synchronization mechanism is not captured in Figure 3-17 on page 53.

Figure 3-17 on page 53 shows the interrupt generation mechanism in DW_axi_dmac.

Figure 3-17 DW_axi_dmac Interrupt Generation


DMAC_CommonReg_IntClearReg
DMAC_CfgReg.INT_EN
Common Register Interrupt Events
clear
AND
AND
DMAC_CommonReg _IntStatusReg
OR
AND
DMAC_CommonReg_Int Status_EnableReg
DMAC_CommonReg_Int Signal_EnableReg
Channel Register Interrupt Events
AND
CHx_IntStatusReg
OR
AND
AND
DMAC_IntStatusReg
OR
AND
CHx_IntStatus _EnableReg
clear
CHx_IntSignal_EnableReg
CHx_IntClearReg
AND
AND
AND
intr_ch
intr_cmnreg
intr

3.9 Single Transaction Region

In certain cases, 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. 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_axi_dmac samples the dma_single status flag and completes the block transfer using single transactions. The peripheral asserts a single status flag to indicate to the DW_axi_dmac that there is enough data or space to complete a single transaction from or to the source or destination peripheral.

For hardware handshaking, the single status flag is a signal on the hardware handshaking interface. For software handshaking, the single status flag is one bit in the software handshaking interface register.

The single transaction region is the time interval where the DW_axi_dmac uses single transactions to complete the block transfer; burst transactions are used only outside this region.

The following terms are used for explaining single transaction region.

- Source single transaction size in bytes:

  src_single_size_bytes = CHx_CTL.SRC_TR_WIDTH/8
- Source burst transaction size in bytes:

  src_burst_size_bytes = CHx_CTL.SRC_MSIZE * src_single_size_bytes
- Destination single transaction size in bytes:

  dst_single_size_bytes = CHx_CTL.DST_TR_WIDTH/8
- Destination burst transaction size in bytes:

  dst_burst_size_bytes = CHx_CTL.DST_MSIZE * dst_single_size_bytes
- Block size in bytes:
  - If DW_axi_dmac is flow controller – With the DW_axi_dmac as the flow controller, the processor programs the DW_axi_dmac with the number of data items (block size) of source transfer width (CHx_CTL.SRC_TR_WIDTH) to be transferred by the DW_axi_dmac in a block transfer; this is programmed into the CHx_BLOCK_TS.BLOCK_TS field. Therefore, the total number of bytes to be transferred in a block is:

    blk_size_bytes_dma = CHx_BLOCK_TS.BLOCK_TS * src_single_size_bytes
  - If source peripheral is 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)
  - If 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)

The single transaction region applies only to a peripheral that is not the flow controller. The precise definition of when this region is entered depends on what acts as the flow controller.

- If the DW_axi_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_dma/src_burst_size_bytes is an integer, then the source peripheral never enters this region, and the source block transfer 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_dma/dst_burst_size_bytes is an integer, then the destination peripheral never enters this region, and the destination block transfer uses only burst transactions.
- If either the source or the destination peripheral is the flow controller:
  - Single transaction region is not applicable for the flow controller peripheral.
  - If the source peripheral is the flow controller, the destination peripheral enters the single transaction region when the flow control peripheral – that is, the source – signals the last transaction in the block and when the amount of data left to be transferred in the destination block is less than the value specified by dst_burst_size_bytes.
  - If the destination peripheral is the flow controller, the source peripheral enters the single transaction region when the flow control peripheral – that is, the destination – signals the last transaction in the block and when the amount of data left to be transferred in the source block is less than the value specified by src_burst_size_bytes.

Note If a peripheral is not the flow controller, DW_axi_dmac ignores dma_single input outside the single transaction region.

3.10 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 depends on whether the peripheral or the DW_axi_dmac is the flow controller. The peripheral uses the handshaking interface to indicate to the DW_axi_dmac that it is ready to transfer or accept data over the AXI bus.

DW_axi_dmac supports a maximum of 16 hardware handshaking interfaces, which are configurable in coreConsultant. The type of handshaking interface used (Software or Hardware) is independently programmable for each channel source and destination in the channel configuration register, CHx_CFG. Software handshaking is accomplished through memory-mapped registers, while hardware handshaking is accomplished using a dedicated handshaking interface.

3.10.1 Hardware Handshaking

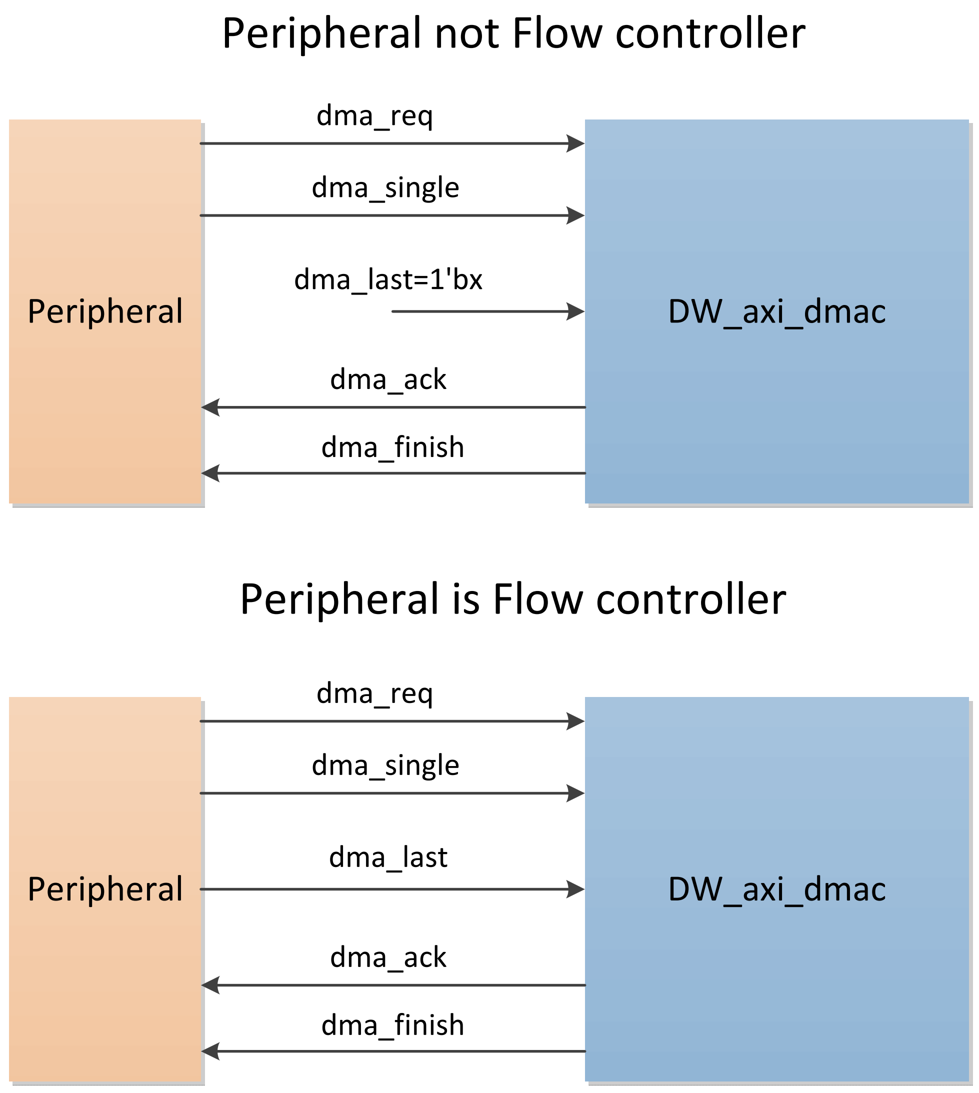
The following set of I/O signals are used for hardware handshaking.

- **dma_req** – Burst transaction request input from peripheral. The functionality of this signal depends on whether the peripheral is the flow controller or not.
    - **If peripheral is not flow controller** – The DW_axi_dmac always interprets the dma_req signal as a burst transaction request, regardless of the level of 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. If an active level on dma_req is detected in the single transaction region, then the block transfer is completed using an early terminated burst transaction.
    - **If peripheral is flow controller** – An active level on dma_req initiates a transaction request. The type of transaction – whether single or burst – is qualified by the dma_single input. 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.
- **dma_single** - Single transaction request input from peripheral. The functionality of this signal depends on whether the peripheral is the flow controller or not.
    - **If peripheral is not flow controller** – The dma_single signal is a status signal that is asserted by a source or destination peripheral when it can transmit or accept at least one source or destination data item; otherwise it is cleared. This signal is sampled by the DW_axi_dmac only in the single transaction region of the block transfer. Outside this region, dma_single is ignored and all transactions are burst transactions, which means DW_axi_dmac waits for dma_req to be asserted. Once dma_single is asserted, it must remain asserted until dma_ack is asserted. When the peripheral that is driving dma_single determines that dma_ack is asserted, it must de-assert dma_single.
    - **If peripheral is flow controller** – The dma_single signal is asserted by a source or destination peripheral to request a single transaction. If dma_single is asserted in the same clock cycle as dma_req is asserted, a single transaction is requested by the peripheral. If dma_single is de-asserted, the peripheral is requesting a burst transaction. Once dma_single is asserted, it must remain asserted until dma_ack is asserted. When the peripheral that is driving dma_single determines that dma_ack is asserted, it must de-assert dma_single.

- **dma_last** - The last transaction in a block request from a peripheral. The functionality of this signal depends on whether the peripheral is the flow controller or not.
  - **If peripheral is not flow controller** – The dma_last signal is not relevant and it is ignored.
  - **If peripheral is flow controller** – The peripheral 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 and the block transfer is complete after this transaction is complete. If dma_single is high in the same cycle as dma_last (and dma_req) is asserted, the last transaction is a single transaction. If dma_single is low in the same cycle as dma_last (and dma_req) is asserted, the last transaction is a burst transaction. Once dma_last is asserted, it must remain asserted until dma_ack is asserted. When the peripheral that is driving dma_last determines that dma_ack is asserted, it must de-assert dma_last.
- **dma_ack** - DW_axi_dmac acknowledges signal output to the peripheral. The functionality of this signal depends on whether the peripheral is the flow controller or not.
  - **If peripheral is not flow controller** – The dma_ack signal is asserted after the last AXI data transfer in the current transaction (single or burst) to the peripheral has completed. For a single transaction, dma_ack remains asserted until the peripheral de-asserts dma_single (that is, dma_ack is de-asserted one dmac_core_clock cycle after the de-assertion of dma_single). For a burst transaction, dma_ack remains asserted until the peripheral de-asserts dma_req (that is, dma_ack is de-asserted one dmac_core_clock cycle after the de-assertion of dma_req).
  - **If peripheral is flow controller** – The dma_ack signal is asserted after the last AXI data 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 (that is, dma_ack is de-asserted one dmac_core_clock cycle after the de-assertion of dma_req).
- **dma_finish** - This is the DW_axi_dmac block transfer complete indication signal output to the peripheral. The dma_finish signal is asserted to signal block completion. This uses the same timing as dma_ack and forms a handshaking loop with dma_req and/or dma_single.

Figure 3-18 on page 58 shows the hardware handshaking interface between a destination or source peripheral and DW_axi_dmac.

Figure 3-18 Hardware Handshaking Interface



Note

- Once a peripheral asserts dma_req, dma_single, or dma_last, de-asserting the signal before the DW_axi_dmac asserts dma_ack is not allowed. Doing so might result in unpredictable behavior.
- Irrespective of who is the flow controller and whether the peripheral is in Single Transaction Region, dma_req, dma_single, and dma_last signals must be de-asserted once DW_axi_dmac asserts dma_ack. It is recommended that you follow this requirement, otherwise, unpredictable behavior may occur.

Table 3-2 shows all possible combinations of Flow Controller and Hardware Handshaking Interface signal values.

Table 3-2 Flow Controller and Hardware Handshaking Interface

Peripheral Flow Controller?	Peripheral in Single Transaction Region?	dma_req	dma_single	dma_last	DW_axi_dmac Action
No	No	0	0	X	Not a valid transaction request
		0	1	X	Not a valid transaction request. If peripheral is not Flow Controller, “dma_single” is not sampled by DW_axi_dmac outside Single Transaction Region.
		1	X	X	Burst transaction request. If peripheral is not Flow Controller, “dma_single” is not sampled by DW_axi_dmac outside Single Transaction Region. If peripheral is not the flow controller, “dma_last” doesn't have any relevance and it is ignored.
	Yes	0	0	X	Not a valid transaction request.
		0	1	X	Single Transaction request. DW_axi_dmac initiates AXI burst of burst length 1. If peripheral is not the flow controller, “dma_last” doesn't have any relevance and it is ignored.
		1	0	X	Burst Transaction request. DW_axi_dmac initiates AXI burst of burst length needed to complete the transaction (AXI burst length <= transaction size). This is called Early Burst Termination. If peripheral is not the flow controller, “dma_last” doesn't have any relevance and it is ignored.
		1	1	X	Burst Transaction request. DW_axi_dmac initiates AXI burst of burst length needed to complete the transaction (AXI burst length <= transaction size). This is called Early Burst Termination. If peripheral is not the flow controller, “dma_last” doesn't have any relevance and it is ignored.

Table 3-2 Flow Controller and Hardware Handshaking Interface (Continued)

Peripheral Flow Controller?	Peripheral in Single Transaction Region?	dma_req	dma_single	dma_last	DW_axi_dmac Action
Yes	Single Transaction Region is not applicable to Flow Control Peripheral	0	0	X	Not a valid transaction request.
		0	1	X	Not a valid transaction request. If peripheral is flow controller, "dma_req" MUST be asserted to in it ate a transaction. DW_axi_dmac waits till "dma_req" is asserted. The type of transaction - single or burst - is qualified by "dma_single" input.
		1	0	0	Burst transaction request (not the last transaction).
		1	0	1	Last Burst transaction request.
		1	1	0	Single transaction request (not the last transaction).
		1	1	1	Last single transaction request.

3.10.1.1 Hardware Handshaking Transaction Examples

This section provides various examples of hardware handshaking scenarios, as follows:

- "Burst Transaction – DMA Flow Controller"
- "Back-to-Back Burst Transaction – DMA Flow Controller" on page 61
- "Single Transaction – DMA Flow Controller" on page 62
- "Burst Followed by Back-to-Back Single Transaction – DMA Flow Controller" on page 63
- "Early Terminated Burst Transaction – DMA Flow Controller" on page 63
- "Burst Transaction Ignored During Active Single Transaction – DMA Flow Controller" on page 64
- "Burst Transaction Followed by Single Transaction – Peripheral Flow Controller" on page 65
- "Back-to-Back Single Transaction – Peripheral Flow Controller" on page 66

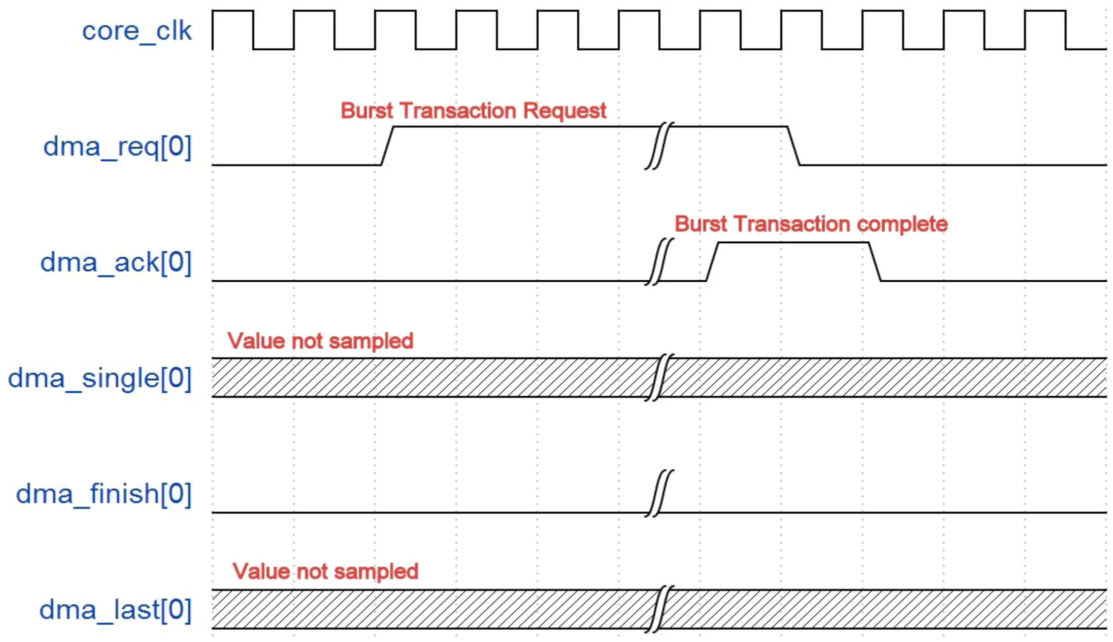
Burst Transaction – DMA Flow Controller

Figure 3-19 on page 61 shows the timing diagram of a burst transaction. Because the peripheral is outside the Single Transaction Region, DW_axi_dmac does not sample dma_single[0].

The handshaking loop is as follows:

- dma_req asserted by peripheral
- dma_ack asserted by DW_axi_dmac
- dma_req de-asserted by peripheral
- dma_ack de-asserted by DW_axi_dmac

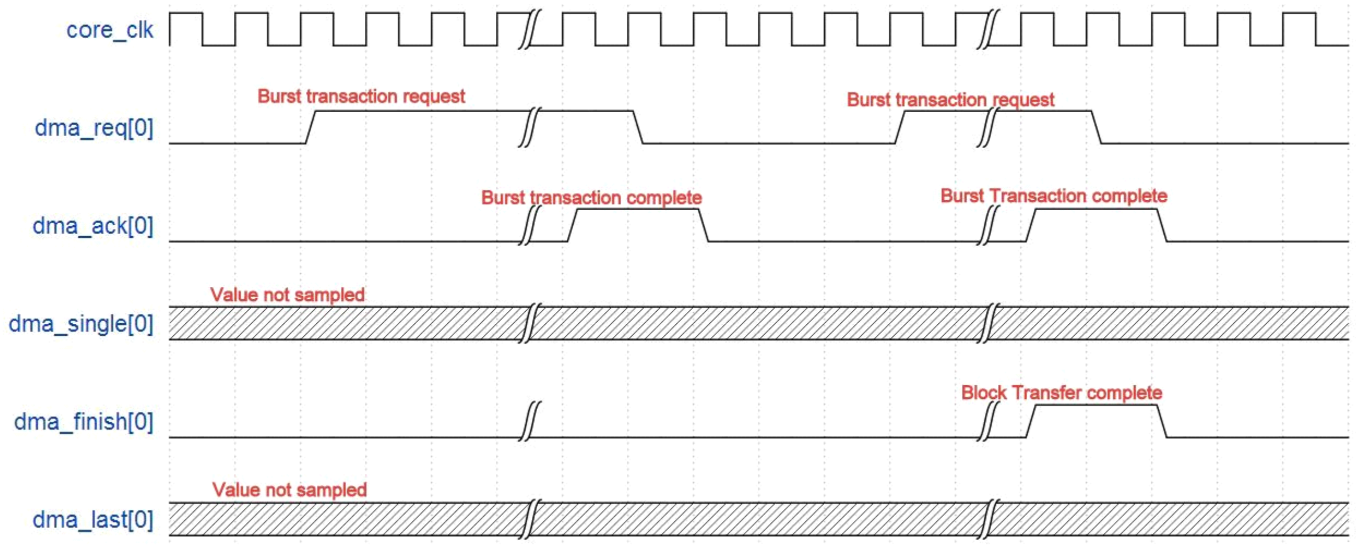
Figure 3-19 Burst Transaction



Back-to-Back Burst Transaction – DMA Flow Controller

Figure 3-20 shows the timing diagram of a back-to-back burst transaction. Because the peripheral is outside the Single Transaction Region, DW_axi_dmac does not sample dma_single[0]. The second burst terminates the block, and dma_finish[0] is asserted to indicate the block completion.

Figure 3-20 Back-to-Back Burst Transaction



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

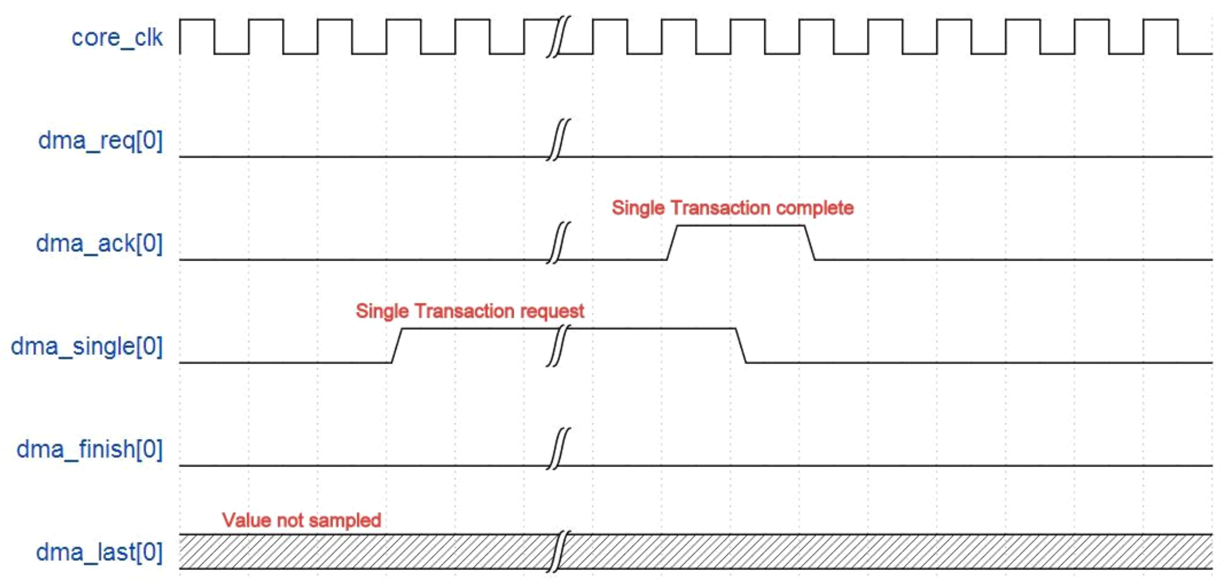
- Once asserted, the dma_req burst request signal must remain asserted until the corresponding dma_ack signal is received, even if the condition that generates dma_req in the peripheral is False.
- The dma_req signal must be de-asserted when dma_ack is asserted, even if the condition that generates dma_req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

Single Transaction – DMA Flow Controller

Figure 3-21 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_axi_dmac
- dma_single de-asserted by peripheral
- dma_ack de-asserted by DW_axi_dmac

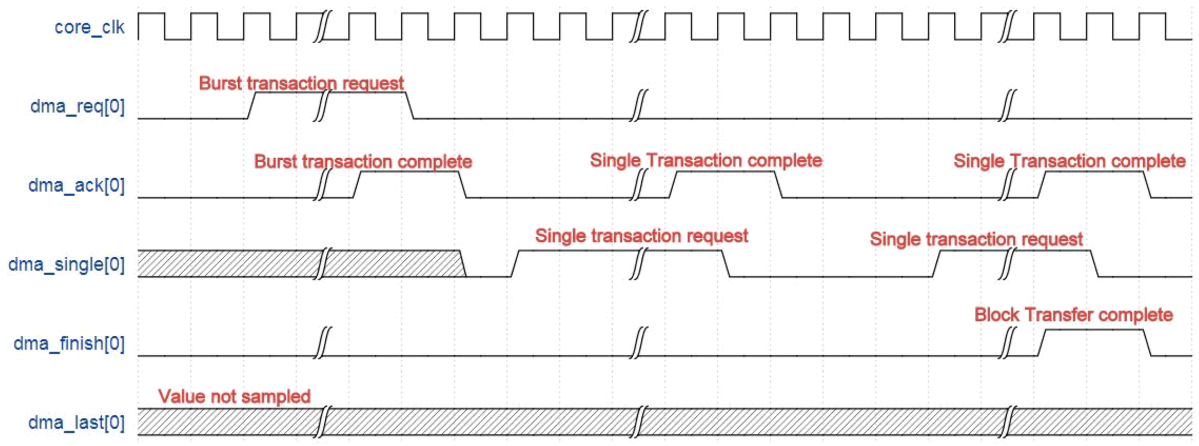
Figure 3-21 Single Transaction



Burst Followed by Back-to-Back Single Transaction – DMA Flow Controller

After the first burst transaction, the peripheral enters the Single Transaction Region and DW_axi_dmac starts sampling the dma_single. The second single transaction terminates the block transfer; dma_finish[0] is asserted to indicate block completion. Figure 3-22 illustrates this scenario.

Figure 3-22 Burst Followed by Back-to-Back Single Transaction

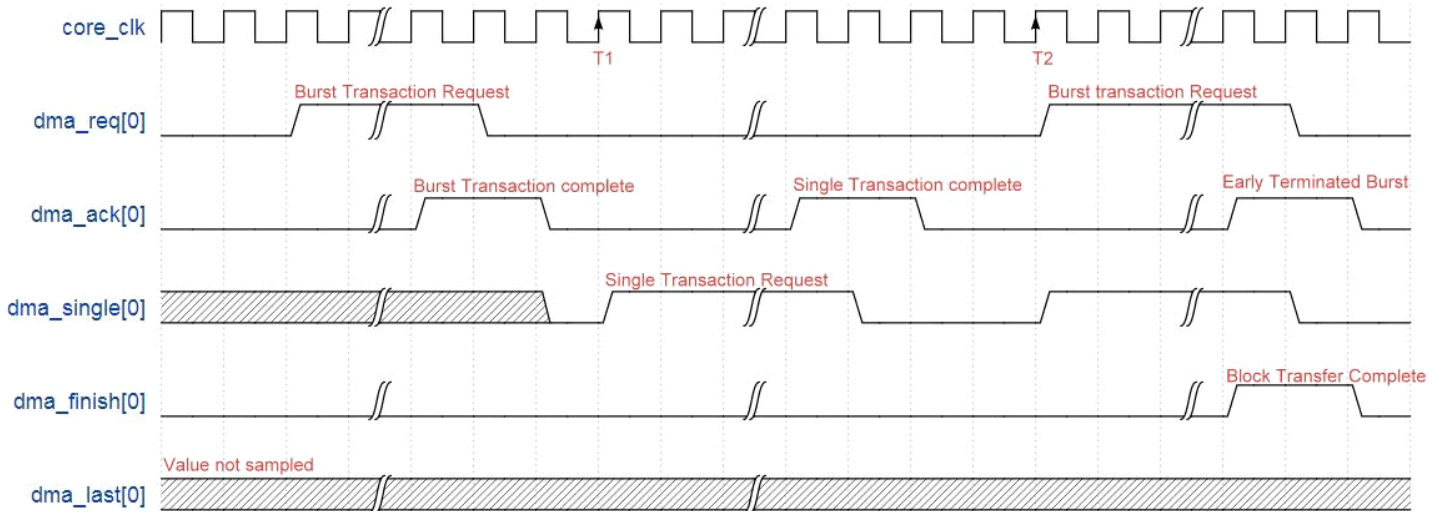

core_clk
dma_req[0]
dma_ack[0]
dma_single[0]
dma_finish[0]
dma_last[0]
Burst transaction request
Burst transaction complete
Single Transaction complete
Single Transaction complete
Single transaction request
Single transaction request
Block Transfer complete
Value not sampled

Early Terminated Burst Transaction – DMA Flow Controller

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 in same cycle as dma_req - then the burst transaction takes precedence over the single transaction, and the block would be completed using an early terminated burst transaction.

In Figure 3-23 on page 64, after the first burst DW_axi_dmac enters the single transaction region and samples dma_single[0] at T1 after that. When one single transaction is done, at time T2 dma_req[0] is asserted which results in an early terminated burst, hence completing the block indicated by dma_finish[0].

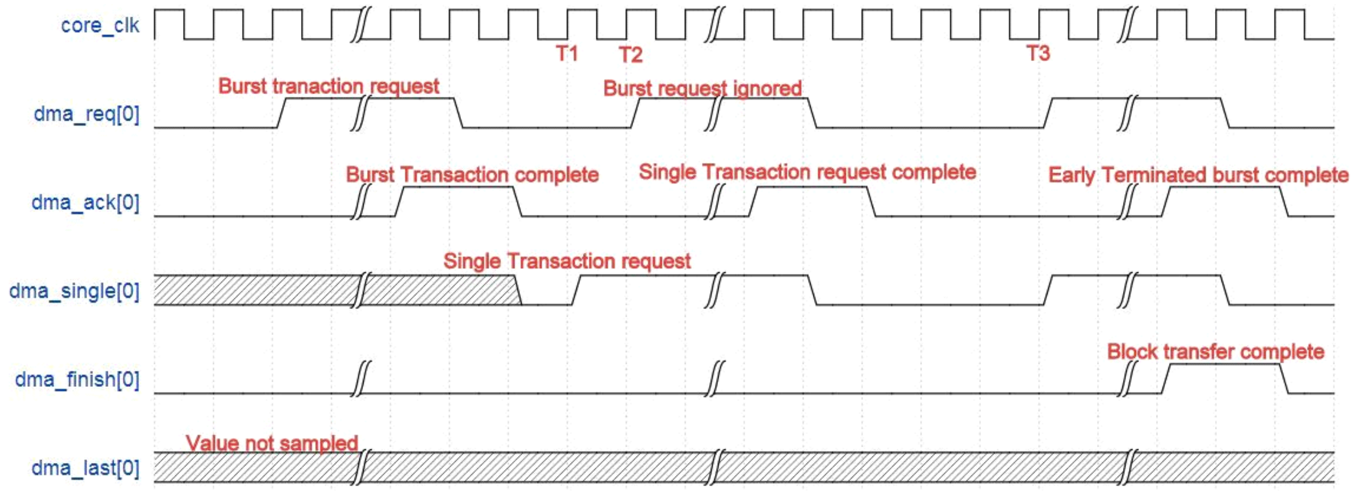
Figure 3-23 Early Terminated Burst Transfer



Burst Transaction Ignored During Active Single Transaction – DMA Flow Controller

As illustrated in Figure 3-24, after the first burst transaction completes, the peripheral is in the Single Transaction Region and DW_axi_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_axi_dmac at time T3. This burst transaction completes the block transfer using an Early-Terminated Burst Transaction.

Figure 3-24 Burst Transaction Ignored During Active Single Transaction



Burst Transaction Followed by Single Transaction – Peripheral Flow Controller

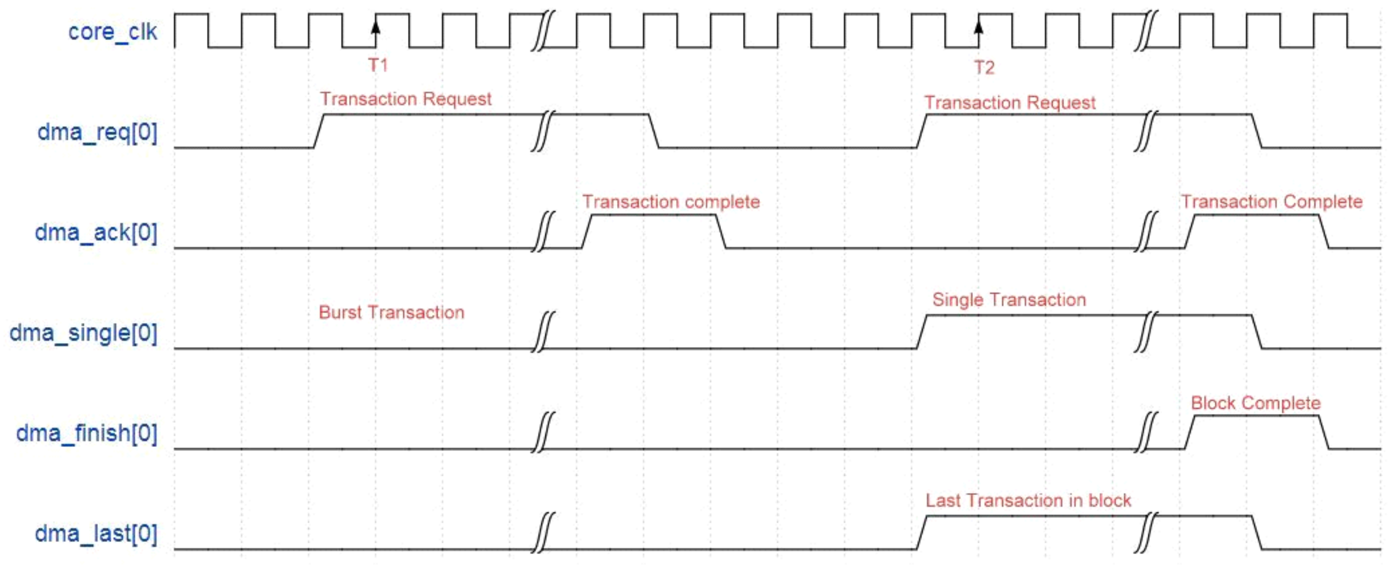
Figure 3-25 shows a burst transaction followed by a single transaction, where the single transaction is the last in the block. On clock edge T1, DW_axi_dmac samples that dma_req[0] is asserted, and dma_single[0] and dma_last[0] are de-asserted. This is a request for a burst transaction, which is not the last transaction in the block.

On clock edge T2, DW_axi_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 handshaking loop is as follows:

- dma_req along with dma_single and dma_last asserted by peripheral
- dma_ack along with dma_finish asserted by DW_axi_dmac
- dma_req, dma_single and dma_last de-asserted by peripheral
- dma_ack and dma_finish de-asserted by DW_axi_dmac

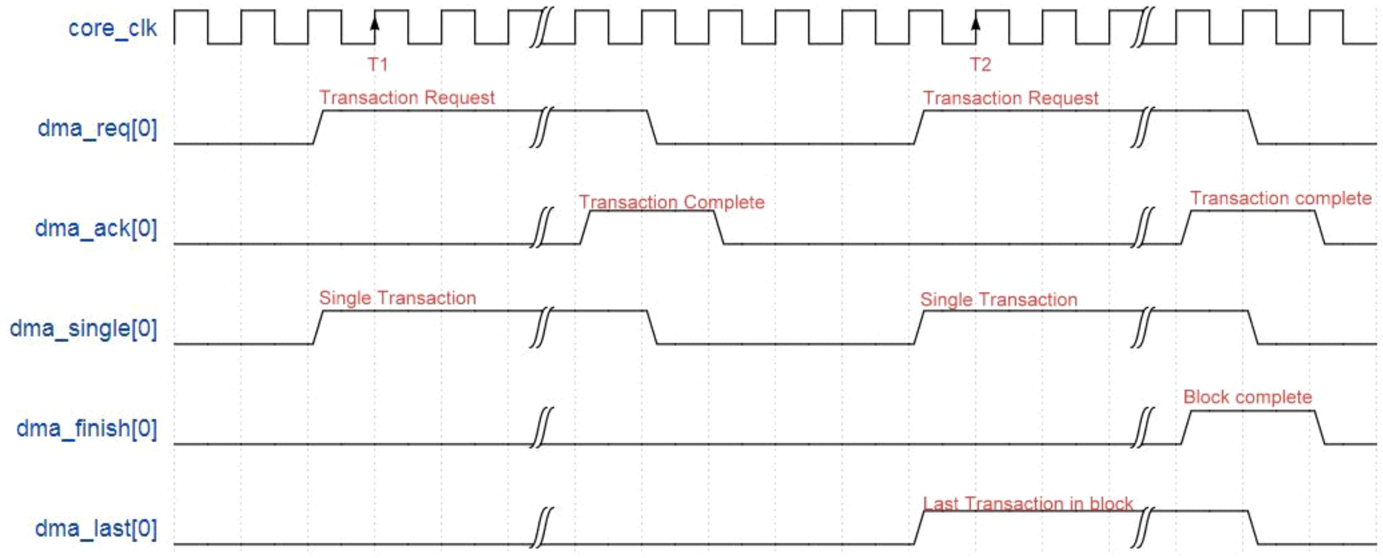
Figure 3-25 Burst Followed by Single Transaction (last in block)



Back-to-Back Single Transaction – Peripheral Flow Controller

Figure 3-26 shows a back-to-back single transaction, where the last single transaction is the last transaction in the block.

Figure 3-26 Back-to-Back Single Transation – Peripheral Flow Controller



3.10.1.2 Peripheral Interrupt Request Interface

This 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.
- The dma_last input is tied low.
- The dma_ack and dma_finish outputs are ignored (unconnected).

This interface can be used when the slave peripheral does not have hardware handshaking signals. To the DW_axi_dmac, this is the same situation as the case of the Hardware Handshaking Interface when the peripheral is not the flow contoller.

The peripheral can never be the flow controller because it cannot connect to the dma_last signal. The interrupt line from the peripheral is tied to the dma_req line and the timing of the interrupt line from the peripheral must be the same as the dma_req line as the case of the Hardware Handshaking Interface when the peripheral is not the flow controller.

Because the peripheral does not sample the dma_ack line, the handshaking loop is as follows:

1. Peripheral generates an interrupt that asserts “dma_req”.
2. DW_axi_dmac completes the burst transaction and generates an end-of-burst transaction interrupt, SRC_TransComp/DST_TransComp. Global Interrupt must be enabled in DMAC_CfgReg register and the corresponding channel's transaction completion indication interrupt must be enabled in CHx_IntStatus_EnableReg and CHx_IntSignal_EnableReg registers.
3. The interrupt service routine clears the interrupt in the peripheral so that the “dma_req” is de-asserted.

3.10.2 Software Handshaking

When the slave peripheral requires the DW_axi_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 handshaking registers to initiate and control a DMA transaction. A group of software registers is used to implement the software handshaking interface. The HS_SEL_SRC/HS_SEL_DST bit in the channel configuration register CHx_CFG must be set to enable software handshaking.

The following registers are used for software handshaking:

- CHx_SWHSSrcReg – Software Handshake Register for Source of Channel x
- CHx_SWHSDstReg – Software Handshake Register for Destination of Channel x

DW_axi_dmac clears the active request bits in SWHSSrcReg to 0 when the corresponding source transaction is completed. Similarly, active request bits in SWHSDstReg are cleared to 0 when the destination transaction is completed.

Note The functionality of the software handshaking register bits are same as that of the corresponding hardware handshaking signals except that there are no register fields corresponding to dma_ack and dma_finish signals.

Suspension of Transfer While Using Software Handshaking

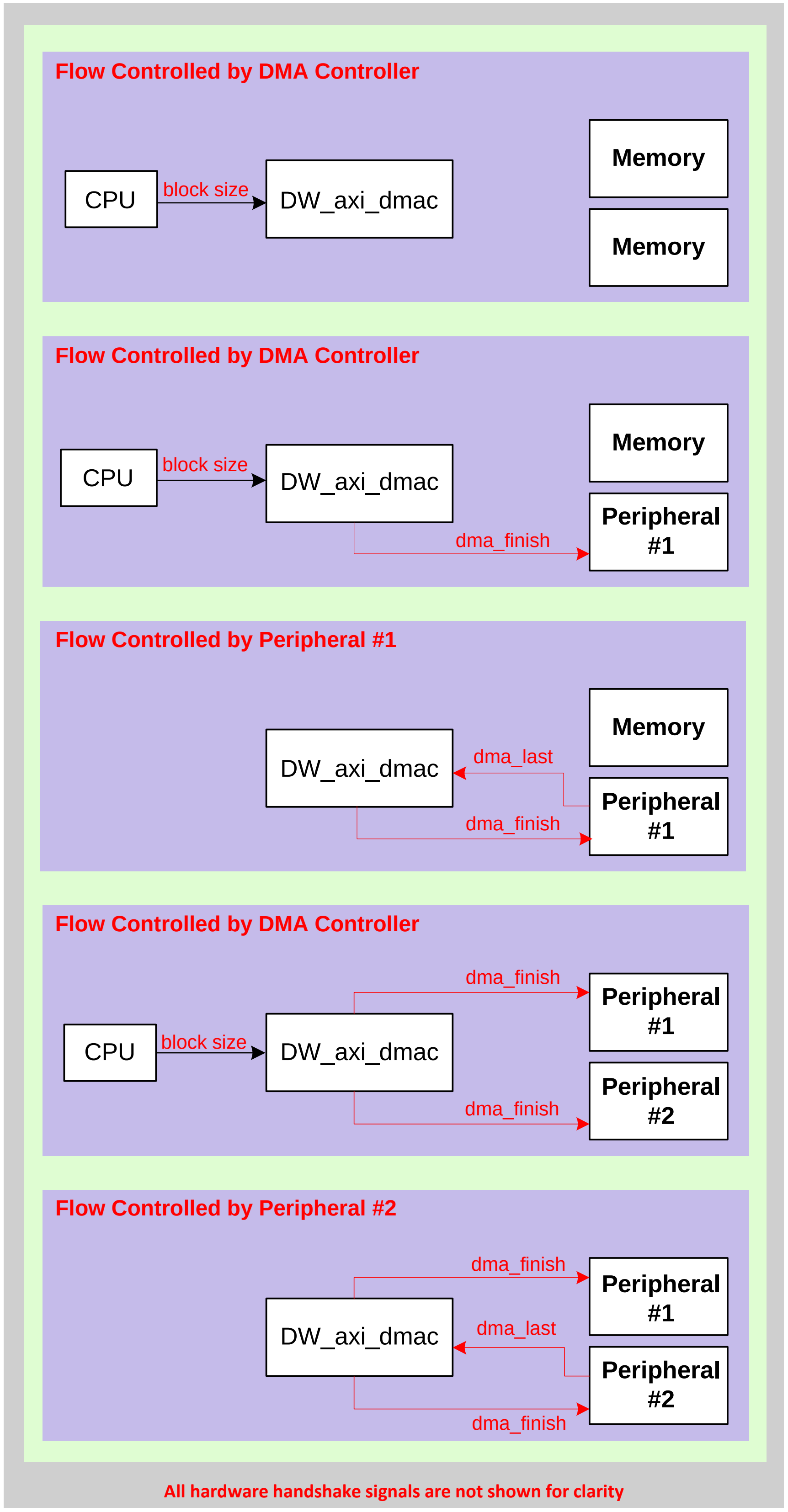
If software handshaking is used for a source and/or destination peripheral, the DMA transfer automatically stalls after completion of the requested source and/or destination transaction. The DW_axi_dmac does not proceed with the transfer until the hardware detects that software has set CHx_SWHSSrcReg.SWHS_SglReq_Src and/or CHx_SWHSDstReg.SWHS_SglReq_Dst bit to '1'.

3.11 Flow Control Configurations

The transfer type and flow control configurations are decided by the value of the TT_FC field in the CHx_CFG register, which is programmable for a particular DMA transfer.

Figure 3-27 on page 68 indicates 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 hardware signals.

Figure 3-27 DW_axi_dmac Flow Control Configuration



3.12 Early Terminated Burst Transaction

When a source or destination peripheral is in the single transaction region, a burst transaction can still be requested. However, in this case, src_burst_size_bytes or dst_burst_size_bytes is greater than the number of bytes left to complete the source or destination block transfer at the time 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. Only the amount of data required to complete the block transfer is transferred.

An early terminated burst transaction occurs only when the peripheral is not the flow controller.

- If the source peripheral is the flow controller and it does not have enough data for a burst transfer towards the end of a transaction, it asserts dma_single until it signals the last transaction by asserting dma_last along with dma_single. (The destination peripheral might enter the single transaction region at this point.)
- If the destination peripheral is in the single transaction region and the destination peripheral still requests a burst transaction by asserting dma_req, then DW_axi_dmac starts a burst transaction and early terminates it at block completion without transferring the programmed amount (CHx_CTL.DST_MSIZE) of data.
- If the destination peripheral is the flow controller and it does not have enough data for a burst transfer towards the end of transaction, it asserts dma_single until it signals the last transaction by asserting dma_last along with dma_single. (The source peripheral might enter the single transaction region at this point.)
- If the source peripheral is in the single transaction region and the source peripheral still requests a burst transaction by asserting dma_req, then DW_axi_dmac starts a burst transaction and early terminates it at block completion without transferring the programmed amount (CHx_CTL.SRC_MSIZE) of data.

Note When the destination peripheral is the flow controller, there can be data loss if the following conditions are met:

- CHx_CTL.SRC_TR_WIDTH > CHx_CTL.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
```

3.13 Transfer Control

Transfer control logic facilitates the data transfer from a channel's source peripheral to the destination peripheral. It interacts with multiple other modules, such as the channel source and destination state machine, linked list control logic, channel registers, channel FIFO control logic and so on.

Data from the source peripheral is temporarily stored in the channel FIFO before being sent to the destination peripheral. DW_axi_dmac implements the logic needed to pack and unpack data to fit the FIFO configuration, if source and destination peripherals use different transfer size (arsize, awsize) for data transfer.

3.13.1 Single Block Transfer

If a DMA transfer consists of a single block, the software sets the multi-block type bits of both source and destination peripherals in the CHx_CFG register to 2'b00. In this case, the DW_axi_dmac disables the channel once the block transfer corresponding to the block length programmed in CHx_BLOCK_TS register is completed.

The CHx_IntStatusReg register is updated with the status corresponding to the completed block transfer. The CHx_IntStatusReg register is updated based on the settings of the CHx_IntMaskReg register and interrupts are generated based on the settings of the CHx_IntMaskReg, DMAC_IntMaskReg, and DMAC_CfgReg registers.

3.13.2 Multiblock Transfer

If DW_axi_dmac is programmed for multiblock transfers, the CHx_SAR and CHx_DAR registers are reprogrammed using either of the following methods on successive blocks of a multiblock transfer:

- Contiguous Address
- Auto Reloading
- Shadow Register
- Linked List

The CHx_CTL and CHx_BLOCK_TS registers are reprogrammed using either of the following methods on successive blocks of a multiblock transfer:

- Auto Reloading
- Shadow Register
- Linked List

The CHx_IntStatusReg register is updated with the status corresponding to the completed block transfer. The CHx_IntStatusReg register is updated based on the settings of the CHx_IntMaskReg and interrupts are generated based on the settings of the CHx_IntMaskReg, DMAC_IntMaskReg, and DMAC_CfgReg registers.

Note

- If the coreConsultant parameter DMAX_CHx_MULTI_BLK_EN is set to 0, the logic for multiblock transfer is disabled and only single block transfers are allowed.
- If a multiblock transfer is enabled, there must be at least two blocks in the complete DMA transfer for Contiguous Address and Auto-Reloading-based multiblock transfers. Otherwise, unpredictable behavior will occur.

The register update method for multiblock transfers depends on the value of the multiblock type bits in the CHx_CFG register. All possible combinations of the multiblock register update methods are captured in the following table.

Table 3-3 Register Update Methods for Multiblock Transfer

DMAX_CHx_MULTI_BLK_EN	Multiblock Type Bits in CHx_CFG Register		Register Update Method				Multiblock Transfer Type (MLTBLK_TFR_TYPE)	Remarks
	SRC_MLTBLK_TYPE	DST_MLTBLK_TYPE	CHx_SAR	CHx_DAR	CHx_CTL	CHx_BLOCK_TS		
0	X X	X X	No update	No update	No update	No update	-	Single block
1	0 0	0 0	No update	No update	No update	No update	-	Single block or last block of multiblock
1	0 0	0 1	Contiguous	Reloaded from initial value	Reloaded from initial value	Reloaded from initial value	1	-
1	0 0	1 0	Contiguous	Loaded from shadow register	Loaded from shadow register	Loaded from shadow register	2	
1	0 1	0 0	Reloaded from initial value	Contiguous	Reloaded from initial value	Reloaded from initial value	3	-
1	0 1	0 1	Reloaded from initial value	Reloaded from initial value	Reloaded from initial value	Reloaded from initial value	4	-
1	0 1	1 0	Reloaded from initial value	Loaded from shadow register	Loaded from shadow register	Loaded from shadow register	5	-
1	1 0	0 0	Loaded from shadow register	Contiguous	Loaded from shadow register	Loaded from shadow register	6	-
1	1 0	0 1	Loaded from shadow register	Reloaded from initial value	Loaded from shadow register	Loaded from shadow register	7	-
1	1 0	1 0	Loaded from shadow register	Loaded from shadow register	Loaded from shadow register	Loaded from shadow register	8	-
1	0 0	1 1	Contiguous	Loaded from next LLI	Loaded from next LLI	Loaded from next LLI	9	-
1	0 1	1 1	Reloaded from initial value	Loaded from next LLI	Loaded from next LLI	Loaded from next LLI	10	-
1	1 1	0 0	Loaded from next LLI	Contiguous	Loaded from next LLI	Loaded from next LLI	11	-

Table 3-3 Register Update Methods for Multiblock Transfer (Continued)

DMAX_CHx_MULTI_BLK_EN	Multiblock Type Bits in CHx_CFG Register: SRC_MLTBLK_TYPE	Multiblock Type Bits in CHx_CFG Register: DST_MLTBLK_TYPE	Register Update Method: CHx_SAR	Register Update Method: CHx_DAR	Register Update Method: CHx_CTL	Register Update Method: CHx_BLOCK_TS	Multiblock Transfer Type (MLTBLK_TFR_TYPE)	Remarks
1	1 1	0 1	Loaded from next LLI	Reloaded from initial value	Loaded from next LLI	Loaded from next LLI	12	-
1	1 1	1 1	Loaded from next LLI	Loaded from next LLI	Loaded from next LLI	Loaded from next LLI	13	-
1	1 0	1 1	Loaded from shadow register	Loaded from next LLI	Loaded from next LLI	Loaded from next LLI	-	Invalid programming. SLVIF_MultiBlkType_ERR interrupt is generated
1	1 1	1 0	Loaded from next LLI	Loaded from shadow register	Loaded from next LLI	Loaded from next LLI	-	Invalid programming. SLVIF_MultiBlkType_ERR interrupt is generated

3.13.2.1 Contiguous Address

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 CHx_CTL.SRC_MLTBLK_TYPE and CHx_CTL.DST_MLTBLK_TYPE register fields.

CHx_SAR and CHx_DAR updates cannot be selected to be contiguous at the same time. If required, this functionality can be achieved indirectly, by using linked lists. To do this, set up the LLI.CHx_SAR address of the next block descriptor to be one greater than the end address of the previous block. Similarly, set up the LLI.CHx_DAR address of the next block descriptor to be one greater than the end address of the previous block.

Note If a Contiguous-Address-based multiblock transfer is enabled, there must be at least two blocks in the complete DMA transfer. Otherwise, unpredictable behavior will occur.

3.13.2.2 Auto Reloading

In this case, the channel transfer control registers are reloaded with their initial values at the completion of each block and these values are used for the new block. Some or all of the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, and CHx_CTL channel registers are reloaded from their initial value at the start of a new block transfer, depending on the multi-block transfer type selected for source and destination peripherals.

The DW_axi_dmac does not proceed to the next block transfer until software clears the corresponding channel's block transfer complete interrupt by writing '1' to the corresponding bit in the DMAC_IntClear_Reg register if this interrupt is not masked off.

If the block transfer completion interrupt is disabled, DW_axi_dmac proceeds to the next block without waiting for the software to clear the interrupt. Hence, in non-interrupt mode of operation, software must ensure multiblock transfer type bits are updated in time so that DW_axi_dmac does not perform unintended block transfers. This issue can happen for the last block in the non-interrupt mode of operation if software does not clear the multiblock transfer type bits before the completion of the last block.

Note If Auto-Reloading-based multi-block transfer is enabled, there should be at least 2 blocks in the complete DMA transfer. Otherwise, it will result in unpredictable behavior.

3.13.2.3 Shadow Register

In this case, the channel transfer control registers are loaded from their corresponding shadow registers at the completion of each block and these values are used for the new block. Some or all of the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, and CHx_CTL channel registers are loaded from their corresponding shadow registers at the start of a new block transfer, depending on the multiblock transfer type selected for source and destination peripherals.

A separate memory map is not defined for the shadow register access. Software always writes to the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers irrespective of the type of multiblock transfer used. If a shadow-register-based multiblock transfer is used for a source or destination transfer, DW_axi_dmac internally routes the data to the corresponding shadow registers. DW_axi_dmac copies the shadow register contents to CHx_SAR, CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers before starting a new block transfer.

Read operations to the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers always return the data corresponding to the current block transfer and not the shadow register contents (which corresponds to the next block).

For shadow-register-based multiblock transfer, the ShadowReg_Or_LLI_Valid bit in the CHx_CTL register indicates whether the shadow register contents are valid or not. Zero indicates that the shadow register contents are invalid, while 1 indicates that the shadow register contents are valid. If this bit is read as zero during a shadow register fetch phase, DW_axi_dmac discards the shadow register contents and generates a ShadowReg_Or_LLI_Invalid_ERR interrupt. DW_axi_dmac waits until the software writes any value to the CHx_BLK_TFR_ResumeReqReg register to indicate valid shadow register availability, before attempting another shadow register fetch operation to continue the next block transfer.

Note If the coreConsultant parameter DMAX_CHx_SHADOW_REG _EN is set to 0, shadow-register-based multi-block transfer is disabled and only other methods of multi-block transfers are allowed.

For more information about programming the Shadow Register, refer to "Programming Flow for Shadow-Register-Based Multi-Block Transfer" on page 197.

3.13.2.4 Linked List

In this case, the DW_axi_dmac reprograms the channel transfer control 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_axi_dmac block chaining uses a linked list pointer register (CHx_LLP) to store the address in memory of the next linked list item. Each LLI contains the following block descriptors:

- CHx_SAR
- CHx_DAR
- CHx_BLOCK_TS
- CHx_CTL
- CHx_LLP

To set up block chaining, a sequence of linked lists should be programmed in memory. DW_axi_dmac allows dynamic extension of linked lists, which eliminates the need for creating the entire linked list in the system memory in advance. The CHx_CTL.ShadowReg_Or_LLI_Valid and CHx_CTL.LLI_Last fields of the LLI are used to achieve this functionality.

For linked-list-based multi-block transfers, the ShadowReg_Or_LLI_Valid bit of the LLI.CHx_CTL register indicates whether the linked list item fetched from the memory is valid or not. If this bit is set to 0, it indicates the LLI is invalid, while 1 indicates the LLI is valid. If the LLI is invalid, DW_axi_dmac discards the LLI and generates an LLI Error interrupt (if the corresponding channel error interrupt mask bit is set to 0). This error condition causes the DW_axi_dmac to halt the corresponding channel gracefully. DW_axi_dmac waits till software writes any value to the CHx_BLK_TFR_ResumeReqReg register to indicate the availability of a valid LLI, before attempting another LLI read operation.

Note In the case of LLI pre-fetching, the ShadowReg_Or_LLI_Invalid_ERR interrupt is not generated even if the ShadowReg_Or_LLI_Valid bit is set to 0 for the pre-fetched LLI. In this case, DW_axi_dmac re-attempts the LLI fetch operation after completing the current block transfer. DW_axi_dmac generates a ShadowReg_Or_LLI_Invalid_ERR interrupt only if the ShadowReg_Or_LLI_Valid bit is still set to 0.

LLI access always uses a burst size (arsize or awsize) that is the same as the data bus width and cannot be changed or programmed to anything other than this. Burst length (awlen or arlen) is chosen based on the data bus width so that the access does not cross one complete LLI structure of 64 bytes. DW_axi_dmac fetches the entire LLI (40 bytes) in one AXI burst, if the burst length is not limited by other setting.

If the status write back option is enabled, DW_axi_dmac writes back CHx_CTL, CHx_LLP_STATUS, CHx_SSTAT, and CHx_DSTAT information to the location defined for this field, which is from address [CHx_LLP] + 0x20 to [CHx_LLP] + 0x34. The CHx_SSTAT and CHx_DSTAT write back can be independently enabled or disabled in coreConsultant and by programming the corresponding field in the

CHx_CTL register. If CHx_SSTAT and/or CHx_DSTAT write back is not enabled, DW_axi_dmac de-asserts the write data strobes corresponding to these write operations.

Note

- CHx_LLP_STATUS[63] and CHx_LLP_STATUS[62] indicates DMA_TFR_DONE and BLOCK_TFR_DONE status respectively and these are the last fields to be updated during LLI write-back. Software should wait until BLOCK_TFR_DONE bit is set to '1' before reading CHx_SSTAT and CHx_DSTAT information. DMA_TFR_DONE bit will be set to '1' along with BLOCK_TFR_DONE bit after transferring the last block in DMA transfer.
- LLI.CHx_CTL.ShadowReg_Or_LLI_Valid bit will be '0' after the LLI write-back operation.

Figure 3-28 DW_axi_dmac Linked List Item (Descriptor)

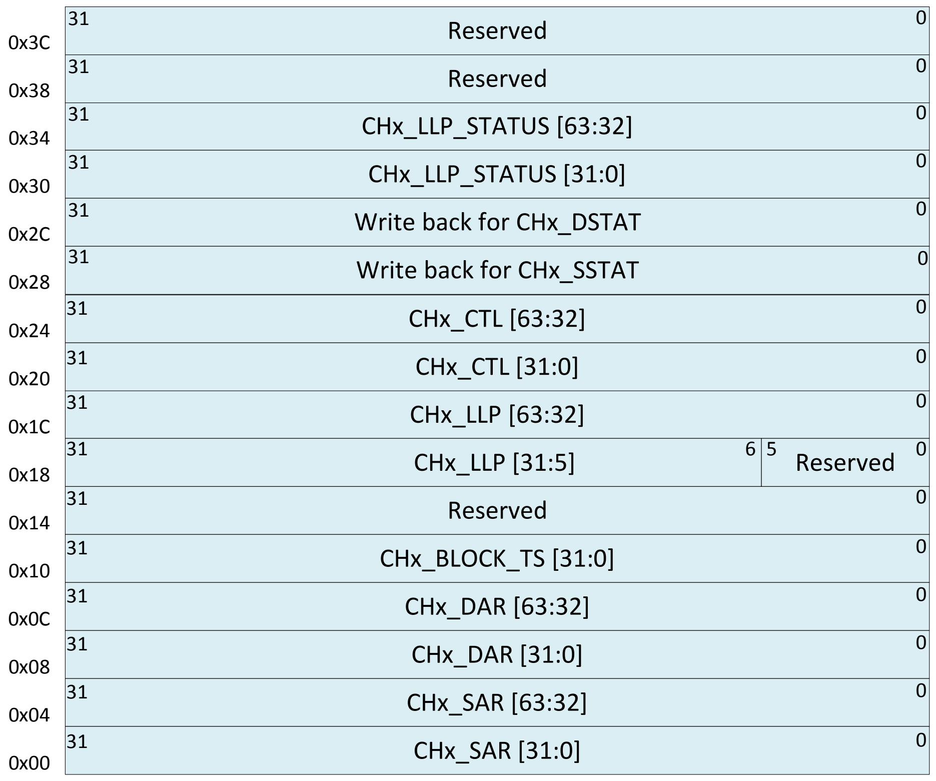
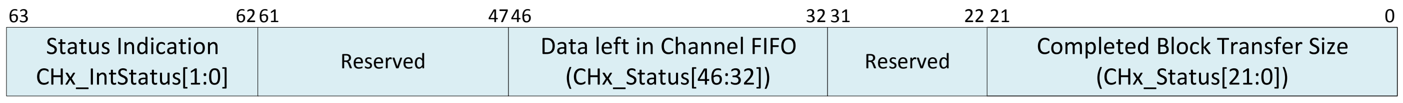


Figure 3-29 CHx_LLP_STATUS Write-Back Field of LLI


63 62 61 47 46 32 31 22 21 0
Status Indication CHx_IntStatus[1:0]
Reserved
Data left in Channel FIFO (CHx_Status[46:32])
Reserved
Completed Block Transfer Size (CHx_Status[21:0])

For more information about programming linked-list-based multi-block transfers, refer to "Programming Flow for Linked-List-Based Multi-Bock Transfer" on page 200.

3.13.2.5 Suspension of Transfers Between Blocks

At the end of every block transfer, a Block Transfer Done Interrupt is asserted if:

- Global Interrupt is enabled (DMAC_CfgReg.INT_EN = 1)
- The channel block transfer completion interrupt is enabled (CHx_InStatus_EnableReg.Enable_BLOCK_TFR_DONE_IntStat = 1 AND CHx_IntSignal_EnableReg. Enable_BLOCK_TFR_DONE_IntSignal = 1 AND CHx_CTL.IOC_BLKTFR = 1)

For Contiguous Address and Auto-Reloading-based multiblock transfers (neither source nor destination peripheral uses Shadow Register or Linked-List-based multiblock transfers), the DMA transfer automatically stalls after the Block Transfer Done Interrupt is asserted, if the Block Transfer Done Interrupt is enabled and unmasked. The DW_axi_dmac does not proceed to the next block transfer until a write to the appropriate field in CHx_IntClearReg register, done by software to clear the channel Block Transfer Done Interrupt, is detected by hardware.

For Contiguous Address and Auto-Reloading-based multi-block transfers (neither source nor destination peripheral uses Shadow Register or Linked-List-based multi-block transfers), the DMA transfer does not stall if either

- Global Interrupt is disabled (DMAC_CfgReg.INT_EN = 0)
- The channel block transfer completion interrupt is not enabled (CHx_InStatus_EnableReg.Enable_BLOCK_TFR_DONE_IntStat = 0 OR CHx_IntSignal_EnableReg. Enable_BLOCK_TFR_DONE_IntSignal = 0 OR CHx_CTL.IOC_BLKTFR = 0)

Channel suspension between blocks is used to ensure that the Block Transfer Done 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 CHx_CFG.SRC_MLTBLK_TYPE and/or CHx_CFG.DST_MLTBLK_TYPE bits before completion of the final block. The CHx_CFG.SRC_MLTBLK_TYPE and/or CHx_CFG.DST_MLTBLK_TYPE bits should be cleared in the Block Transfer Done ISR for the next-to-last block transfer.

3.13.2.6 End of Multi-Block Transfers

If either source or destination peripheral uses Shadow Register or Linked-List-based multi-block transfers, the corresponding last block indication bit, ShadowReg_Or_LLI_Last, in the CHx_CTL_ShadowReg / LLI.CHx_CTL register indicates whether the current block is the last block in the transfer or not. If this bit is set to 1 for the current block, the DW_axi_dmac understands that the current block is the final block in the transfer and completes the DMA transfer operation at the end of current block transfer.

For Contiguous Address and Auto-Reloading-based multiblock transfers (if neither source nor destination peripheral uses Shadow Register or Linked-List-based multi-block transfers), if the corresponding multi-block type selection bits namely CHx_CFG.SRC_MLTBLK_TYPE and/or CHx_CFG.DST_MLTBLK_TYPE bits are seen to be 2'b00 at the end of a block transfer, the DW_axi_dmac understands that the previous block was the final block in the transfer and completes the DMA transfer operation.

3.14 Channel Suspend, Disable, and Abort

Under normal operation, software enables a channel by writing a 1 to the channel enable register, DMAC_ChEnReg.CH_EN, and hardware disables a channel on transfer completion by clearing the DMAC_ChEnReg.CH_EN register.

Software can suspend, disable, or abort a channel before a transfer completes. The suspend, disable, and abort procedures are explained in the following sections.

3.14.1 Channel Suspend

To suspend a channel during DMA transfer:

1. Software writes a 1 to the channel suspend bit CH_SUSP in the channel enable register, DMAC_ChEnReg.
2. DW_axi_dmac gracefully halts all transfers from the source peripheral, after completing all AXI transfers initiated on the source peripheral.
3. DW_axi_dmac sets CHx_IntStatusReg.CH_SRC_SUSPENDED bit to 1 to indicate that source data transfer is suspended and generates the interrupt if it is not masked off.

Note If the channel FIFO is full and the destination peripheral is not requesting data transfer, DW_axi_dmac cannot receive any more data on the corresponding master interface, which could lead to a deadlock.

- The requests initiated by source/destination/LLI state machines and present in the Master Interface Read Address Channel and Write Address Channel FIFO's to be sent on the AXI Master Interface will be sent on the AXI Master Interface even if the Channel Suspend request is initiated. Based on the Master Interface Read Address and Write Address Channel FIFO depth configuration, maximum 8 read/write requests may be initiated and DW_axi_dmac waits for the data/response for these requests also before suspending the channel.

4. DW_axi_dmac transfers all the data in channel FIFO to destination peripheral.

   When CHx_CTL.SRC_TR_WIDTH < CHx_CTL.DST_TR_WIDTH and the DMAC_ChEnReg.CH_SUSP bit is high, there may still be data in the channel FIFO, but not enough to form a single transfer of CHx_CTL.DST_TR_WIDTH. The data remaining in the channel FIFO will be transferred to destination if channel is resumed later which results in filling of more data in channel FIFO.
5. DW_axi_dmac clears channel locking and resets the channel locking settings in the CHx_CFG register.
6. DW_axi_dmac sets the CHx_IntStatusReg.ChLock_Cleared bit to 1 to indicate that channel locking is cleared.
7. DW_axi_dmac sets the CHx_IntStatusReg.CH_SUSPENDED bit to 1 to indicate that the channel is suspended.
8. DW_axi_dmac generates a CH_SUSPENDED interrupt (if it is not masked off).

After a channel suspend, software may either resume the channel after some time, or disable the channel.

3.14.2 Channel Suspend and Resume

To suspend and resume a channel:

1. Follow steps 1 to 4 in Channel Suspend.
2. Software writes a 0 to the channel suspend bit CH_SUSP in the channel enable register, DMAC_ChEnReg.
3. DW_axi_dmac resumes the DMA transfer from the point where it got suspended.

Note Once software initiates the channel suspend procedure by writing a 1 to the channel suspend bit DMAC_ChEnReg.CH_SUSP, writing 0 to DMAC_ChEnReg.CH_SUSP bit to resume the channel before DW_axi_dmac asserts CHx_IntStatusReg.CH_SUSPENDED bit is not allowed. DW_axi_dmac ignores this write operation.

3.14.3 Channel Suspend and Disable Prior to Transfer Completion

To suspend and disable a channel:

1. Follow steps 1 to 4 in Channel Suspend.
2. To disable the suspended channel using software, write a 0 to the channel enable bit (CH_EN) in the channel enable register (DMAC_ChEnReg) after DW_axi_dmac asserts the CHx_IntStatusReg.CH_SUSPENDED bit to 1 to indicate that channel is suspended.

   When CHx_CTL.SRC_TR_WIDTH < CHx_CTL.DST_TR_WIDTH and the DMAC_ChEnReg.CH_SUSP bit is high, there may still be data in the channel FIFO, but not enough to form a single transfer of CHx_CTL.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 and is lost.
3. DW_axi_dmac sets the CHx_IntStatusReg.CH_DISABLED bit to 1 to indicate that the channel is disabled.
4. DW_axi_dmac generates a CH_DISABLED interrupt (if it is not masked off).
5. DW_axi_dmac clears the DMAC_ChEnReg.CH_EN bit to 0.

3.14.4 Channel Disable Prior to Transfer Completion without Suspend

To disable a channel without suspending:

1. Software writes a 0 to the channel enable bit CH_EN in the channel enable register, DMAC_ChEnReg.
2. DW_axi_dmac gracefully halts all transfers from the source peripheral, after completing all AXI transfers initiated on the source peripheral.

Note

- If the channel FIFO is full and the destination peripheral is not requesting data transfer, DW_axi_dmac cannot receive any more data on the corresponding master interface, which could lead to a deadlock.
- The requests initiated by source/destination/LLI state machines and present in the Master Interface Read Address Channel and Write Address Channel FIFO's to be sent on the AXI Master Interface will be sent on the AXI Master Interface even if the Channel Suspend request is initiated. Based on the Master Interface Read Address and Write Address Channel FIFO depth configuration, maximum 8 read/write requests may be initiated and DW_axi_dmac waits for the data/response for these requests also before suspending the channel.

3. DW_axi_dmac transfers all the data in the channel FIFO to the destination peripheral.

   If CHx_CTL.SRC_TR_WIDTH is less than CHx_CTL.DST_TR_WIDTH and the DMAC_ChEnReg.CH_EN bit is low, there may still be data in the channel FIFO, but not enough to form a single transfer of CHx_CTL.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 and is lost.
4. DW_axi_dmac clears channel locking and resets the channel locking settings in the CHx_CFG register.
5. DW_axi_dmac sets the CHx_IntStatusReg.ChLock_Cleared bit to 1 to indicate that channel locking is cleared.
6. DW_axi_dmac sets the CHx_IntStatusReg.CH_DISABLED bit to 1 to indicate that the channel is disabled.
7. DW_axi_dmac generates a CH_DISABLED interrupt (if it is not masked off).
8. DW_axi_dmac clears the DMAC_ChEnReg.CH_EN bit to 0.

Note

Once software initiates a channel disable procedure by writing a 0 to the channel enable bit DMAC_ChEnReg.CH_EN, writing a 1 to the DMAC_ChEnReg.CH_EN bit to re-enable the channel before DW_axi_dmac asserts the CHx_IntStatusReg.CH_DISABLED bit is not allowed. DW_axi_dmac ignores this write operation.

3.14.5 Abnormal Channel Abort

Aborting a channel is not a recommended procedure. This procedure should be used only when software wants to disable a channel without resetting the entire DW_axi_dmac, for example, if a particular channel hangs due to not receiving a response from the corresponding handshaking interface. Before aborting a channel, it is recommended that you first attempt to disable a channel.

To abort a channel:

1. Software writes a 1 to the channel abort bit CH_ABORT in the channel enable register, DMAC_ChEnReg.
2. DW_axi_dmac gracefully halts all transfers from the source/destination peripheral after completing all AXI transfers initiated on the source/destination peripheral.
3. The data in Channel FIFO is flushed and essentially be lost.
4. DW_axi_dmac clears channel locking and resets the channel locking settings in the CHx_CFG register.
5. DW_axi_dmac sets CHx_IntStatusReg.ChLock_Cleared bit to 1 to indicate that channel locking is cleared.
6. DW_axi_dmac sets CHx_IntStatusReg.CH_ABORTED bit to 1 to indicate that channel is aborted.
7. DW_axi_dmac generates CH_ABORTED interrupt if it is not masked off.
8. DW_axi_dmac clears DMAC_ChEnReg.CH_EN bit to 0.

3.15 Debug Interface

The Debug Interface in DW_axi_dmac consists of Status Indication signals and DMAC Hold Control signals.

3.15.1 Slave Interface and DMAC Core Status Indication

DW_axi_dmac supports status indication using two signals on the I/O: slvif_busy and dmac_busy.

- slvif_busy – indicates whether the Slave Interface is busy. This signal is asserted High if there is an active transfer on the slave interface.
- dmac_busy – indicates whether the DMAC Core is busy or not. This signal is asserted High if any of the channels in DW_axi_dmac are in a non-idle state. A non-idle state of the channel corresponds to the following cases.
  - There is an active transfer on the Master Interface (including posted requests) corresponding to one or multiple Channels.
  - Any of the Channels are about to initiate a transfer, which might correspond to waiting for the grant of the master interface from the arbiter.

3.15.2 DMAC Hold Control

DW_axi_dmac supports freezing an operation. This is supported using two signals on the I/O, dmac_hold_req and dmac_hold_ack. The dmac_hold_req signal is the request to put DW_axi_dmac in hold (freeze) mode. Asserting this request puts the entire DW_axi_dmac in freeze mode without violating the AXI protocol. DW_axi_dmac asserts dmac_hold_ack after entering the hold mode.

To put DW_axi_dmac in hold mode:

1. External device (master) asserts dmac_hold_req to put DW_axi_dmac in hold mode.
2. DW_axi_dmac halts all transfers from the source peripheral of all channels.
   - DW_axi_dmac does not initiate any new read request on the AXI read address channel.
   - DW_axi_dmac de-asserts arvalid_m*N* output once arready_m*N* is received.
   - DW_axi_dmac de-asserts rready_m*N* output so that no new data is received on the AXI read data channel and hence no new data is written to the channel FIFO.

Note There may be outstanding read requests of different channels on the master interfaces, for which data transfer is not completed. This can lead to timeout issues in the interconnect.

3. DW_axi_dmac halts all transfers to the destination peripheral of all channels, which means the following is true:
   - DW_axi_dmac does not initiate any new write request on AXI write address channel.
   - DW_axi_dmac de-asserts awvalid_m*N* and wvalid_m*N* output when awready_m*N* and wready_m*N* is received.
   - DW_axi_dmac de-asserts bready_m*N* output when data transfer on the AXI write data channel is completed.

   Note There may be write requests from many channels on the master interfaces, waiting for a response. This can lead to timeout issues in the interconnect.

4. The channel FIFOs are not guaranteed to be emptied. There may still be data in the channel FIFO after entering the hold mode.
5. DW_axi_dmac asserts dmac_hold_ack to indicate the entry to hold mode.
6. To exit the DMAC hold mode, dmac_hold_req can be de-asserted after DW_axi_dmac asserts dmac_hold_ack.
7. After after dmac_hold_req is de-asserted, DW_axi_dmac de-asserts dmac_hold_ack.

Note When a DMAC hold request is initiated by asserting dmac_hold_req, de-asserting dmac_hold_req to exit the DMAC hold mode before DW_axi_dmac asserts dmac_hold_ack is not allowed. DW_axi_dmac ignores this operation.

3.15.3 Error Handling

DW_axi_dmac can receive an error response from a source peripheral, a destination peripheral, or an LLP peripheral. Upon occurrence of an error in an AXI transfer (source or destination data transfer or status fetch, LLI fetch, or LLI write-back), the following occurs:

1. DMA transfer in progress stops gracefully.
2. If channel locking to arbiter is enabled at any transfer level, the locking is cleared.
3. Error Status bits in CHx_IntStatusReg register is updated if not masked off.
4. Source/Destination Transaction Completion, Block Transfer Completion and DMA Transfer Completion Status bits in CHx_IntStatusReg register is cleared to '0' if not masked off.
5. Interrupt Status registers are updated and an interrupt is issued, if not masked.
6. Relevant channel is disabled and CH_DISABLED Status bit is set to '1'.

If an error occurs on a source or destination peripheral, the steps that need to access that peripheral are not performed. If multiple channels are enabled, only the channel where the AXI error was detected is disabled.

The FIFO pointers are reset, therefore, the previous FIFO contents 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 or partial block transfer has to be re-initiated by the software in order to be successfully completed.

If hardware handshaking is enabled for source or destination, the DW_axi_dmac does not signal the end of a transfer. If a request from a peripheral is active when the error occurs (that is, dma_req is high), the channel is disabled without the DMA ever asserting dma_ack and 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.

If software handshaking is enabled for SRC/DST, the DW_axi_dmac does not signal the end of a transfer. In practice, this means that if a request from a peripheral is active when the error occurs (all or some of SWHS_Req_Src, SWHS_SglReq_Src, SWHS_Lst_Src, SWHS_Req_Dst, SWHS_SglReq_Dst, SWHS_Lst_Dst bits are high), the channel is disabled without the DMA ever clearing the request bits to 0 in CHx_SWHSSrcReg and/or CHx_SWHSDstReg register.

4 Parameters

This chapter describes the parameters used by the DW_axi_dmac. You use coreConsultant or coreAssembler to configure the following parameters and generate the configured code.

Attention When using coreConsultant or coreAssembler, you can right-click on a parameter label to access a "What's This" popup dialog that will tell you the details for that particular parameter. The information in each What's This dialog essentially matches the information in the parameter descriptions below.

4.1 Parameter Descriptions

Table 4-1 lists the DW_axi_dmac parameter descriptions. In this table, the values 0 and 1 occasionally appear in parentheses in the descriptions for the parameters. These are the logical values for parameter settings that appear in the coreConsultant GUI as check boxes, drop-down lists, a multiple selection, and so on.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
DW_axi_dmac Global Configuration	
DW_axi_dmac ID	**Parameter Name:** DMAX_ID_NUM **Legal Values:** 0x0 to 0xffffffff **Default Value:** 0x0 **Dependencies:** None **Description**: A 64-bit value that is hard-wired and read back by a read to the DW_axi_dmac ID Register (DMAC_IDReg).
DW_axi_dmac Component Version	**Parameter Name:** DMAX_COMP_VER **Legal Values:** 0x0 to 0xffffffff **Default Value:** 0x0 **Dependencies:** None **Description**: A 64-bit value that is hardwired and read back by a read to the DW_axi_dmac Component Version Register (DMAC_CompVerReg).

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
DW_axi_dmac Master Interface Protocol	**Parameter Name:** DMAX_MSTIF_MODE **Legal Values:** AXI3 (0), AXI4 (1) **Default Value:** AXI3 (0) **Dependencies:** None **Description**: The protocol used for the AXI master interface.
DW_axi_dmac Slave Interface Protocol	**Parameter Name:** DMAX_SLVIF_MODE **Legal Values:** AHB (0), AXI4-Lite (1), APB3 (2) **Default Value:** AHB (0) **Dependencies:** None **Description**: The protocol used for the AXI slave interface. Only AHB is supported in this version.
Number of AXI Master Interfaces	**Parameter Name:** DMAX_NUM_MASTER_IF **Legal Values:** 1 to 2 **Default Value:** 1 **Dependencies:** None **Description**: Creates the specified number of AXI master interfaces. A channel source or destination device can be programmed to be on any of the configured AXI layers attached to the AXI master interface. This setting determines if the AXI Master2 interface signals are present on the I/O or not. AXI Master1 interface signals are always present.
Master Interface 1 Outstanding Request Limit	**Parameter Name:** DMAX_MSTIF1_OSR_LMT **Legal Values:** 16, 32, 48, 64, 80, 96,112 and128 **Default Value:** DMAX_NUM_CHANNELS*16 **Dependencies:** None **Description:** Sets Outstanding Requests Limit on Master 1 AXI interface for Write and Read address channel. Maximum value of Outstanding Transaction is (16 * Number of DMAC Channels).
Master Interface 2 Outstanding Request Limit	**Parameter Name**: DMAX_MSTIF2_OSR_LMT **Legal Values**: 16, 32, 48, 64, 80, 96,112 and128 **Default Value**: DMAX_NUM_CHANNELS*16 **Dependencies**: DMAX_NUM_MASTER_IF = 2 **Description**: Sets Outstanding Requests Limit on Master 2 AXI interface for Write and Read address channel. Maximum value of Outstanding Transaction is (16 * Number of DMAC Channels).

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Number of DMA Channels	**Parameter Name:** DMAX_NUM_CHANNELS **Legal Values:** 1 to 8 **Default Value:** 1 **Dependencies:** None **Description**: Creates the specified number of DW_axi_dmac channels, each of which is unidirectional and transfers data from the channel source to the channel destination. The channel source and destination AXI layer, system address, and handshaking interface are under software control.
Number of Handshaking Interfaces	**Parameter Name:** DMAX_NUM_HS_IF **Legal Values:** 0 to 16 **Default Value:** 2 **Dependencies:** None **Description**: Creates the specified number of hardware handshaking interfaces. DW_axi_dmac can be programmed 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.
Interrupt Pins to Appear as Outputs?	**Parameter Name:** DMAX_INTR_IO_TYPE **Legal Values:** COMBINED_ONLY (1), CHANNEL_AND_COMMONREG (2), ALL_INTERRUPT_OUTPUTS (3) **Default Value:** COMBINED_ONLY (1) **Dependencies:** None **Description**: Selects which interrupt-related signals appear as outputs on the design. ■ **COMBINED_ONLY:** Only “intr” output exists. Bitwise OR of all bits of DMAC_IntStatusReg register is driven onto “intr” output. ■ **CHANNEL_AND_COMMONREG:** “intr_ch” and “intr_cmnreg” outputs exist. Bitwise OR of all the corresponding channel bits of DMAC_IntStatusReg register is driven onto the respective “intr_ch” output. Bitwise OR of all the bits of DMAC_CommonReg_IntStatusReg register is driven onto the respective “intr_cmnreg” output. ■ **ALL_INTERRUPT_OUTPUTS:** “intr”, “intr_ch” and “intr_cmnreg” outputs exist. Bitwise OR of all bits of DMAC_IntStatusReg register is driven onto “intr” output. Bitwise OR of all the corresponding channel bits of DMAC_IntStatusReg register is driven onto the respective “intr_ch” output. Bitwise OR of all the bits of DMAC_CommonReg_IntStatusReg register is driven onto the respective “intr_cmnreg” output.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Synchronize Interrupt Outputs to Slave Interface Clock?	**Parameter Name:** DMAX_INTR_SYNC2SLVCLK **Legal Values:** Yes (1) or No (0) **Default Value:** No (0) **Dependencies:** DMAX_SLVIF_CLOCK_MODE = 1. If DMAX_SLVIF_CLOCK_MODE = 0, the slave interface is synchronous to the core clock (dmac_core_clock) and there is no need for additional synchronization. **Description**: Interrupt output, by default, is synchronous to dmac_core_clock. When this parameter is set to 1, the interrupt output pins are synchronous to the slave interface clock. In this case, additional synchronizers are added inside DW_axi_dmac.
Include Status Indication Output on Slave Bus Interface?	**Parameter Name:** DMAX_SLVIF_STATUS_OP _EN **Legal Values:** True (1) or False (0) **Default Value:** True (1) **Dependencies:** None **Description**: By default, this option creates a slave interface status indication (Busy/Idle) signal on the I/O, which is synchronous to the slave interface clock. When you set this option to False (0), the signal is not included on the I/O.
Include DMAC Internal Status Indication Output? (Synchronous to dmac_core_clock)	**Parameter Name:** DMAX_CORE_STATUS_OP _EN **Legal Values:** True (1) or False (0) **Default Value:** True (1) **Dependencies:** None **Description**: By default, this option creates a DMAC internal status indication (Busy/Idle) signal on the I/O, which is synchronous to dmac_core_clock. When you set this option to False (0), the signal is not included on the I/O.
Include DMAC Hold Request Input & Hold Acknowledgement Output? (Synchronous to dmac_core_clock)	**Parameter Name:** DMAX_HOLD_IO _EN **Legal Values:** True (1) or False (0) **Default Value:** True (1) **Dependencies:** None **Description**: By default, this option creates a DMAC Hold Request Input signal (dmac_hold_req) and DMAC Hold Acknowledgement Output signal (dmac_hold_ack) on the I/O, which are synchronous to dmac_core_clock. When you set this option to False (0), the signal is not included on the I/O.
Include Debug Ports?	**Parameter Name:** DMAX_DEBUG_PORTS_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** None **Description:** Parameter to enable debug_* ports in DW_axi_dmac

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Include Channel Abort Logic?	**Parameter Name:** DMAX_CH_ABORT_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** None **Description**: Setting this option to 1 enables logic to support Channel Abort feature in DW_axi_dmac.
Clock Parameters	
DW_axi_dmac Slave Interface Clocking Mode	**Parameter Name:** DMAX_SLVIF_CLOCK_MODE **Legal Values:** Synchronous (0), Asynchronous (1) **Default Value:** 0 **Dependencies:** None **Description**: Selects the relationship between the Slave Interface clock (AHB/AXI4-Lite/APB3) and Core clock. ■ 0 – Slave Interface clock is synchronous to core clock ■ 1 – Slave Interface clock is asynchronous to core clock
Slave to Core Synchronization Depth	**Parameter Name:** DMAX_S_2_C_SYNC_DEPTH **Legal Values:** 1, 2, 3, and 4 **Default Value:** 2 **Dependencies:** DMAX_SLVIF_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Slave clock domain to DW_axi_dmac Core Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge
Core to Slave Synchronization Depth	**Parameter Name:** DMAX_C_2_S_SYNC_DEPTH **Legal Values:** 1, 2, 3 and 4 **Default Value:** 2 **Dependencies:** DMAX_SLVIF_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Core clock domain to Slave Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
DW_axi_dmac Master1 Interface Clocking Mode	**Parameter Name:** DMAX_MSTIF1_CLOCK_MODE **Legal Values:** Synchronous (0), Asynchronous (1) **Default Value:** Synchronous (0) **Dependencies:** None **Description**: Selects the relationship between the Master1 interface clock (aclk1) and the core clock. ■ 0 – Master1 interface clock is synchronous to core clock ■ 1 – Master 1 Interface clock is asynchronous to core clock
Master 1 to Core Synchronization Depth	**Parameter Name:** DMAX_M1_2_C_SYNC_DEPTH **Legal Values:** 1, 2, 3 and 4 **Default Value:** 2 **Dependencies:** DMAX_MSTIF1_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Master 1 clock domain to DW_axi_dmac Core Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge Synchronization Depths used for Master 1 interface FIFO Signals are: ■ 1 – One-stage synchronization ■ 2 – Two-stage synchronization ■ 3 – Three-stage synchronization ■ 4 – Four-stage synchronization

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Core to Master 1 Synchronization Depth	**Parameter Name:** DMAX_C_2_M1_SYNC_DEPTH **Legal Values:** 1, 2, 3 and 4 **Default Value:** 2 **Dependencies:** MSTIF1_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Core clock domain to Master 1 Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge Synchronization Depths used for Master 1 interface FIFO Signals are: ■ 1 – One-stage synchronization ■ 2 – Two-stage synchronization ■ 3 – Three-stage synchronization ■ 4 – Four-stage synchronization
DW_axi_dmac Master2 Interface Clocking Mode	**Parameter Name:** DMAX_MSTIF2_CLOCK_MODE **Legal Values:** Synchronous (0), Asynchronous (1) **Default Value:** 0 **Dependencies:** DMAX_NUM_MASTER_IF = 2 **Description**: Selects the relationship between the Master 2 Interface clock (aclk2) and Core clock. ■ 0 – Master 2 Interface clock is synchronous to core clock ■ 1 – Master 2 Interface clock is asynchronous to core clock

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Master 2 to Core Synchronization Depth	**Parameter Name:** DMAX_M2_2_C_SYNC_DEPTH **Legal Values:** 1, 2, 3 and 4 **Default Value:** 2 **Dependencies:** DMAX_MSTIF2_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Master 2 clock domain to DW_axi_dmac Core Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge Synchronization Depths used for Master 2 interface FIFO Signals are: ■ 1 – One-stage synchronization ■ 2 – Two-stage synchronization ■ 3 – Three-stage synchronization ■ 4 – Four-stage synchronization
Core to Master 2 Synchronization Depth	**Parameter Name:** DMAX_C_2_M2_SYNC_DEPTH **Legal Values:** 1, 2, 3 and 4 **Default Value:** 2 **Dependencies:** DMAX_MSTIF2_CLOCK_MODE = 1 **Description**: Defines the number of synchronization register stages for signals passing from the DW_axi_dmac Core clock domain to Master 2 Clock domain. ■ 1 – Two-stage synchronization: First stage - negative edge; Second stage - positive edge ■ 2 – Two-stage synchronization, both stages positive edge ■ 3 – Three-stage synchronization, all stages positive edge ■ 4 – Four-stage synchronization, all stages positive edge Synchronization Depths used for Master 2 interface FIFO Signals are: ■ 1– One-stage synchronization ■ 2 – Two-stage synchronization ■ 3 – Three-stage synchronization ■ 4 – Four-stage synchronization

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
AMBA Layer Parameters	
Statically Configure Endian Scheme of Master Interfaces?	**Parameter Name:** DMAX_STATIC_ENDIAN_SELECT_MSTIF **Legal Values:** True (1) or False (0) **Default Value:** True (1) **Dependencies:** None **Description**: The endian scheme of the DW_axi_dmac master interfaces 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 controls the endianness of all AXI master interfaces. For the dynamic case, there is an individual pin for each of the AXI master interfaces. If this option is selected, the endianness is configured based on the value of DMAX_ENDIAN_FORMAT_MSTIF. Otherwise, it is decided based on the value of input pin, dmac_endian_format_mstif*N*.
Master Interface Endian Format	**Parameter Name:** DMAX_ENDIAN_FORMAT_MSTIF **Legal Values:** 0 to 1 **Default Value:** 0 **Dependencies:** This parameter is enabled only if DMAX_STATIC_ENDIAN_SELECT_MSTIF is set to true. **Description**: DMAX_ENDIAN_FORMAT_MSTIF values for different AXI master interface endian formats are as follows: ■ 0 = Little Endian ■ 1 = Big Endian BE-8

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Include Little Endian Scheme Selection Pin for LLI Access on AXI Master Interfaces?	**Parameter Name:** DMAX_LLI_ENDIAN_SELECTION_PIN_EN **Legal Values:** 0 to 1 **Default Value:** 0 **Dependencies:** This parameter is disabled (0) if: ■ DMAX_STATIC_ENDIAN_SELECT_MSTIF = 1; AND DMAX_ENDIAN_FORMAT_MSTIF = 0 OR ■ Multi-block type is hardcoded to non-linked-list-based schemes for all channels. This parameter is enabled only if: ■ DMAX_STATIC_ENDIAN_SELECT_MSTIF = 1; AND DMAX_ENDIAN_FORMAT_MSTIF = 1; OR ■ DMAX_STATIC_ENDIAN_SELECT_MSTIF = 0; AND dmac_endian_format_mstif*N* = 1 **Description**: If this parameter is enabled, additional inputs are enabled to control the endian scheme used for LLI access. An individual pin is added for each of the AXI master interfaces. LLI access on each AXI master interface can be independently configured to support the big endian scheme (BE-8) based on the endian scheme selected for that particular master interface, or the little endian scheme irrespective of the endian scheme selected for that particular master interface. ■ 0 = Endian scheme used for LLI access is same as that used for data access for Master1/Master2 interfaces ■ 1 = Endian scheme used for LLI access is same as that used for data access for Master1/Master2 interfaces; or ■ Little Endian method is used for LLI access irrespective of the endian scheme used for data access, depending on the value of the dmac_le_select_lli_mstif*N* input.
Slave Interface Data Bus Width	**Parameter Name:** DMAX_S_DATA_WIDTH **Legal Values:** 32 bits, 64 bits **Default Value:** 32 bits **Dependencies:** For the APB3 interface, only 32 bit bus width is supported. **Description**: Specifies the data bus width for the AHB/APB3/AXI4-Lite slave interface.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Master Interface Address Bus Width	**Parameter Name:** DMAX_M_ADDR_WIDTH **Legal Values:** 32 bits to 64 bits **Default Value:** 32 bits **Dependencies:** None **Description**: AXI Master1 interface address bus width.
Master Interface Data Bus Width	**Parameter Name:** DMAX_M_DATA_WIDTH **Legal Values:** 32, 64, 128, 256, or 512 bits **Default Value:** 32 bits **Dependencies:** None **Description**: AXI master interface data bus width.
Master Interface ID Width	**Parameter Name:** DMAX_M_ID_WIDTH **Legal Values:** 1 to 12 bits **Default Value:** 1 bit **Dependencies:** None **Description**: AXI master interface ID width (common for awid_m*N*, arid_m*N*, wid_m*N*, and rid_m*N*) **NOTE:** The minimum value of DMAX_M_ID_WIDTH is equal to log2(DMAX_NUM_CHANNELS) if multiblock transfer type is hardcoded to non-linked-list-based schemes, otherwise, [log2(DMAX_NUM_CHANNELS)] +1.
Master Interface Burst Length Width	**Parameter Name:** DMAX_M_BURSTLEN_WIDTH **Legal Values:** 4 to 8 bits **Default Value:** 4 bits **Dependencies:** None **Description**: AXI master interface burst length (arlen_m*N*/awlen_m*N*) width.
Master Interface Read/Write Address Channel FIFO Depth	**Parameter Name:** DMAX_M_ADDR_FIFO_DEPTH **Legal Values:** 4,8 locations **Default Value:** 4 **Dependencies:** None **Description**: AXI master interface read/write address channel FIFO depth. Setting appropriate value based on the system requirement allows some logic optimization of the implementation.
Master Interface Read/Write Data Channel FIFO Depth	**Parameter Name:** DMAX_M_DATA_FIFO_DEPTH **Legal Values:** 4,8 locations **Default Value:** 4 **Dependencies:** None **Description**: AXI master interface read/write data channel FIFO depth. Setting appropriate value based on the system requirement allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	**Parameter Definition**
Master Interface Write Response Channel FIFO Depth	**Parameter Name:** DMAX_M_BRESP_FIFO_DEPTH **Legal Values:** 4,8 locations **Default Value:** 4 **Dependencies:** None **Description**: AXI master interface write response channel FIFO depth. Setting appropriate value based on the system requirement allows some logic optimization of the implementation.
Include Transfer Completion Indication Signal on Master Interface?	**Parameter Name:** DMAX_ENABLE_LAST_WRITE **Legal Values:** 0 to 1 **Default Value:** 0 **Dependencies:** None **Description**: Enables the additional handshaking signal LAST_WRITEN on all AXI master interfaces. The last_write_m*N* signal is asserted on the last data phase of every destination block transfer and remains asserted until the last data phase completes.
Channel x Configuration, where x = 1 to 8 **(NOTE:** The channel-specific coreConsultant parameters are enabled only for configured channels.)	
Channel x FIFO depth	**Parameter Name:** DMAX_CH*x*_FIFO_DEPTH **Legal Values:** 4,8,16, 32, 64,128, 256 locations of width DMAX_CH*x*_FIFO_WIDTH DMAX_CH*x*_FIFO_WIDTH is an internal parameter and is defined as follows: ■ If DMAX_CHx_STW != 0 && DMAX_CHx_DTW != 0 DMAX_CHx_FIFO_WIDTH = max of (DMAX_CHx_STW, DMAX_CHx_DTW) ■ If DMAX_CHx_STW == 0 || DMAX_CHx_DTW == 0 DMAX_CHx_FIFO_WIDTH = DMAX_M_DATA_WIDTH **Default Value:** 8 **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Channel x FIFO depth. Setting appropriate value based on the system requirement allows some logic optimization of the implementation.
Maximum Value of Burst Transaction Size	**Parameter Name:** DMAX_CH*x*_MAX_MSIZE **Legal Values:** 1,4,8,16, 32, 64, 128, 256, 512, or 1024 **Default Value:** 8 **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Maximum value of burst transaction size that can be programmed for channel x (CHx_CTL.SRC_MSIZE and CHx_CTL.DST_MSIZE). Setting appropriate value based on the system requirement allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Maximum Value of Burst Block Size in Source Transfer Width	**Parameter Name:** DMAX_CH*x*_MAX_BLOCK_TS **Legal Values:** 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767, 65535, 131071, 262143, 524287, 1048575, 2097151, 4194303 **Default Value:** 31 **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: The description of this parameter depends on what is assigned as the flow controller. ■ If DW_axi_dmac is the flow controller: Maximum block size, in multiples of source transfer width (CHx_BLOCK_TS.BLOCK_TS), that can be programmed for channel *x*. A programmed value greater than this will result in inconsistent behavior. ■ If the source or destination peripheral is assigned as the flow controller: In this case, the blocks can be greater than DMAX_CHx_MAX_BLOCK_TS in size, but the logic that keeps track of the size of a block saturates at DMAX_CHx_MAX_BLOCK_TS. This does not result in inconsistent behavior, but a read back by software of the block size is incorrect when the block size exceeds the saturated value. Setting appropriate value based on the system requirement allows some logic optimization of the implementation.
Maximum Value of AMBA Burst Length	**Parameter Name:** DMAX_CH*x*_MAX_AMBA_BURST_LENGTH **Legal Values:** 1, 4, 8, 16, 32, 64, 128, 256 **Default Value:** 8 **Dependencies:** Value of DMAX_M_BURSTLEN_WIDTH determines the range of this parameter. ■ DMAX_CH*x*_MAX_AMBA_BURST_LENGTH <= DMAX_CH*x*_MAX_BLOCK_TS ■ DMAX_CH*x*_MAX_AMBA_BURST_LENGTH <= DMAX_CH*x*_MAX_MSIZE ■ *x* <= DMAX_NUM_CHANNELS **Description**: Setting appropriate value based on the system requirement allows some logic optimization of the implementation by reducing the internal FIFO depth requirement.
Include Logic to Enable Channel Locking on Channel x?	**Parameter Name:** DMAX_CH*x*_LOCK_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: If set to 1, includes logic to enable channel locking on channel *x*. When set to 1, software can program the DW_axi_dmac to lock the arbitration for the master bus interface over the DMA transfer, block transfer, or transaction. Disabling this option allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Hardcode Channel *x* Transfer Type and Flow Control Device	**Parameter Name:** DMAX_CH*x*_TT_FC **Legal Values:** 0 to 8 MEM2MEM_DMAC (0): Memory to Memory Transfer with DW_axi_dmac as flow controller MEM2PER_DMAC (1): Memory to Peripheral Transfer with DW_axi_dmac as flow controller PER2MEM_DMAC (2): Peripheral to Memory Transfer with DW_axi_dmac as flow controller PER2PER_DMAC (3): Peripheral to Peripheral Transfer with DW_axi_dmac as flow controller PER2MEM_SRCPER (4): Peripheral to Memory Transfer with Source Peripheral as flow controller PER2PER_SRCPER (5): Peripheral to Peripheral Transfer with Source Peripheral as flow controller MEM2PER_DSTPER (6): Memory to Peripheral Transfer with Destination Peripheral as flow controller PER2PER_DSTPER (7): Peripheral to Peripheral Transfer with Destination Peripheral as flow controller NO_HARDCODE (8): Transfer Type and Flow Controller depends on the value of CHx_CFG.TT_FC field. **Default Value:** NO_HARDCODE (8) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Hardcodes the transfer type and flow control peripheral for the channel. If NO_HARDCODE is selected, then the transfer type and flow control device is not hardcoded, and software selects the transfer type and flow control device for a DMA transfer. Hardcoding the transfer type and flow control device allows some logic optimization of the implementation.
Hardcode the Master Interface Attached to the Source of Channel *x?*	**Parameter Name:** DMAX_CH*x*_SMS **Legal Values:** MASTER_1 (0), MASTER_2 (1), NO_HARDCODE (2) **Default Value:** ■ If DMAX_NUM_MASTER_IF = 1, MASTER_1 (0) ■ If DMAX_NUM_MASTER_IF > 1, NO_HARDCODE (2) **Dependencies:** *x* <= DMAX_NUM_CHANNELS AND *x* <= DMAX_NUM_CHANNELS **Description**: Hardcodes the AXI master interface attached to the source of channel *x*. If this is not hardcoded, software can program the source of channel *x* to be attached to any of the configured layers. Hardcoding this value allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Hardcode the Master Interface Attached to the Destination of Channel *x?*	**Parameter Name:** DMAX_CH*x*_DMS **Legal Values:** MASTER_1 (0), MASTER_2 (1), NO_HARDCODE (2) **Default Value:** ■ If DMAX_NUM_MASTER_IF = 1, MASTER_1 (0) ■ If DMAX_NUM_MASTER_IF > 1, NO_HARDCODE (2) **Dependencies:** *x* <= DMAX_NUM_CHANNELS AND *x* <= DMAX_NUM_CHANNELS **Description**: Hardcode the AXI master interface attached to the channel *x* destination. If this is not hardcoded, software can program the destination of channel *x* to be attached to any of the configured layers. Hardcoding this value allows some logic optimization of the implementation.
Hardcode Channel *x*'s Source Transfer Width	**Parameter Name:** DMAX_CH*x*_STW **Legal Values:** BYTE (8), HALFWORD (16), WORD (32), TWO_WORD (64), FOUR_WORD (128), EIGHT_WORD (256), SIXTEEN_WORD (512) or NO HARDCODE (0) **Default Value:** WORD (32) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Hardcode the source transfer width for transfers from the source of channel *x*. If this is not hardcoded, software can program the source transfer width. Hardcoding the source transfer width allows some logic optimization of the implementation.
Hardcode Channel *x*'s Destination Transfer Width	**Parameter Name:** DMAX_CH*x*_DTW **Legal Values:** BYTE (8), HALFWORD (16), WORD (32), TWO_WORD (64), FOUR_WORD (128), EIGHT_WORD (256), SIXTEEN_WORD (512) or NO HARDCODE (0) **Default Value:** WORD (32) **Dependencies:** *x* <= DMAX_NUM_CHANNELS (See NOTE above). **Description**: Hardcode the destination transfer width for transfers to the destination of channel *x*. If this is not hardcoded, software can program the destination transfer width. Hardcoding the destination transfer width allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Enable LLI Prefetching on Channel *x*?	**Parameter Name:** DMAX_CH*x*_LLI_PREFETCH_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** ■ If DMAX_CH*x*_MULTI_BLK_EN = 0, this option is disabled. ■ If DMAX_CH*x*_MULTI_BLK_TYPE is hardcoded to a value that does not use LLI, this option is disabled. ■ *x* <= DMAX_NUM_CHANNELS **Description**: Set this parameter to 1 to enable the LLI prefetching logic on channel *x*. If this parameter is enabled, DW_axi_dmac tries to fetch **one** LLI in advance (while the data transfer corresponding to the previous LLI is in progress) and store it internally, increasing bus utilization. Enabling this parameter does not, however, guarantee that LLI will be always pre-fetched. This parameter needs to be enabled only if linked-list-based multiblock transfer is used. Setting this parameter to 0 allows some logic optimization of the implementation.
Include Shadow Registers on Channel *x*?	**Parameter Name:** DMAX_CH*x*_SHADOW_REG_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** ■ If DMAX_CH*x*_MULTI_BLK_EN = 0, this option is disabled. ■ If DMAX_CH*x*_MULTI_BLK_TYPE is hardcoded to a value that does not use shadow, this option is disabled. ■ *x* <= DMAX_NUM_CHANNELS **Description**: Set this parameter to 1 to include shadow registers on channel *x*. This parameter needs to be enabled only if shadow-register-based multiblock transfer is used. Setting this parameter to 0 allows some logic optimization of the implementation.
Include Logic to Enable Multiblock DMA Transfers on Channel *x*?	**Parameter Name:** DMAX_CH*x*_MULTI_BLK_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Includes or excludes logic to enable multiblock DMA transfers on channel *x*. If this option is set to 0, hardware hardwires channel *x* to perform only single block transfers. Setting this parameter to 0 allows some logic optimization of the implementation.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Hardcode Multiblock Transfer Type?	**Parameter Name:** DMAX_CH*x*_MULTI_BLK_TYPE **Legal Values:** 0 to 15 NO_HARDCODE (0) – Allows all types of multiblock transfers. MLTBLK_TFR_TYPE 1 (1) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 2 (2) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 3 (3) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 4 (4) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 5 (5) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 6 (6) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 7 (7) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 8 (8) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 9 (9) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 10 (10) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 11 (11) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 12 (12) – Refer Table 3-3 for details MLTBLK_TFR_TYPE 13 (13) – Refer Table 3-3 for details **Default Value:** NO_HARDCODE (0) ■ **Dependencies:** ■ DMAX_CHx_MULTI_BLK_EN = 1 ■ *x* <= DMAX_NUM_CHANNELS **Description**: Hardcode the type of multiblock transfers DW_axi_dmac can perform. This results in some logic optimization of the implementation.
Hardcode the Master Interface Attached to the LLP Peripheral of Channel *x*?	**Parameter Name:** DMAX_CH*x*_LMS **Legal Values:** MASTER_1 (0), MASTER_2 (1), NO_HARDCODE (2) **Default Value:** ■ MASTER_1 (0) – if DMAX_NUM_MASTER_IF = 1 ■ NO_HARDCODE – if DMAX_NUM_MASTER_IF > 1 **Dependencies:** ■ DMAX_NUM_MASTER_IF > 1 and (DMAX_CH*x*_MULTI_BLK_TYPE = 0 or DMAX_CH*x*_MULTI_BLK_TYPE >= 9) ■ *x* <= DMAX_NUM_CHANNELS **Description**: Hardcode the AXI master interface attached to the peripheral that stores the LLI information for channel *x*. If this is not hardcoded, 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 access always uses the burst size (arsize/awsize), which is the same as the data bus width. LLI access cannot be changed or programmed to anything other than this value. Burst length (awlen/arlen) is chosen based on the data bus width so that the access doesn't cross one complete LLI structure of 64 bytes. DW_axi_dmac fetches the entire LLI (40 bytes) in one AXI burst, if the burst length is not limited by other settings.

Table 4-1 DW_axi_dmac Parameters

Field Label	Parameter Definition
Fetch Status from Source of Channel x	**Parameter Name:** DMAX_CHx_SRC_STAT_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Include or exclude logic to fetch the status from the source peripheral of channel *x* pointed to by the content of the CHx_SSTATAR register, and store it in the CHx_SSTAT register, if the CHx_CTL.SRC_STAT_EN bit is set to 1. This value is written back to the CHx_SSTAT location of the linked list at the end of each block transfer, if DMAX_CHx_LLI_WB_EN is set to 1, and if a linked-list-based multiblock transfer is used by either the source or the destination peripheral. Setting this parameter to 0 allows some logic optimization of the implementation.
Include the Logic to Fetch Status from Destination of Channel x	**Parameter Name:** DMAX_CHx_DST_STAT_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** *x* <= DMAX_NUM_CHANNELS **Description**: Include or exclude logic to fetch the status from the destination peripheral of channel *x* pointed to by the content of CHx_DSTATAR register and stores it in the CHx_DSTAT register, if the CHx_CTL.DST_STAT_EN bit is set to 1. This value is written back to the CHx_DSTAT location of the linked list at the end of each block transfer, if DMAX_CHx_LLI_WB_EN is set to 1, and if linked-list-based multiblock transfer is used by either the source or the destination peripheral. Setting this parameter to 0 allows some logic optimization of the implementation.
Include Logic to Enable Register Write-back After Each Block Transfer?	**Parameter Name:** DMAX_CHx_LLI_WB_EN **Legal Values:** True (1) or False (0) **Default Value:** False (0) **Dependencies:** ■ If DMAX_CHx_MULTI_BLK_TYPE is hardcoded to a value that does not use LLI, this option is disabled. ■ *x* <= DMAX_NUM_CHANNELS **Description**: Include or exclude logic to enable write-back of the CHx_CTL, CHx_LLP_STATUS, CHx_SSTAT and CHx_DSTAT registers at the end of every block transfer. Write back happens only if linked-list-based multiblock transfer is used by either the source or the destination peripheral. Setting this parameter to 0 allows some logic optimization of the implementation.

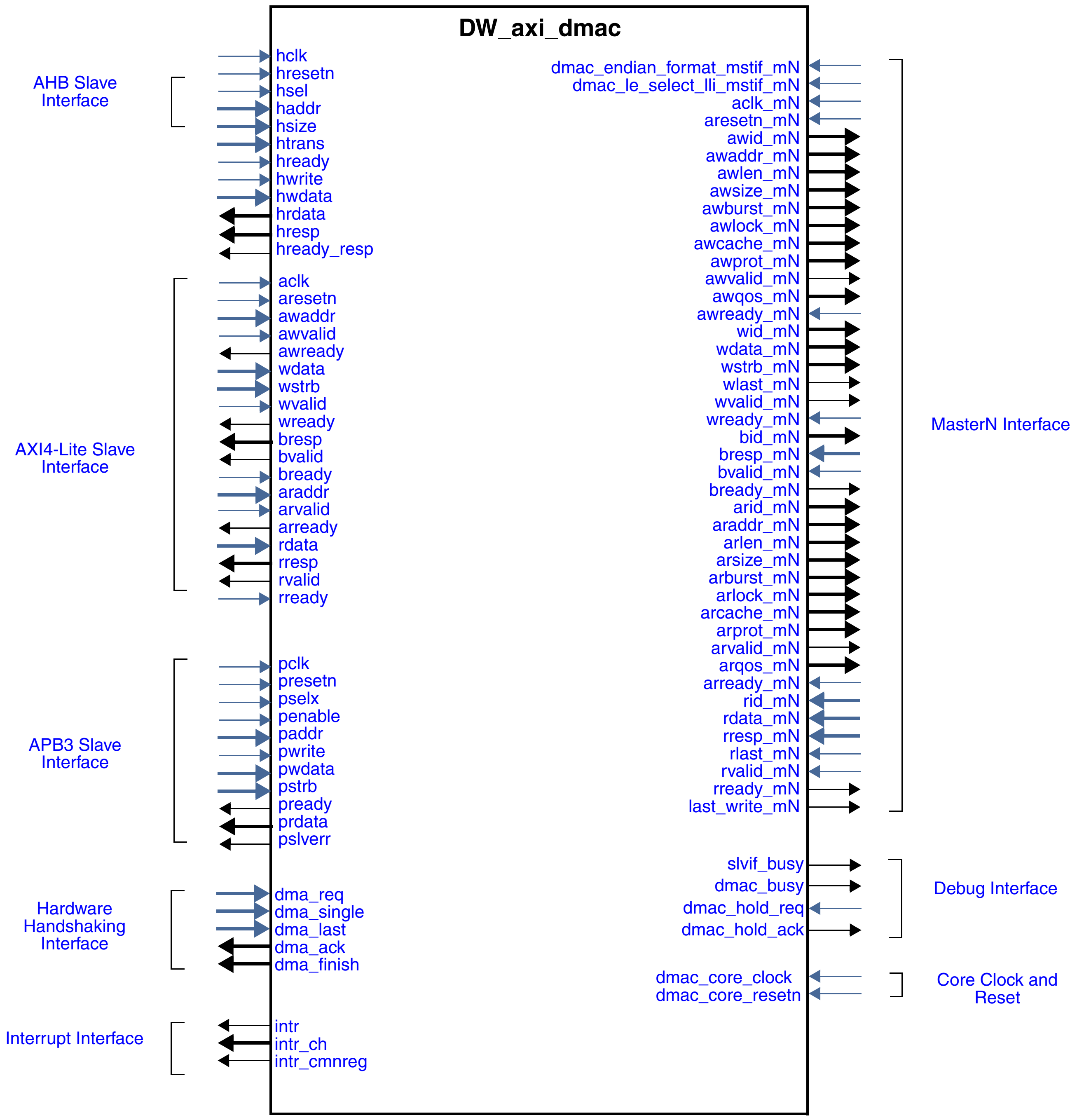
5 Signals

The following subsections describe how to interface to the DW_axi_dmac.

5.1 DW_axi_dmac Interface Diagram

Figure 5-1 shows the I/O signals for DW_axi_dmac. For signal descriptions, see Table 5-1 on page 101.

Figure 5-1 DW_axi_dmac Interface Diagram



5.2 DW_axi_dmac Signal Descriptions

Table 5-1 identifies the signals that are associated with the DW_axi_dmac.

Note The Description column in Table 5-1 provides detailed information about each signal.

- In the **Registered** field, a "Yes" indicates whether an I/O signal is directly connected to an internal register and nothing else. An I/O signal is also considered to be registered if the signal is connected to one or more inverters or buffers between the I/O port and internal register, but not connected to any logic that involves another signal.
- The **Input/Output Delay** field provides the percentage of the clock cycle assumed to be used by logic outside this design. The given value is used to automatically define the default synthesis constraints for input/output delay. You can override these default values in the Specify Port Constraints activity in coreConsultant or coreAssembler.

Table 5-1 DW_axi_dmac Signal Description

Name	Width	I/O	Description
Slave Interface **(NOTE:** Signals for the slave interface are included on the I/O depending on the value of the DMAX_SLVIF_MODE configuration parameter. The signals actually present on the I/O are for either the AHB, AXI4-Lite, or APB3 slave interface. Signals for all three slave interfaces are not present at the same time.)			
AHB Slave Interface (**NOTE:** AHB Slave Interface supports only SINGLE transfer (hburst = 3'b000), hence hburst signal is not available on the I/O.)			
hclk	1 bit	In	AHB slave interface clock. **Registered:** N/A **Synchronous to:** N/A **Dependencies:** DMAX_SLVIF_MODE = 0 AND DMAX_SLVIF_CLOCK_MODE = 1
hresetn	1 bit	In	AHB slave interface reset. **Active State:** Low **Registered:** N/A **Synchronous to:** Asynchronous assertion, synchronous de-assertion. The reset must be synchronously de-asserted after the rising edge of hclk. DW_axi_dmac does not contain logic to perform this synchronization, so it must be provided externally. **Dependencies:** DMAX_SLVIF_MODE = 0 AND DMAX_SLVIF_CLOCK_MODE = 1
hsel	1 bit	In	Slave select. Asserted when the current transfer on the AHB bus is intended for the DW_axi_dmac. **Active State:** High **Registered:** No **Synchronous to:** hclk **Dependencies:** DMAX_SLVIF_MODE = 0

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
haddr	32 bits	In	Address. **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hsize	3 bits	In	Transfer size. **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
htrans	2 bits	In	Transfer type. **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hready	1 bit	In	Current transfer is complete. **Active State:** High **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hwrite	1 bit	In	Transfer direction. When high, this signal indicates a write transfer. When low, this signal indicates a read transfer. **Active State:** 1 for write, 0 for read **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hwdata	[ds-1:0]	In	Write data to slave. **Width:** ds = (32/64) for AHB and depends on the parameter DMAX_S_DATA_WIDTH. **Registered:** No **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hrdata	[ds-1:0]	Out	Read data from slave. **Width:** ds = (32/64) for AHB and depends on the parameter DMAX_S_DATA_WIDTH. **Registered:** Yes **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
hresp	2 bits	Out	Response type from slave. When the DW_axi_dmac is configured for the 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. **Registered:** Yes **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
hready_resp	1 bit	Out	Transfer complete. **Active State:** High **Registered:** Yes **Synchronous to:** hclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 0
AXI4-Lite Slave Interface			
aclk	1 bit	In	AXI4-Lite interface clock. **Active State:** High, all AXI4-Lite signals are sampled on the rising edge of the clock. **Registered:** No **Synchronous to:** N/A **Dependencies:** DMAX_SLVIF_MODE = 1 AND DMAX_SLVIF_CLOCK_MODE = 1
aresetn	1 bit	In	AXI4-Lite interface reset. Active low input that asynchronously resets the AXI4-Lite interface to its default state. **Active State:** Low **Registered:** N/A **Synchronous to:** Asynchronous assertion, synchronous de-assertion. The reset must be synchronously de-asserted after the rising edge of aclk. DW_axi_dmac does not contain logic to perform this synchronization, so it must be provided externally. **Dependencies:** DMAX_SLVIF_MODE = 1 AND DMAX_SLVIF_CLOCK_MODE = 1
awaddr	32 bits	In	AXI4-Lite interface write address. Specifies the address of the DW_axi_dmac register to be written. **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
awvalid	1 bit	In	AXI4-Lite interface write address valid. Indicates the availability of a valid write address. **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
awready	1 bit	Out	AXI4-Lite interface write address ready. Indicates the DW_axi_dmac readiness to accept the write address. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
wdata	[ds-1:0]	In	AXI4-Lite interface write data. **Width:** ds = 32/64 for AXI4-Lite and depends on the parameter DMAX_S_DATA_WIDTH. **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
wstrb	See Desc.	In	AXI4-Lite Interface write data strobe. Indicates which byte lanes to update in register. There is one data strobe for each 8 bits of the write data. **Width:** DMAX_S_DATA_WIDTH/8 bits **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
wvalid	1 bit	In	AXI4-Lite interface write data valid. Indicates the availability of valid write data and strobes. **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
wready	1 bit	Out	AXI4-Lite interface write data ready. Indicates the DW_axi_dmac readiness to accept the write data and strobes. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
bresp	2 bits	Out	AXI4-Lite interface write response. Indicates the status of the write transaction. **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
bvalid	1 bit	Out	AXI4-Lite interface write response valid. Indicates the availability of a valid write response. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
bready	1 bit	In	AXI4-Lite interface write response ready. Indicates the AXI4-Lite master readiness to accept the write response from DW_axi_dmac. **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
araddr	32 bits	In	AXI4-Lite interface read address. Specifies the address of the DW_axi_dmac register to be read. **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
arvalid	1 bit	In	AXI4-Lite interface read address valid. Indicates the availability of a valid read address. **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
arready	1 bit	Out	AXI4-Lite interface read address ready. Indicates the DW_axi_dmac readiness to accept the read address. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
rdata	[ds-1:0]	In	AXI4-Lite interface read data. **Width:** ds = 32/64 for AXI4-Lite and depends on the parameter DMAX_S_DATA_WIDTH. **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
rresp	2 bits	Out	AXI4-Lite interface read response. Indicates the status of the read transaction. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
rvalid	1 bit	Out	AXI4-Lite interface read valid. Indicates the availability of valid read data and read response. **Active State:** High **Registered:** Yes **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
rready	1 bit	In	AXI4-Lite interface read data ready. Indicates the AXI4-Lite master readiness to accept the read data from DW_axi_dmac. **Active State:** High **Registered:** No **Synchronous to:** aclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 1
APB3 Slave Interface			
pclk	1 bit	In	APB3 interface clock **Active State:** High, all APB3 signals are sampled on the rising edge of the clock **Registered:** No **Synchronous to:** N/A **Dependencies:** DMAX_SLVIF_MODE = 2 AND DMAX_SLVIF_CLOCK_MODE = 1
presetn	1 bit	In	APB3 interface reset. active low input that asynchronously resets the APB3 interface to its default state. **Active State:** Low **Registered:** N/A **Synchronous to:** Asynchronous assertion, synchronous de-assertion. The reset must be synchronously de-asserted after the rising edge of pclk. DW_axi_dmac does not contain logic to perform this synchronization, so it must be provided externally. **Dependencies:** DMAX_SLVIF_MODE = 2 AND DMAX_SLVIF_CLOCK_MODE = 1

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
pselx	1 bit	In	APB3 interface select signal. Selects the DW_axi_dmac APB slave interface. Must be active for requests to be accepted. **Active State:** High **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
penable	1 bit	In	APB3 interface enable signal. **Active State:** High **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
paddr	32 bits	In	APB3 interface address. **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
pwrite	1 bit	In	APB3 interface write select signal. When active, the request is a write request. **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
pwdata	32 bits	In	APB3 interface write data. **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
pstrb	4 bits	In	APB3 interface write data strobe. Indicates which byte lanes to update in register. There is one data strobe for each 8 bits of the write data. Supported only on AMBA4 APB devices. **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2 This can be removed if APB4 support is not needed.
pready	1 bit	Out	APB3 interface ready signal. Indicates whether a request cycle was accepted. Supported only on AMBA3 APB devices. **Active State:** High **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
prdata	32 bits	Out	APB3 interface read data from DW_axi_dmac. **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
pslverr	1 bit	Out	APB3 interface flag for the slave error response from DW_axi_dmac. Supported only on AMBA3 APB devices. **Active State:** High **Registered:** No **Synchronous to:** pclk/dmac_core_clock **Dependencies:** DMAX_SLVIF_MODE = 2
MasterN Interface (N = 1, 2) **(NOTE**: Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF)			
dmac_endian_format_mstif_m*N*	1 bit	In	Master*N* interface endian format selection pins. Following are the endian formats supported. ■ 0 = Little Endian ■ 1 = Big Endian BE-8 **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This input is present only if DMAX_STATIC_ENDIAN_SELECT_MSTIF = 0.
dmac_le_select_lli_mstif_m*N*	1 bit	In	Master*N* interface endian format selection pin for LLI access (LLI fetch and LLI status write-back). The following are the endian formats supported: ■ 0 = Endian scheme used for LLI access is same as that used for data access on M*N* interface ■ 1 = Little endian method is used for LLI access irrespective of the endian scheme used for data access on M*N* interface. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This input is present only if DMAX_LLI_ENDIAN_SELECTION_PIN_EN = 1.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
aclk_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface clock **Active State:** High, all AXI3/AXI4 signals are sampled on the rising edge of the clock **Registered:** N/A **Synchronous to:** N/A **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF. aclk_m*N* is included on the I/O only if DMAX_MSTIF*N*_CLOCK_MODE = 1.
aresetn_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface reset. Active low input that asynchronously resets the AXI3/AXI4 Master*N* interface to its default state. **Active State:** Low **Registered:** N/A **Synchronous to:** Asynchronous assertion, synchronous de-assertion. The reset must be synchronously de-asserted after the rising edge of aclk_m*N*/dmac_core_clock. DW_axi_dmac does not contain logic to perform this synchronization, so it must be provided externally. **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF. aresetn_m*N* is included on the I/O only if DMAX_MSTIFN_CLOCK_MODE = 1.
awid_m*N*	[IDW-1:0]	Out	AXI3/AXI4 Master*N* interface write address ID. Identification tag for the write address group of signals. The upper 4 bits of awid_m*N* is derived from the channel number of the channel that is currently accessing the master interface (channel 1 = 4'b0000, channel 8 = 4'b0111, and so on). The lower bits are the same as the value programmed in the CHx_AXI_IDReg.AXI_Write_ID_Suffix field. **Width:** IDW = DMAX_M_ID_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
awaddr_m*N*	[aws-1:0]	Out	AXI3/AXI4 Master*N* interface write address. Specifies the address of the AXI write burst transaction. **Width:** aws = 32 to 64 bits, depending on the DMAX_M_ADDR_WIDTH parameter **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awlen_m*N*	[awls-1:0]	Out	AXI3/AXI4 Master*N* interface write burst length. **Width:** awl*s* = DMAX_M_BURSTLEN_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awsize_m*N*	3 bits	Out	AXI3/AXI4 Master*N* interface write burst size. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awburst_m*N*	2 bits	Out	AXI3/AXI4 Master*N* interface write burst type. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awlock_m*N*	2 bits	Out	AXI3/AXI4 Master*N* interface write lock type. **NOTE:** ■ awlock*N*[0] = 0 as DW_axi_dmac doesn't initiate exclusive transactions. ■ awlock*N*[1] = 0 as DW_axi_dmac doesn't support AXI bus locking. ■ awlock*N*[1] is not present in AXI4 mode. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** awlock*N*[1] is present only if DMAX_MSTIF_MODE = 0. Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
awcache_m*N*	4 bits	Out	AXI3/AXI4 Master*N* interface write cache type.CHx_CTL.AWCACHE field determines the value of this signal. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awprot_m*N*	3 bits	Out	AXI3/AXI4 Master*N* interface write protection type.CHx_CTL.AWPROT field determines the value of this signal. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awvalid_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface write address valid. Indicates the availability of valid write address and associated control signals. **Registered:** Yes **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awqos_m*N*	4 bits	Out	AXI4 Master*N* interface Quality Of Service signal for write channel. The value of awqos_m*N* is the same as the value programmed in the CHx_AXI_QoSReg.AXI_AWQOS field of the channel that is currently accessing the master interface. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This signal is present only if DMAX_MSTIF_MODE = 1. Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
awready_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface write address ready. Indicates the AXI slave readiness to accept write address and associated control signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
wid_m*N*	[IDW-1:0]	Out	AXI3 MasterN interface write data ID. Identification tag for the write data group of signals. The upper 4 bits of wid_m*N* is derived from the channel number of the channel that is currently accessing the master interface (channel 1 = 4'b0000, channel 8 = 4'b0111, and so on). The lower bits are the same as the value programmed in the CHx_AXI_IDReg.AXI_Write_ID_Suffix field. DW_axi_dmac does not support write data interleaving. Hence, the wid_m*N* for a particular write transaction is same as the awid_m*N* of the corresponding write request. **Width:** IDW = DMAX_M_ID_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This signal is present only if DMAX_MSTIF_MODE = 0. Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
wdata_m*N*	[wds-1:0]	Out	AXI3/AXI4 Master*N* interface write data. **Width:** wd*s* = 32/64/128/256/512 bits and depends on the parameter DMAX_M_DATA_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
wstrb_m*N*	[wss-1:0]	Out	AXI3/AXI4 Master*N* interface write data strobe. **Width:** wss = DMAX_M_DATA_WIDTH /8 **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
wlast_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface write last. Indicates the last transfer in a write burst. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
wvalid_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface write data valid. Indicates the availability of valid write data and associated control signals. **Registered:** Yes **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
wready_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface write data ready. Indicates the AXI slave readiness to accept write data and associated control signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
bid_m*N*	4 bits	Out	AXI3/AXI4 Master*N* interface write response ID. Identification tag for the write response group of signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
bresp_m*N*	2 bits	In	AXI3/AXI4 Master*N* interface write response. AXI slave indicates the status of write transaction. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
bvalid_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface write response valid. AXI slave indicates the availability of a valid write response. **Registered:** Yes **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
bready_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface write response ready. Indicates the DW_axi_dmac readiness to accept write response. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
arid_m*N*	[IDW-1:0]	Out	AXI3/AXI4 Master*N* interface read address ID. Identification tag for the read address group of signals. The upper 4 bits of arid_m*N* is derived from the channel number of the channel that is currently accessing the master interface. This varies for LLI fetch and source data transfer. ■ For source data transfer, arid_m*N* for channel 1 = 4'b0000, arid_m*N* for channel 8 = 4'b0111, and so on. ■ For LLI fetch access, arid_m*N* for channel 1 = 4'b1000, arid_m*N* for channel 8 = 4'b1111, and so on. Lower bits are the same as the value programmed in the CHx_AXI_IDReg.AXI_Read_ID_Suffix field. **Width:** IDW = DMAX_M_ID_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
araddr_m*N*	[ars-1:0]	Out	AXI3/AXI4 Master*N* interface read address. Specifies the address of the AXI read burst transaction. **Width:** ar*s* = 32 to 64 bits and depends on the parameter DMAX_M_ADDR_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arlen_m*N*	[arls-1:0]	Out	AXI3/AXI4 Master*N* interface read burst length. **Width:** arl*s* = DMAX_M_BURSTLEN_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arsize_m*N*	3 bits	Out	AXI3/AXI4 Master*N* interface read burst size. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
arburst_m*N*	2 bits	Out	AXI3/AXI4 Master*N* interface read burst type. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arlock_m*N*	2 bits	Out	AXI3/AXI4 Master*N* interface read lock type. **NOTE:** ■ arlock_*mN*[0] = 0 as DW_axi_dmac does not initiate exclusive transactions. ■ arlock_*mN*[1] = 0 as DW_axi_dmac does not support AXI bus locking. ■ arlock_*mN*[1] = 0 in AXI4 mode. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** arlock_m*N*[1] is present only if DMAX_MSTIF_MODE = 0. Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arcache_m*N*	4 bits	Out	AXI3/AXI4 Master*N* interface write cache type. The CHx_CTL.ARCACHE field determines the value of this signal. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arprot_m*N*	3 bits	Out	AXI3/AXI4 Master*N* interface read protection type.The CHx_CTL.ARPROT field determines the value of this signal. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arvalid_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface read address valid. Indicates the availability of a valid read address and associated control signals. **Registered:** Yes **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
arqos_m*N*	4 bits	Out	AXI4 Master*N* interface Quality of Service signal for read channel. The value of arqos_m*N* is the same as the value programmed in the CHx_AXI_QoSReg.AXI_ARQOS field of the channel that is currently accessing the master interface. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This signal is present only if DMAX_MSTIF_MODE = 1. Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
arready_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface read address ready. Indicates the AXI slave readiness to accept read address and associated control signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
rid_m*N*	4 bits	In	AXI3/AXI4 Master*N* interface read data ID. Identification tag for the read data group of signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
rdata_m*N*	[rds-1:0]	In	AXI3/AXI4 Master*N* interface read data. **Width:** rd*s* = 32/64/128/256/512 bits, depends on the parameter DMAX_M_DATA_WIDTH **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
rresp_m*N*	2 bits	In	AXI3/AXI4 Master*N* Interface read response. AXI slave indicates the status of read transaction. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
rlast_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface read last. Indicates the last transfer in a read burst. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
rvalid_m*N*	1 bit	In	AXI3/AXI4 Master*N* interface read data valid. Indicates the availability of valid read data and associated control signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
rready_m*N*	1 bit	Out	AXI3/AXI4 Master*N* interface read data ready. Indicates the DW_axi_dmac readiness to accept read address and associated control signals. **Registered:** No **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** Signals for master interface 2 are included on the I/O depending on the value of configuration parameter DMAX_NUM_MASTER_IF.
last_write_m*N*	1 bit	Out	Last write data of block transfer. **Active State:** High **Registered:** Yes **Synchronous to:** aclk_m*N*/dmac_core_clock **Dependencies:** This signal is present only if DMAX_ENABLE_LAST_WRITE = 1.
Hardware Handshaking Interface (**NOTE:** Signals for hardware handshaking interface are included on the I/O depending on the value of configuration parameter DMAX_NUM_HS_IF. If DMAX_NUM_HS_IF = 0, there are no hardware handshaking interface signals on the I/O.) In the hardware handshaking signals described below, h = DMAX_NUM_HS_IF			
dma_req	[*h*-1:0]	In	DMA transaction request from peripheral. **Width:** *h* = number (1-16) of DW_axi_dmac hardware handshaking interfaces selected (parameter DMAX_NUM_HS_IF). **Registered:** Yes **Synchronous to:** dmac_core_clock **Dependencies:** If DMAX_NUM_HS_IF = 0, then this hardware handshaking interface does not exist.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
dma_single	[*h*-1:0]	In	Single transfer request/status. **Width:** *h* = number (1-16) of DW_axi_dmac hardware handshaking interfaces selected (parameter DMAX_NUM_HS_IF). **Registered:** Yes **Synchronous to:** dmac_core_clock **Dependencies:** The function of this signal depends on whether the peripheral assigned to this handshaking interface is the flow controller or not. If DMAX_NUM_HS_IF = 0, then the hardware handshaking interface does not exist and this signal does not appear on the I/O.
dma_last	[*h*-1:0]	In	Last transaction in block indicator. **Width:** *h* = number (1-16) of DW_axi_dmac hardware handshaking interfaces selected (parameter DMAX_NUM_HS_IF). **Registered:** Yes **Synchronous to:** dmac_core_clock **Dependencies:** The function of this signal depends on whether the peripheral assigned to this handshaking interface is the flow controller or not. This signal is useful only when the peripheral assigned to this handshaking interface is the flow controller. If DMAX_NUM_HS_IF = 0, then the hardware handshaking interface does not exist and this signal does not appear on the I/O.
dma_ack	[*h*-1:0]	Out	Transaction complete acknowledge signal. **Width:** *h* = number (1-16) of *DW_axi_dmac* hardware handshaking interfaces selected (parameter DMAX_NUM_HS_IF). **Registered:** No **Synchronous to:** dmac_core_clock **Dependencies:** If DMAX_NUM_HS_IF = 0, then this hardware handshaking interface does not exist and this signal does not appear on the I/O.
dma_finish	[*h*-1:0]	Out	DMA block complete signal. **Width:** *h* = number (1-16) of DW_axi_dmac hardware handshaking interfaces selected (parameter DMAX_NUM_HS_IF). **Registered:** No **Synchronous to:** dmac_core_clock **Dependencies:** If DMAX_NUM_HS_IF = 0, then this hardware handshaking interface does not exist and this signal does not appear on the I/O.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
Interrupt Interface			
intr	1 bit	Out	Logical OR of all individual interrupts. **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock. If the option to synchronize the interrupt to slave interface clock is selected (DMAX_INTR_SYNC2SLVCLK = 1), then it is synchronous to hclk/aclk/pclk. **Dependencies:** DMAX_INTR_IO_TYPE = 0 AND DMAC_CfgReg.INT_EN = 1
intr_ch	[*nc*-1:0]		Logical OR of all individual interrupts of each enabled channels. **Width:** *nc* = number (1-8) of DW_axi_dmac channels selected (parameter DMAX_NUM_CHANNELS). **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock. If the option to synchronize the interrupt to slave interface clock is selected (DMAX_INTR_SYNC2SLVCLK = 1), then it is synchronous to hclk/aclk/pclk **Dependencies:** DMAX_INTR_IO_TYPE > 0 AND DMAC_CfgReg.INT_EN = 1
intr_cmnreg	1 bit		Logical OR of all common register interrupts. **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock. If the option to synchronize the interrupt to slave interface clock is selected (DMAX_INTR_SYNC2SLVCLK = 1), then it is synchronous to hclk/aclk/pclk **Dependencies:** DMAX_INTR_IO_TYPE > 0 AND DMAC_CfgReg.INT_EN = 1
Debug Interface			
slvif_busy	1 bit	Out	Indicates whether the slave interface is busy. **Active State:** High **Registered:** No **Synchronous to:** hclk/aclk/pclk/dmac_core_clock based on DMAX_SLVIF_MODE AND DMAX_SLVIF_CLOCK_MODE **Dependencies:** This output is present only if DMAX_SLVIF_STATUS_OP _EN = 1.

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
dmac_busy	1 bit	Out	Indicates whether DW_axi_dmac core is busy. **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock **Dependencies:** This output is present only if DMAX_CORE_STATUS_OP _EN = 1.
dmac_hold_req	1 bit	In	Request to put DW_axi_dmac in hold (freeze) mode. Asserting this request puts the entire DW_axi_dmac in freeze mode without violating the AXI protocol. To exit the hold mode, this signal can be de-asserted after DW_axi_dmac asserts dmac_hold_ack. **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock **Dependencies:** This output is present only if DMAX_HOLD_IO_EN = 1
dmac_hold_ack	1 bit	Out	Hold request acknowledgement. DW_axi_dmac asserts this signal after entering the hold (freeze) mode. This signal is de-asserted when dmac_hold_req is de-asserted. **NOTE:** It is not allowed to de-assert dmac_hold_req before asserting dmac_hold_ack. **Active State:** High **Registered:** No **Synchronous to:** dmac_core_clock **Dependencies:** This output is present only if DMAX_HOLD_IO_EN = 1.
debug_*	[ ]	Out	Debug ports included on the I/O for verification purposes. These signals should be left disconnected. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Dependencies:** DMAX_DEBUG_PORTS_EN =1
Core Clock and Reset			
dmac_core_clock	1 bit	In	DW_axi_dmac core clock**.** **Active State:** High, all AXI3/AXI4 signals are sampled on the rising edge of the clock **Registered:** N/A **Synchronous to:** N/A **Dependencies:** None

Table 5-1 DW_axi_dmac Signal Description (Continued)

Name	Width	I/O	Description
dmac_core_resetn	1 bit	In	DW_axi_dmac core reset. Active low input that asynchronously resets the DW_axi_dmac. **Active State:** Low **Registered:** N/A **Synchronous to:** Asynchronous assertion, synchronous de-assertion. The reset must be synchronously de-asserted after the rising edge of dmac_core_clock. DW_axi_dmac does not contain logic to perform this synchronization, so it must be provided externally. **Dependencies:** None

6 Registers

This chapter includes information on the DW_axi_dmac programmable software registers.

All registers in DW_axi_dmac are 64 bits wide. DW_axi_dmac supports a 32/64-bit data bus width for the slave interface, depending on the type of slave interface used. The transfer size (width) used for register access must be the same as the data bus width.

- For a 64-bit wide access, the address must be aligned to a double word boundary (where the lower 3 bits are 0s).
- For a 32-bit wide access, the address must be aligned to a word boundary (where the lower 2 bits are 0s).
- For a 32-bit wide access, if the address bit 2 is '0' (for example, 12'h100, 12'h108, and so on), the lower 32 bits of the corresponding 64-bit register are accessed, as shown as follows:

  0x100: DMAC_CH1_SAR [31:0]

  0x108: DMAC_CH1_DAR [31:0]

  If the address bit 2 is '1' (for example, 12'h104, 12'h10C, and so on), the upper 32 bits of the corresponding 64-bit register are accessed, as shown as follows:

  0x104: DMAC_CH1_SAR [63:32]

  0x10C: DMAC_CH1_DAR [63:32]

6.1 Register Memory Map

Table 6-1 shows the high level register memory map for the DW_axi_dmac. The complete memory map is shown in Table 6-2.

Table 6-1 High-level Register Memory Map of DW_axi_dmac

Address Range	Register Address Space Name
32’h0000_0100 to 32’h0000_01FF	Channel 1 Register Space
32’h0000_0200 to 32’h0000_02FF	Channel 2 Register Space
32’h0000_0300 to 32’h0000_03FF	Channel 3 Register Space
32’h0000_0400 to 32’h0000_04FF	Channel 4 Register Space
32’h0000_0500 to 32’h0000_05FF	Channel 5 Register Space
32’h0000_0600 to 32’h0000_06FF	Channel 6 Register Space
32’h0000_0700 to 32’h0000_07FF	Channel 7 Register Space
32’h0000_0800 to 32’h0000_08FF	Channel 8 Register Space
32’h0000_0900 to 32’h0000_0FFF	Undefined Register Space

Table 6-2 Complete Memory Map of DW_axi_dmac

Name	Address Offset	Width	R/W	Description
Common Registers				
DMAC_IDReg	0x000	64	R	DMAC ID Register Reset Value: DMAX_ID_NUM Reg Exist: Yes
DMAC_CompVerReg	0x008	64	R	DMAC Component Version Register Reset Value: DMAC_COMP_VER Reg Exist: Yes
DMAC_CfgReg	0x010	64	R/W	DMAC Configuration Register Reset Value: 0 Reg Exist: Yes
DMAC_ChEnReg	0x018	64	R/W	DMAC Channel Enable Register Reset Value: 0 Reg Exist: Yes
DMAC_IntStatusReg	0x030	64	R	DMAC Interrupt Status Register Reset Value: 0 Reg Exist: Yes

Table 6-2 Complete Memory Map of DW_axi_dmac (Continued)

Name	Address Offset	Width	R/W	Description
DMAC_CommonReg_IntClearReg	0x038	64	W	DMAC Interrupt Clear Register Reset Value: 0 Reg Exist: Yes
DMAC_CommonReg_IntStatus_EnableReg	0x040	64	R	DW_axi_dmac Common Register Interrupt Status Enable Register Reset Value: 64'hFFFFFFFF Reg Exist: Yes
DMAC_CommonReg_IntSignal_EnableReg	0x048	64	R/W	DMAC Common Register Interrupt Signal Enable Register Reset Value: 64'hFFFFFFFF Reg Exist: Yes
DMAC_CommonReg_IntStatusReg	0x050	64	R	DMAC Common Register Interrupt Status Register Reset Value: 0 Reg Exist: Yes
DMAC_ResetReg	0x058	64	R	DMAC Reset Register1 Reset Value: 0 Reg Exist: Yes
DW_axi_dmac Channel Registers, where *x* = 1 to 8 (based on DMAX_NUM_CHANNELS registers)				
CHx_SAR for *x* = 1 to 8 CH1 _SAR CH2_SAR CH3_SAR CH4_SAR CH5_SAR CH6_SAR CH7_SAR CH8_SAR	 0x100 0x200 0x300 0x400 0x500 0x600 0x700 0x800	64	R/W	Channel*x* Source Address Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*
CHx_DAR for *x* = 1 to 8 CH1_DAR CH2_DAR CH3_DAR CH4_DAR CH5_DAR CH6_DAR CH7_DAR CH8_DAR	 0x108 0x208 0x308 0x408 0x508 0x608 0x708 0x808	64	R/W	Channel*x* Destination Address Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*

Table 6-2 **Complete Memory Map of DW_axi_dmac (Continued)**

Name	Address Offset	Width	R/W	Description
CHx_BLOCK_TS		64	R/W	Channel*x* Block Transfer Size Register
for *x* = 1 to 8				Reset Value: 0
CH1_BLOCK_TS	0x110			Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH2_BLOCK_TS	0x210			
CH3_BLOCK_TS	0x310			
CH4_BLOCK_TS	0x410			
CH5_BLOCK_TS	0x510			
CH6_BLOCK_TS	0x610			
CH7_BLOCK_TS	0x710			
CH8_BLOCK_TS	0x810			
CHx_CTL		64	R/W	Channel*x* Control Register
for *x*= 1 to 8				Reset Value: 64'h00001200
CH1_CTL	0x118			Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH2_CTL	0x218			
CH3_CTL	0x318			
CH4_CTL	0x418			
CH5_CTL	0x518			
CH6_CTL	0x618			
CH7_CTL	0x718			
CH8_CTL	0x818			
CHx_CFG		64	R/W	Channel*x* Configuration Register
for *x* = 1 to 8				Reset Value: See the **Reset Value** column in Table 6-19.
CH1_CFG	0x120			Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH2_CFG	0x220			
CH3_CFG	0x320			
CH4_CFG	0x420			
CH5_CFG	0x520			
CH6_CFG	0x620			
CH7_CFG	0x720			
CH8_CFG	0x820			
CHx_LLP		64	R/W	Channel*x* Linked List Pointer Register
for *x*= 1 to 8				Reset Value: 0
CH1_LLP	0x128			Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH2_LLP	0x228			
CH3_LLP	0x328			
CH4_LLP	0x428			
CH5_LLP	0x528			
CH6_LLP	0x628			
CH7_LLP	0x728			
CH8_LLP	0x828			

Table 6-2 Complete Memory Map of DW_axi_dmac (Continued)

Name	Address Offset	Width	R/W	Description
CHx_STATUSREG for *x*= 1 to 8 CH1_StatusReg CH2_StatusReg CH3_StatusReg CH4_StatusReg CH5_StatusReg CH6_StatusReg CH7_StatusReg CH8_StatusReg	 0x130 0x230 0x330 0x430 0x530 0x630 0x730 0x830	64	R	Channel*x* Status Register Reset Value:0 Reg Exist: DMAX_NUM_CHANNELS >= *x*
CHx_SWHSSRCREG for *x*= 1 to 8 CH1_SWHSSrcReg CH2_SWHSSrcReg CH3_SWHSSrcReg CH4_SWHSSrcReg CH5_SWHSSrcReg CH6_SWHSSrcReg CH7_SWHSSrcReg CH8_SWHSSrcReg	 0x138 0x238 0x338 0x438 0x538 0x638 0x738 0x838	64	R/W	Channel*x* Software Handshake Source Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*
CHx_SWHSDSTREG for *x*= 1 to 8 CH1_SWHSDstReg CH2_SWHSDstReg CH3_SWHSDstReg CH4_SWHSDstReg CH5_SWHSDstReg CH6_SWHSDstReg CH7_SWHSDstReg CH8_SWHSDstReg	 0x140 0x240 0x340 0x440 0x540 0x640 0x740 0x840	64	R/W	Channel*x* Software Handshake Destination Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*
CHx_BLK_TFR_RESUMEREQREG for *x*= 1 to 8 CH1_BLK_TFR_ResumeReqReg CH2_BLK_TFR_ResumeReqReg CH3_BLK_TFR_ResumeReqReg CH4_BLK_TFR_ResumeReqReg CH5_BLK_TFR_ResumeReqReg CH6_BLK_TFR_ResumeReqReg CH7_BLK_TFR_ResumeReqReg CH8_BLK_TFR_ResumeReqReg	 0x148 0x248 0x348 0x448 0x548 0x648 0x748 0x848	64	W	Channel*x* Block Transfer Resume Request Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_CH*x*_MULTI_BLK_EN =1

Table 6-2 Complete Memory Map of DW_axi_dmac (Continued)

Name	Address Offset	Width	R/W	Description
CHx_AXI_IDREG		64	R/W	Channel*x* AXI ID Register
for *x* = 1 to 8				Reset Value: 0
CH1_AXI_IDReg	0x150			Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH2_AXI_IDReg	0x250			
CH3_AXI_IDReg	0x350			
CH4_AXI_IDReg	0x450			
CH5_AXI_IDReg	0x550			
CH6_AXI_IDReg	0x650			
CH7_AXI_IDReg	0x750			
CH8_AXI_IDReg	0x850			
CHx_AXI_QOSREG		64	R/W	Channel*x* AXI QOS Register
for *x* = 1 to 8				Reset Value: 0
CH1_AXI_QoSReg	0x158			Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_MSTIF_MODE = 1
CH2_AXI_QoSReg	0x258			
CH3_AXI_QoSReg	0x358			
CH4_AXI_QoSReg	0x458			
CH5_AXI_QoSReg	0x558			
CH6_AXI_QoSReg	0x658			
CH7_AXI_QoSReg	0x758			
CH8_AXI_QoSReg	0x858			
CHx_SSTAT		64	R	Channel*x* Source Status Register
for *x*= 1 to 8				Reset Value: 0
CH1_SSTAT	0x160			Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_CHx_SRC_STAT_EN = 1
CH2_SSTAT	0x260			
CH3_SSTAT	0x360			
CH4_SSTAT	0x460			
CH5_SSTAT	0x560			
CH6_SSTAT	0x660			
CH7_SSTAT	0x760			
CH8_SSTAT	0x860			
CHx_DSTAT		64	R	Channel*x* Destination Status Register
for *x* = 1 to 8				Reset Value: 0
CH1_DSTAT	0x168			Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_CHx_DST_STAT_EN = 1
CH2_DSTAT	0x268			
CH3_DSTAT	0x368			
CH4_DSTAT	0x468			
CH5_DSTAT	0x568			
CH6_DSTAT	0x668			
CH7_DSTAT	0x768			
CH8_DSTAT	0x868			

Table 6-2 Complete Memory Map of DW_axi_dmac (Continued)

Name	Address Offset	Width	R/W	Description
CHx_SSTATAR for *x* = 1 to 8		64	R/W	Channel*x* Source Status Fetch Address Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_CHx_SRC_STAT_EN = 1
CH1_SSTATAR	0x170			
CH2_SSTATAR	0x270			
CH3_SSTATAR	0x370			
CH4_SSTATAR	0x470			
CH5_SSTATAR	0x570			
CH6_SSTATAR	0x670			
CH7_SSTATAR	0x770			
CH8_SSTATAR	0x870			
CHx_DSTATAR for *x* = 1 to 8		64	R/W	Channel*x* Destination Status Fetch Address Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x* AND DMAX_CHx_DST_STAT_EN = 1
CH1_DSTATAR	0x178			
CH2_DSTATAR	0x278			
CH3_DSTATAR	0x378			
CH4_DSTATAR	0x478			
CH5_DSTATAR	0x578			
CH6_DSTATAR	0x678			
CH7_DSTATAR	0x778			
CH8_DSTATAR	0x878			
CHx_INTSTATUS_ENABLEREG for *x* = 1 to 8		64	R/W	Channel*x* Interrupt Status Enable Register Reset Value: 64'hFFFFFFFF Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH1_INTSTATUS_ENABLEREG	0x180			
CH2_INTSTATUS_ENABLEREG	0x280			
CH3_INTSTATUS_ENABLEREG	0x380			
CH4_INTSTATUS_ENABLEREG	0x480			
CH5_INTSTATUS_ENABLEREG	0x580			
CH6_INTSTATUS_ENABLEREG	0x680			
CH7_INTSTATUS_ENABLEREG	0x780			
CH8_INTSTATUS_ENABLEREG	0x880			
CHx_INTSTATUSREG for *x* = 1 to 8		64	R/W	Channel*x* Interrupt Status Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*
CH1_INTSTATUS	0x188			
CH2_INTSTATUS	0x288			
CH3_INTSTATUS	0x388			
CH4_INTSTATUS	0x488			
CH5_INTSTATUS	0x588			
CH6_INTSTATUS	0x688			
CH7_INTSTATUS	0x788			
CH8_INTSTATUS	0x888			

Table 6-2 Complete Memory Map of DW_axi_dmac (Continued)

Name	Address Offset	Width	R/W	Description
CHx_INTSIGNAL_ENABLEREG for *x* = 1 to 8 CH1_INTSIGNAL_ENABLEREG CH2_INTSIGNAL_ENABLEREG CH3_INTSIGNAL_ENABLEREG CH4_INTSIGNAL_ENABLEREG CH5_INTSIGNAL_ENABLEREG CH6_INTSIGNAL_ENABLEREG CH7_INTSIGNAL_ENABLEREG CH8_INTSIGNAL_ENABLEREG	 0x190 0x290 0x390 0x490 0x590 0x690 0x790 0x890	64	R/W	Channel*x* Interrupt Signal Enable Register Reset Value: 64'hFFFFFFFF Reg Exist: DMAX_NUM_CHANNELS >= *x*
CHx_INTCLEARREG for *x* = 1 to 8 CH1_INTCLEARREG CH2_INTCLEARREG CH3_INTCLEARREG CH4_INTCLEARREG CH5_INTCLEARREG CH6_INTCLEARREG CH7_INTCLEARREG CH8_INTCLEARREG	 0x198 0x298 0x398 0x498 0x598 0x698 0x798 0x898	64	W	Channel*x* Interrupt Clear Register Reset Value: 0 Reg Exist: DMAX_NUM_CHANNELS >= *x*

6.2 Register and Field Descriptions

The following sections contain the field descriptions for the individual registers.

6.2.1 Common Registers

This section provides details on the DW_axi_dmac common registers.

6.2.1.1 DMAC_IDReg

This register contains a 32-bit value that is hardwired and read back by a read to the DW_axi_dmac ID register.

- **Name:** DW_axi_dmac ID Register
- **Size:** 64 bits
- **Address Offset:** 0x000
- **Read/Write Access:** Read

Table 6-3 DMAC_IDREG

Bits	Name	R/W	Reset Value	Description
63:0	DMAC_ID	R	DMAX_ID_NUM	DW_axi_dmac ID

6.2.1.2 DMAC_CompVerReg

This is the DW_axi_dmac Component Version register, which is a read-only register that specifies the version of the packaged component.

- **Name:** DW_axi_dmac ID Register
- **Size:** 64 bits
- **Address Offset:** 0x008
- **Read/Write Access:** Read

Table 6-4 DMAC_CompVerReg

Bits	Name	R/W	Reset Value	Description
63:32	Reserved and read as zero			
31:0	DMAC_COMP_VER	R	DMAC_COMP_VER	DMAC Component Version Number

6.2.1.3 DMAC_CfgReg

This register is used to enable the DW_axi_dmac, which must be done before any channel activity can begin. This register also contains global interrupt enable bit.

- **Name:** DW_axi_dmac Configuration Register
- **Size:** 64 bits
- **Address Offset:** 0x010
- **Read/Write Access:** Read/Write

Table 6-5 Fields for Register: DMAC_CfgReg

Bits	Name	R/W	Reset Value	Description
63:2	Reserved and read as zero			
1	INT_EN	R/W	0	Global Interrupt Enable bit 0: DW_axi_dmac interrupt bits are Disabled. 1: DW_axi_dmac interrupt bits are Enabled; individual interrupt bits need to be unmasked to enable each interrupt.
0	DMAC_EN	R/W	0	**DW_axi_dmac Enable bit**. 0: DW_axi_dmac Disabled 1: DW_axi_dmac Enabled Reset value: 0 **Note:** If DMAC_EN is cleared while any channel is still active, then this bit still returns 1 to indicate that there are channels still active until DW_axi_dmac hardware has terminated all activity on all channels, at which point this bit returns zero(0).

6.2.1.4 DMAC_ChEnReg

If software wants to set up a new channel, it can read this register to find out which channels are currently inactive and then enable an inactive channel with the required priority. This register also contains fields to suspend or abort a particular channel.

- **Name:** DW_axi_dmac Channel Enable Register
- **Size:** 64 bits
- **Offset:** 0x18
- **Read/Write Access:** Read/Write

Table 6-6 Fields for Register: DMAC_ChEnReg

Bits	Name	R/W	Reset Value	Description
63:(40+NC[a])	Reserved and read as zero			
(39+NC):40	CH_ABORT_WE	W	0	DW_axi_dmac Channel Abort Write Enable bit. Read back value of this register bit is always 0.
39:(32+NC)	Reserved and read as zero			
(31+NC):32	CH_ABORT	R/W	0	Channel Abort Request. Software sets this bit to 1 to request channel abort. If this bit is set to 1, DW_axi_dmac disables the channel immediately. Aborting the channel is not recommended and should be used only in situations where a particular channel hangs due to no response from the corresponding handshaking interface and software wants to disable the channel without resetting the entire DW_axi_dmac. It is recommended to try channel disabling first and then only opt for channel aborting. 0: No Channel Abort Request. 1: Request for Channel Abort. DW_axi_dmac clears this bit to 0 once the channel is aborted (when it sets CHx_Status.CH_ABORTED bit to 1).
31:(24+NC)	Reserved and read as zero			
(23+NC):24	CH_SUSP_WE	W	0	DW_axi_dmac Channel Suspend Write Enable bit. Read back value of this register bit is always 0.
23:(16+NC)	Reserved and read as zero			

Table 6-6 Fields for Register: DMAC_ChEnReg (Continued)

Bits	Name	R/W	Reset Value	Description
(15+NC):16	CH_SUSP	R/W	0	Channel Suspend Request. Software sets this bit to 1 to request channel suspend. If this bit is set to 1, DW_axi_dmac suspends all DMA data transfers from the source gracefully until this bit is cleared. There is no guarantee that the current DMA transaction will complete. This bit can also be used in conjunction with CHx_IntStatusReg.CH_SUSPENDED to cleanly disable the channel without losing any data. In this case, software first sets CH_SUSP bit to 1 and polls CHx_Status.CH_SUSPENDED until it is set to 1. Software can then clear CH_EN bit to 0 to disable the channel. 0: No Channel Suspend Request. 1: Request for Channel Suspend. Software can clear CH_SUSP bit to 0, after DW_axi_dmac sets CHx_IntStatusReg.CH_SUSPENDED bit to 1, to exit the channel suspend mode. **Note:** CH_SUSP is cleared when channel is disabled.
15:(8+NC)	Reserved and read as zero			
(7:NC):8	CH_EN_WE	W	0	DW_axi_dmac Channel Enable Write Enable bit. Read back value of this register bit is always '0'.
7:NC	Reserved and read as zero			
(NC-1):0	CH_EN	R/W	0	DW_axi_dmac Channel Enable bit 0: DW_axi_dmac Channel is disabled. 1: DW_axi_dmac Channel is enabled. The DMAC_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.

a. NC = DMAX_NUM_CHANNELS

All bits of this register are cleared to 0 when the DW_axi_dmac Global Enable bit (DMAC_CfgReg.DMAC_EN) is 0. When DMAC_CfgReg.DMAC_EN is 0, a write to the DMAC_ChEnReg register is ignored and a read always reads back 0.

The channel enable bit, DMAC_ChEnReg.CH_EN, is written only if the corresponding channel write enable bit, DMAC_ChEnReg.CH_EN_WE, is asserted on the same slave interface write transfer. For example, writing hex XXXX01X1 writes a 1 into DMAC_ChEnReg [0], while DMAC_ChEnReg [7:1] remains unchanged. Writing hex XXXX00XX leaves DMAC_ChEnReg [7:0] unchanged.

The channel suspend bit, DMAC_ChEnReg.CH_SUSP, is written only if the corresponding channel write enable bit, DMAC_ChEnReg.CH_SUSP_WE, is asserted on the same slave interface write transfer. For example, writing hex 01X1XXXX writes a 1 into DMAC_ChEnReg [16], while DMAC_ChEnReg [23:17] remains unchanged. Writing hex 00XXXXXX leaves DMAC_ChEnReg [23:16] unchanged. The channel abort bit, DMAC_ChEnReg.CH_ABORT, is written only if the corresponding channel write enable bit, DMAC_ChEnReg.CH_ABORT_WE, is asserted on the same slave interface write transfer.

6.2.1.5 DMAC_IntStatusReg

This register captures the combined channel interrupt for each channel and combined common register block interrupt.

- **Name:** DW_axi_dmac Interrupt Status Register
- **Size:** 64 bits
- **Offset:** 0x30
- **Read/Write Access:** Read

Table 6-7 Fields for Register: DMAC_IntStatusReg

Bits	Name	R/W	Reset Value	Description
63:17	Reserved and read as zero			
16	CommonReg_IntStat	R	0	Common Register Interrupt Status Bit.
15:8	Reserved and read as zero			
7	CH8_IntStat	R	0	Channel 8 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS = 8
6	CH7_IntStat	R	0	Channel 7 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS >= 7
5	CH6_IntStat	R	0	Channel 6 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS > = 6
4	CH5_IntStat	R	0	Channel 5 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS > = 5
3	CH4_IntStat	R	0	Channel 4 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS >= 4
2	CH3_IntStat	R	0	Channel 3 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS >= 3
1	CH2_IntStat	R	0	Channel 2 Interrupt Status Bit **Exists**: DMAX_NUM_CHANNELS >= 2
0	CH1_IntStat	R	0	Channel 1 Interrupt Status Bit

6.2.1.6 DMAC_CommonReg_IntClearReg

Writing 1 to specific field clears the corresponding field in the DMAC Common Register Interrupt Status Register (DMAC_CommonReg_IntStatusReg).

- **Name:** DW_axi_dmac Common Register Interrupt clear Register
- **Size:** 64 bits
- **Offset:** 0x38
- **Read/Write Access:** Write

Table 6-8 Fields for Register: DMAC_CommonReg_IntClearReg

Bits	Name	Memory Access	Reset Value	Description
63:9	Reserved and read as zero			
8	Clear_SLVIF_UndefinedReg_DEC_ERR_IntStat	W	0	Slave Interface Undefined Register Decode Error Interrupt Clear Bit Writing a 1 to this bit clears the corresponding channel interrupt status bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_UndefinedReg_DEC_ERR_IntStat).
7:4	Reserved and read as zero			
3	Clear_SLVIF_CommonReg_WrOnHold_ERR_IntStat	W	0	Slave Interface Common Register Write On Hold Error Interrupt clear Bit Writing a 1 to this bit clears the corresponding channel interrupt status bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_WrOnHold_ERR_IntStat).
2	Clear_SLVIF_CommonReg_RD2WO_ERR_IntStat	W	0	Slave Interface Common Register Read to Write only Error Interrupt clear Bit Writing a 1 to this bit clears the corresponding channel interrupt status bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_RD2WO_ERR_IntStat).
1	Clear_SLVIF_CommonReg_WR2RO_ERR_IntStat	W	0	Slave Interface Common Register Write to Read only Error Interrupt Clear Bit Writing a 1 to this bit clears the corresponding channel interrupt status bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_WR2RO_ERR_IntStat)
0	Clear_SLVIF_CommonReg_DEC_ERR_IntStat	W	0	Slave Interface Common Register Decode Error Interrupt Clear Bit Writing a 1 to this bit clears the corresponding channel interrupt status bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_DEC_ERR_IntStat)

6.2.1.7 DMAC_CommonReg_IntStatus_EnableReg

- **Name:** DW_axi_dmac Common Register Interrupt Status Enable Register
- **Size:** 64 bits
- **Offset:** 0x40
- **Read/Write Access:** Read/Write

Table 6-9 Fields for Register: DMAC_CommonReg_IntStatus_EnableReg

Bits	Name	Memory Access	Reset Value	Description
63:9	Reserved and read as one			
8	Enable_SLVIF_UndefinedReg_DEC_ERR_IntStat	R/W	1	Slave Interface Undefined Register Decode Error Interrupt Status Enable Bit Writing 1 in this bit enables the corresponding bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_UndefinedReg_DEC_ERR_IntStat).
7:4	Reserved and read as one			
3	Enable_SLVIF_CommonReg_WrOnHold_ERR_IntStat	R/W	1	Slave Interface Common Register Write On Hold Error Interrupt Status Enable Bit Writing 1 in this bit enables the corresponding bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_WrOnHold_ERR_IntStat).
2	Enable_SLVIF_CommonReg_RD2WO_ERR_IntStat	R/W	1	Slave Interface Common Register Read to Write only Error Interrupt Status Enable Bit Writing 1 in this bit enables the corresponding bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_RD2WO_ERR_IntStat).
1	Enable_SLVIF_CommonReg_WR2RO_ERR_IntStat	R/W	1	Slave Interface Common Register Write to Read only Error Interrupt Status Enable Bit Writing 1 in this bit enables the corresponding bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_WR2RO_ERR_IntStat).
0	Enable_SLVIF_CommonReg_DEC_ERR_IntStat	R/W	1	Slave Interface Common Register Decode Error Interrupt Status Enable Bit Writing 1 in this bit enables the corresponding bit in the DMAC_CommonReg_IntStatusReg register (SLVIF_CommonReg_DEC_ERR_IntStat).

6.2.1.8 DMAC_CommonReg_IntSignal_EnableReg

- **Name:** DW_axi_dmac Common Register Interrupt Signal Enable Register
- **Size:** 64 bits
- **Offset:** 0x48
- **Read/Write Access:** Read/Write

Table 6-10 Fields for Register: DMAC_CommonReg_IntSignal_EnableReg

Bits	Name	Memory Access	Reset Value	Description
63:9	Reserved and read as one			
8	Enable_SLVIF_UndefinedReg_DEC_ERR_IntSignal	R/W	1	Slave Interface Undefined register Decode Error Interrupt Signal Enable Bit Writing 1 to this bit propagates the corresponding channel interrupt status in the DMAC_CommonReg_IntStatusReg register to generate a port level interrupt.
7:4	Reserved and read as one			
3	Enable_SLVIF_CommonReg_WrOnHold_ERR_IntSignal	R/W	1	Slave Interface Common Register Write On Hold Error Interrupt Signal Enable Bit Writing 1 to this bit propagates the corresponding channel interrupt status in the DMAC_CommonReg_IntStatusReg register to generate a port level interrupt. **Note:** This feature is not supported in this release.
2	Enable_SLVIF_CommonReg_RD2WO_ERR_IntSignal	R/W	1	Slave Interface Common Register Read to Write only Error Interrupt Signal Enable Bit Writing 1 to this bit propagates the corresponding channel interrupt status in the DMAC_CommonReg_IntStatusReg register to generate a port level interrupt.
1	Enable_SLVIF_CommonReg_WR2RO_ERR_IntSignal	R/W	1	Slave Interface Common Register Write to Read only Error Interrupt Signal Enable Bit Writing 1 to this bit propagates the corresponding channel interrupt status in the DMAC_CommonReg_IntStatusReg register to generate a port level interrupt.
0	Enable_SLVIF_CommonReg_DEC_ERR_IntSignal	R/W	1	Slave Interface Common Register Decode Error Interrupt Signal Enable Bit Writing 1 to this bit propagates the corresponding channel interrupt status in the DMAC_CommonReg_IntStatusReg register to generate a port level interrupt.

6.2.1.9 DMAC_CommonReg_IntStatusReg

This Register captures the following Slave interface access errors:

- ❑ Decode Error
- ❑ Write to read only register
- ❑ Read to write only register
- ❑ write on hold
- ❑ undefined address

- **Name:** DW_axi_dmac Common Register Interrupt Status Register
- **Size:** 64 bits
- **Offset:** 0x50
- **Read/Write Access:** Read

Table 6-11 Fields for Register: DMAC_CommonReg_IntStatusReg

Bits	Name	Memory Access	Reset Value	Description
63:9	Reserved and read as zero			
8	SLVIF_UndefinedReg_DEC_ERR_IntStat	R	0	Slave Interface Undefined register Decode Error Interrupt Signal Enable Bit Decode Error generated by DW_axi_dmac during register access. This error occurs if the register access is to undefined address range (>0x8FF if 8 channels are configured, >0x4FF if 4 channels are configured, and so on), resulting in error response by DW_axi_dmac slave interface. 0: No Slave Interface Decode Errors 1: Slave Interface Decode Error detected The Error Interrupt Status is generated if the corresponding Status Enable bit in the DMAC_CommonReg_IntStatus_Enable register is set to 1. Error Interrupt output is generated if the corresponding channel interrupt signal enable bit in the DMAC_CommonReg_IntSignal_EnableReg is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in DMAC_CommonReg_IntClearReg on enabling the channel (needed when the interrupt is not enabled).
7:4	Reserved and read as zero			

Table 6-11 Fields for Register: DMAC_CommonReg_IntStatusReg (Continued)

Bits	Name	Memory Access	Reset Value	Description
3	SLVIF_CommonReg_WrOnHold_ERR_IntStat	R	0	Slave Interface Common Register Write On Hold Error Interrupt Status Bit This error occurs if an illegal write operation is performed on a common register; this happens if a write operation is performed on a common register except DMAC_ResetReg with the DMAC_RST field set to '1' when DW_axi_dmac is in Hold mode. 0: No Slave Interface Common Register Write On Hold Errors 1: Slave Interface Common Register Write On Hold Error detected The Error Interrupt Status is generated if the corresponding Status Enable bit in the DMAC_CommonReg_IntStatus_Enable register is set to 1. Error Interrupt output is generated if the corresponding channel interrupt signal enable bit in the DMAC_CommonReg_IntSignal_EnableReg is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in DMAC_CommonReg_IntClearReg on enabling the channel (needed when the interrupt is not enabled).
2	SLVIF_CommonReg_RD2WO_ERR_IntStat	R	0	Slave Interface Common Register Read to Write only Error Interrupt Status Bit This error occurs if Read operation is performed to a Write Only register in the common register space (0x000 to 0x0FF). 0 = No Slave Interface Read to Write Only Errors 1 = Slave Interface Read to Write Only Error detected The Error Interrupt Status is generated if the corresponding Status Enable bit in the DMAC_CommonReg_IntStatus_Enable register is set to 1. Error Interrupt output is generated if the corresponding channel interrupt signal enable bit in the DMAC_CommonReg_IntSignal_EnableReg is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in DMAC_CommonReg_IntClearReg on enabling the channel (needed when the interrupt is not enabled).

Table 6-11 Fields for Register: DMAC_CommonReg_IntStatusReg (Continued)

Bits	Name	Memory Access	Reset Value	Description
1	SLVIF_CommonReg_WR2RO_ERR_IntStat	R	0	Slave Interface Common Register Write to Read Only Error Interrupt Status Bit This error occurs if write operation is performed to a Read Only register in the common register space (0x000 to 0x0FF). 0 = No Slave Interface Write to Read Only Errors 1 = Slave Interface Write to Read Only Error detected The Error Interrupt Status is generated if the corresponding Status Enable bit in the DMAC_CommonReg_IntStatus_Enable register is set to 1. Error Interrupt output is generated if the corresponding channel interrupt signal enable bit in the DMAC_CommonReg_IntSignal_EnableReg is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in DMAC_CommonReg_IntClearReg on enabling the channel (needed when the interrupt is not enabled).
0	SLVIF_CommonReg_DEC_ERR_IntStat	R	0	Slave Interface Common Register Decode Error Interrupt Status Bit Decode Error generated by DW_axi_dmac during register access. This error occurs if the register access is to invalid address in the common register space (0x000 to 0x0FF) resulting in error response by DW_axi_dmac slave interface. 0 = No Slave Interface Decode Errors. 1 = Slave Interface Decode Error detected. The Error Interrupt Status is generated if the corresponding Status Enable bit in the DMAC_CommonReg_IntStatus_Enable register is set to 1. Error Interrupt output is generated if the corresponding channel interrupt signal enable bit in the DMAC_CommonReg_IntSignal_EnableReg is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in DMAC_CommonReg_IntClearReg on enabling the channel (needed when the interrupt is not enabled).

6.2.1.10 DMAC_ResetReg

This register is used to initiate the Software Reset to DW_axi_dmac.

- **Name:** DW_axi_dmac Reset Register
- **Size:** 64 bits
- **Offset:** 0x58
- **Read/Write Access:** Read/Write

Table 6-12 Fields for Register: DMAC_ResetReg

Bits	Name	Memory Access	Reset Value	Description
63:1	Reserved for future use.			
0	DMAC_RST	R/W	0	DMAC Reset Request bit Software writes 1 to this bit to reset the DW_axi_dmac and polls this bit to see it as 0. DW_axi_dmac resets all the modules except the slave bus interface module and clears this bit to 0. **Note:** Software is not allowed to write 0 to this bit.

6.2.2 DW_axi_dmac Channel Registers

This section provides details on the channel registers of the DW_axi_dmac.

6.2.2.1 CH*x*_SAR

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 AXI transfer.

Note: SAR address must be programmed to be aligned to CH*x*_CTL.SRC_TR_WIDTH.

- **Name:** Source Address Register for Channel*x*
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_SAR: 0x100
  CH2_SAR: 0x200
  CH3_SAR: 0x300
  CH4_SAR: 0x400
  CH5_SAR: 0x500
  CH6_SAR: 0x600
  CH7_SAR: 0x700
  CH8_SAR: 0x800
- **Read/Write Access:** Read/Write

Table 6-13 Fields for Register: CH*x*_SAR

Bits	Name	Memory Access	Reset Value	Description
63:0	SAR	R/W	0	Current Source Address of DMA transfer. Updated after each source transfer. The SINC fields in the CH*x*_CTL register determines whether the address increments or is left unchanged on every source transfer throughout the block transfer.

6.2.2.2 CH*x*_DAR

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 AXI transfer.

Note: DAR address must be programmed to be aligned to CH*x*_CTL.DST_TR_WIDTH.

- **Name:** Source Address Register for Channel*x*
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_DAR: 0x108
  CH2_DAR: 0x208
  CH3_DAR: 0x308
  CH4_DAR: 0x408
  CH5_DAR: 0x508
  CH6_DAR: 0x608
  CH7_DAR: 0x708
  CH8_DAR: 0x808
- **Read/Write Access:** Read/Write

Table 6-14 Fields for Register: CH*x*_DAR

Bits	Name	Memory Access	Reset Value	Description
63:0	DAR	R/W	0	Current Destination Address of DMA transfer. Updated after each destination transfer. The DINC fields in the CH*x*_CTL register determines whether the address increments or is left unchanged on every destination transfer throughout the block transfer.

6.2.2.3 CH*x*_BLOCK_TS

When DW_axi_dmac is the flow controller, the DMAC uses this register before the channel is enabled for block size.

- **Name:** Block Transfer Size Register for Channel*x*
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_BLOCK_TS: 0x110
  CH2_BLOCK_TS: 0x210
  CH3_BLOCK_TS: 0x310
  CH4_BLOCK_TS: 0x410
  CH5_BLOCK_TS: 0x510
  CH6_BLOCK_TS: 0x610
  CH7_BLOCK_TS: 0x710
  CH8_BLOCK_TS: 0x810
- **Read/Write Access:** Read/Write

Table 6-15 Fields for Register: CH*x*_BLOCK_TS

Bits	Name	Memory Access	Reset Value	Description
63:22	Reserved and read as zero			
21:0	BLOCK_TS	R/W	0	Block Transfer Size. The number programmed into BLOCK_TS field indicates the total number of data of width CH*x*_CTL.SRC_TR_WIDTH to be transferred in a DMA block transfer. Block Transfer Size = BLOCK_TS+1

6.2.2.4 CH*x*_CTL

This register contains fields that control the DMA transfer. This register should be programmed prior to enabling the channel except for LLI-based multi block transfer. When LLI-based multi-block transfer is enabled, the CH1_CTL register is loaded from the corresponding location of the LLI and it can be varied on a block-by-block basis within a DMA transfer. The software is not allowed to directly update this register through DW_axi_dmac slave interface. Any write to this register during LLI-based multi-block transfer is ignored.

- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_CTL: 0x118
  CH2_CTL: 0x218
  CH3_CTL: 0x318
  CH4_CTL: 0x418
  CH5_CTL: 0x518
  CH6_CTL: 0x618
  CH7_CTL: 0x718
  CH8_CTL: 0x818
- **Read/Write Access:** Read/Write

Table 6-16 Fields for Register: CH*x*_CTL

Bits	Name	Memory Access	Reset Value	Description
63	SHADOWREG_OR_LLI_VALID	R/W	0	Shadow Register content/Linked List Item Valid Indicates whether the content of shadow register or the linked list item fetched from the memory is valid or not. 0 = Shadow Register content/LLI is invalid 1 = Last Shadow Register/LLI is valid **LLI-based multiblock transfer:** The CHx_CTL register is loaded from the LLI. Hence, the software is not allowed to directly update this register through the DW_axi_dmac slave interface. This field can be used to dynamically extend the LLI by the software. On seeing this bit as 0, DW_axi_dmac discards the LLI and generates the ShadowReg_Or_LLI_Invalid_ERR Interrupt if the corresponding channel error interrupt mask bit is set to 0. In the case of LLI pre-fetching, the ShadowReg_Or_LLI_Invalid_ERR interrupt is not generated even if the ShadowReg_Or_LLI_Valid bit is seen to be 0 for the pre-fetched LLI. In this case, DW_axi_dmac attempts the LLI fetch operation again after completing the current block transfer and generates the ShadowReg_Or_LLI_Invalid_ERR interrupt only if ShadowReg_Or_LLI_Valid bit is still seen to be 0.

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
63 *Cont'd*	SHADOWREG_OR_LLI_VALID	R/W	0	*Continued* This error condition causes the DW_axi_dmac to halt the corresponding channel gracefully. DW_axi_dmac waits until software writes (any value) to CHx_BLK_TFR_ResumeReqReg to indicate valid LLI availability before attempting another LLI read operation. This bit is cleared to 0 and written back to the corresponding LLI location after block transfer completion when LLI write-back option is enabled. Hence, for LLI-based multi-block transfers, the software might manipulate/redefine any descriptor with the ShadowReg_Or_LLI_Valid bit set to 0 if LLI write-back option is enabled. **Shadow-Register-based multiblock transfer** On seeing this bit as 0 during shadow register fetch phase, DW_axi_dmac discards the Shadow Register contents and generates ShadowReg_Or_LLI_Invalid_ERR Interrupt. In this case, software has to write (any value) to CHx_BLK_TFR_ResumeReqReg after updating the shadow registers and setting ShadowReg_Or_LLI_Valid bit to 1 to indicate to DW_axi_dmac that shadow register contents are valid and the next block transfer can be resumed. DW_axi_dmac clears this bit to 0 after copying the shadow register contents. Software can reprogram the shadow registers only if ShadowReg_Or_LLI_Valid bit is 0. Software needs to read this register in block completion interrupt service routine (if interrupt is enabled)/ continuously poll this register (if interrupt is not enabled) to make sure that this bit is 0 before updating the shadow registers. If shadow-register-based multi-block transfer is enabled and software tries to write to the shadow register when ShadowReg_Or_LLI_Valid bit is 1, DW_axi_dmac generates SLVIF_ShadowReg_WrOnValid_ERR interrupt. ■ **Dependencies:** This field is relevant only if linked-list / shadow-register-based multi block transfer is enabled.

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
62	SHADOWREG_OR_LLI_LAST	R/W	0	Last Shadow Register/Linked List Item. Indicates whether shadow register content or the linked list item fetched from the memory is the last one. 0 = Not last Shadow Register/LLI 1 = Last Shadow Register/LLI **LLI-based multiblock transfer:** DW_axi_dmac uses this bit to decide if another LLI fetch is needed in the current DMA transfer. ■ If this bit is 0, DW_axi_dmac fetches the next LLI from the address pointed out by LLP field in the current LLI. ■ If this bit is 1, DW_axi_dmac understands that current block is the final block in the DMA transfer and ends the DMA transfer once the AMBA transfer corresponding to the current block completes. **Shadow-Reg-based multiblock transfer:** DW_axi_dmac uses this bit to decide if another Shadow Register fetch is needed in the current DMA transfer. ■ If this bit is 0, DW_axi_dmac understands that there are one or more blocks to be transferred in the current block and hence one or more shadow register set contents will be valid and needs to be fetched. ■ If this bit is 1, DW_axi_dmac understands that current block is the final block in the DMA transfer and ends the DMA transfer once the AMBA transfer corresponding to the current block completes. **Dependencies:** This field is relevant only if shadow-register or linked-list-based multi block transfer is enabled. **Note:** ShadowReg_Or_LLI_Valid and ShadowReg_Or_LLI_Last fields can be used to achieve undefined length (total number of blocks is not known in advance) DMA transfer by dynamic extension of Shadow Register contents/LLI.
61:59	Reserved and read as zero.			

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
58	IOC_BlkTfr	R/W	0	Interrupt On completion of Block Transfer This bit is used to control the block transfer completion interrupt generation on a block by block basis for shadow-register or linked-list-based multi-block transfers. Writing 1 to this register field enables the CHx_IntStatusReg.BLOCK_TFR_DONE_IntStat field if this interrupt generation is enabled in CHx_IntStatus_EnableReg register and the external interrupt output is asserted if this interrupt generation is enabled in CHx_IntSignal_EnableReg register. **Note:** If a linked-list or shadow-register-based multi-block transfer is not used for both source and destination (for instance if source and destination use contiguous-address or auto-reload-based multi-block transfer), the value of this field can not be modified per block. Additioanlly, the value programmed before the channel is enabled is used for all the blocks in the DMA transfer.
57	DST_STAT_EN	R/W	0	Destination Status Enable Enable the logic to fetch status from destination peripheral of channel x pointed to by the content of CHx_DSTATAR register and stores it in CHx_DSTAT register. This value is written back to the CHx_DSTAT location of linked list at end of each block transfer if DMAX_CHx_LLI_WB_EN is set to 1 and if linked-list-based multi-block transfer is used by either source or destination peripheral. **Dependencies:** This field is present only if coreConsultant parameter DMAX_CHx_DST_STAT_EN is True.
56	SRC_STAT_EN	R/W	0	Source Status Enable Enable the logic to fetch status from source peripheral of channel x pointed to by the content of CH1_SSTATAR register and stores it in CHx_SSTAT register. This value is written back to the CHx_SSTAT location of linked list at end of each block transfer if DMAX_CHx_LLI_WB_EN is set to 1 and if linked-list-based multi-block transfer is used by either source or destination peripheral. **Dependencies:** This field is present only if coreConsultant parameter DMAX_CHx_SRC_STAT_EN is True.

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
55:48	AWLEN	R/W	0	Destination Burst Length AXI Burst length used for destination data transfer. The specified burst length is used for destination data transfer until the extent possible; remaining transfers will use maximum possible. The maximum value of AWLEN is limited by DMAX_CHx_MAX_AMBA_BURST_LENGTH **Note:** The AWLEN setting may not be honored towards end-of-block transfers, the end of a transaction (only applicable to non-memory peripherals), and during 4K boundary crossings.
47	AWLEN_EN	R/W	0	Destination Burst Length Enable If this bit is set to 1, DW_axi_dmac uses the value of CH*x*_CTL.AWLEN as AXI Burst length for destination data transfer until the extent possible; remaining transfers will use maximum possible burst length. If this bit is set to 0, DW_axi_dmac uses any possible value which is less than or equal to DMAX_CHx_MAX_AMBA_BURST_LENGTH as AXI Burst length for destination data transfer.
46:39	ARLEN	R/W	0	Source Burst Length AXI Burst length used for source data transfer. The specified burst length is used for source data transfer until the extent possible; remaining transfers will use maximum possible value which is less than or equal to DMAX_CH1_MAX_AMBA_BURST_LENGTH. The maximum value of ARLEN is limited by DMAX_CH1_MAX_AMBA_BURST_LENGTH. **Note:** The ARLEN setting may not be honored towards end-of-block transfers, the end of a transaction (only applicable to non-memory peripherals), and during 4K boundary crossings.
38	ARLEN_EN	R/W	0	Source Burst Length Enable If this bit is set to 1, DW_axi_dmac uses the value of CH*x*_CTL.ARLEN as AXI Burst length for source data transfer till the extent possible; remaining transfers will use maximum possible burst length. If this bit is set to 0, DW_axi_dmac uses any possible value which is less than or equal to DMAX_CHx_MAX_AMBA_BURST_LENGTH as AXI Burst length for source data transfer.
37:35	AW_PROT	R/W	0x000	AXI 'aw_prot' signal
34:32	AR_PROT	R/W	0x000	AXI 'ar_prot' signal
31	Reserved and read as zero.			

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
30	NonPosted_LastWrite_En	R/W	0	Non Posted Last Write Enable This bit decides whether posted writes can be used throughout the block transfer or not. ■ 0 = Posted writes may be used throughout the block transfer ■ 1 = Posted writes may be used till the end of the block (inside a block) and the last write in the block must be non-posted. This is to synchronize block completion interrupt generation to the last write data reaching the end memory/peripheral.
29:26	AW_CACHE	R/W	0x0000	AXI 'aw_cache' signal
25:22	AR_CACHE	R/W	0x0000	AXI 'ar_cache' signal
21:18	DST_MSIZE	R/W	0x0000	Destination Burst Transaction Length. Number of data items, each of width CH*x*_CTL.DST_TR_WIDTH, to be written to the destination every time a destination burst transaction request is made from the corresponding hardware or software handshaking interface. Table 6-17 on page 153 lists the decoding for this field. The maximum value of DST_MSIZE is limited by DMAX_CHx_MAX_MULT_SIZE. **Note:** This Value is not related to the AXI awlen signal.
17:14	SRC_MSIZE	R/W	0x0000	Source Burst Transaction Length. Number of data items, each of width CH*x*_CTL.SRC_TR_WIDTH, to be read from the source every time a source burst transaction request is made from the corresponding hardware or software handshaking interface. Table 6-17 on page 153 lists the decoding for this field. The maximum value of DST_MSIZE is limited by DMAX_CHx_MAX_MULT_SIZE **Note:** This Value is not related to the AXI arlen signal.
13:11	DST_TR_WIDTH	R/W	0x010	Destination Transfer Width Table 6-18 on page 153 lists the decoding for this field. Mapped to AXI bus awsize. This value must be less than or equal to DMAX_M_DATA_WIDTH **Dependencies:** This field does not exist if the parameter DMAX_CH*x*_DTW is hardcoded. In this case, the read-back value is always the hardcoded destination transfer width, DMAX_CHx_DTW, where x is the AXI layer 1 to 2 where the source resides.

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
10:8	SRC_TR_WIDTH	R/W	0x010	Source Transfer Width Table 6-18 on page 153 lists the decoding for this field. Mapped to AXI bus arsize. This value must be less than or equal to DMAX_M_DATA_WIDTH. **Dependencies:** This field does not exist if the parameter DMAX_CHx_STW is hardcoded. In this case, the read-back value is always the hardcoded source transfer width, DMAX_CHx_STW, where x is the AXI layer 1 to 2 where the source resides.
7	Reserved and read as zero			
6	DINC	R/W	0	Destination Address Increment Indicates whether to increment the destination address on every destination transfer. If the device is writing data from a source peripheral FIFO with a fixed address, then set this field to 'No change'. 0 = Increment 1 = No Change **Note:** Increment aligns the address to the next CH*x*_CTL.DST_TR_WIDTH boundary
5	Reserved and read as zero			
4	SINC	R/W	0	Source Address Increment Indicates whether to increment 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'. 0 = Increment 1 = No Change **Note:** Increment aligns the address to the next CH*x*_CTL.SRC_TR_WIDTH boundary
3	Reserved and read as zero			
2	DMS	R/W	0	Destination Master Select Identifies the Master Interface layer from which the destination device (peripheral or memory) is accessed. 0 = AXI master 1 1 = AXI Master 2 **Dependencies:** This field does not exist if the configuration parameter DMAX_CHx_DMS is defined (hardcoded); in this case, the read-back value is always the hardcoded value.
1	Reserved and read as zero			

Table 6-16 Fields for Register: CH*x*_CTL (Continued)

Bits	Name	Memory Access	Reset Value	Description
0	SMS	R/W	0	Source Master Select. Identifies the Master Interface layer from which the source device (peripheral or memory) is accessed. 0 = AXI master 1 1 = AXI Master 2 **Dependencies:** This field does not exist if the configuration parameter DMAX_CHx_SMS is defined (hardcoded); in this case, the read-back value is always the hardcoded value.

Table 6-17 CH*x*_CTL.SRC_MSIZE and CH*x*_CTL.DST_MSIZE Decoding

CHx_CTL.SRC_MSIZE/ CHx_CTL.DST_MSIZE	Number of data items to be transferred (of width CHx_CTL.SRC_TR_WIDTH or CHx_CTL.DST_TR_WIDTH)
0000	1
0001	4
0010	8
0011	16
0100	32
0101	64
0110	128
0111	256
1000	512
1001	1024
1010	1024
1011	1024
1100	1024
1101	1024
1110	1024
1111	1024

Table 6-18 CH*x*_CTL.SRC_TR_WIDTH and CH*x*_CTL.DST_TR_WIDTH Decoding

CHx_CTL.SRC_TR_WIDTH / CHx_CTL.DST_TR_WIDTH	Size (bits)
000	8
001	16
010	32

Table 6-18 CH*x*_CTL.SRC_TR_WIDTH and CH*x*_CTL.DST_TR_WIDTH Decoding

CHx_CTL.SRC_TR_WIDTH / CHx_CTL.DST_TR_WIDTH	Size (bits)
011	64
100	128
101	256
110	512
111	512

6.2.2.5 CH*x*_CFG

This register contains fields that configure the DMA transfer. This register should be programmed prior to enabling the channel.

Bits [63:32] of the channel configuration register remains fixed for all blocks of a multi-block transfer and can be programmed only when channel is disabled.

Bits [3:0] of the channel configuration register can be programmed even when channel is enabled.

Software clears these bits to end the multi-block transfers. For Contiguous-Address and Auto-Reloading-based multi-block transfers (if neither source nor destination peripheral uses Shadow-Register or Linked-List-based multi-block transfers), if the corresponding multi-block type selection bits namely CH1_CFG.SRC_MLTBLK_TYPE and/or CH1_CFG.DST_MLTBLK_TYPE bits are seen to be 2'b00 at the end of a block transfer, the DW_axi_dmac understands that the previous block was the final block in the transfer and completes the DMA transfer operation.

- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_CFG: 0x120
  CH2_CFG: 0x220
  CH3_CFG: 0x320
  CH4_CFG: 0x420
  CH5_CFG: 0x520
  CH6_CFG: 0x620
  CH7_CFG: 0x720
  CH8_CFG: 0x820
- **Read/Write Access:** Read/Write

Table 6-19 Fields for Register: CH*x*_CFG

Bits	Name	Memory Access	Reset Value	Description
63	Reserved and read as zero			
62:59	DST_OSR_LMT	R/W	0	Destination Outstanding Request Limit ■ Maximum outstanding request supported = 16 ■ Destination Outstanding Request Limit = DST_OSR_LMT + 1
58:55	SRC_OSR_LMT	R/W	0	Source Outstanding Request Limit ■ Maximum outstanding request supported = 16 ■ Source Outstanding Request Limit = SRC_OSR_LMT + 1

Table 6-19 Fields for Register: CH*x*_CFG (Continued)

Bits	Name	Memory Access	Reset Value	Description
54:53	LOCK_CH_L	R/W	0x00	Channel Lock Level Indicates the duration over which CH1_CFG.LOCK_CH bit applies. ■ 00: Over complete DMA transfer ■ 01: Over DMA block transfer ■ 1x: Reserved This field does not exist if the configuration parameter DMAX_CH*x*_LOCK_EN is set to False; in that case, the read-back value is always 0.
52	LOCK_CH	R/W	0	Channel Lock bit When the channel is granted control of the master bus interface and if the CH*x*_CFG.LOCK_CH bit is asserted, then no other channels are granted control of the master bus interface for the duration specified in CH*x*_CFG.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 CH*x*_CFG.LOCK_CH_L. This field does not exist if the configuration parameter DMAX_CH*x*_LOCK_EN is set to False; in this case, the read-back value is always 0. Locking the channel locks AXI Read Address, Write Address and Write Data channels on the corresponding master interface. **Note:** Channel locking feature is supported only for memory to memory transfer at Block Transfer and DMA Transfer levels. Hardware doesn't check for the validity of channel locking setting, hence the software should take care of enabling the channel locking only for memory to memory transfers at Block Transfer or DMA Transfer levels. Illegal programming of channel locking might result in unpredictable behavior.
51:49	CH_PRIOR	R/W	See Desc.	Channel Priority A priority of 7 is the highest priority, and 0 is the lowest. This field must be programmed within the following range: 0: {DMAX_NUM_CHANNELS – 1) A programmed value outside this range causes erroneous behavior. Reset Value**:** 8-Channel Number. For example, Channel 1 Priority =7, Channel 2 Priority = 6 and so on.
48	Reserved and read as zero			

Table 6-19 Fields for Register: CH*x*_CFG (Continued)

Bits	Name	Memory Access	Reset Value	Description
b:44	DST_PER	R/W	0x0	Assigns a hardware handshaking interface (0 - DMAX_NUM_HS_IF-1) to the destination of channel1 if the CH*x*_CFG.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. **Note 1:** For correct DMA operation, only one peripheral (source or destination) should be assigned to the same handshaking interface. **Note 2:** This field does not exist if the configuration parameter DMAX_NUM_HS_IF is set to 0. **Note 3:** b = 44 if DMAX_NUM_HS_IF is 1 b = ceil(log2(DMAX_NUM_HS_IF)) + 43 if DMAX_NUM_HS_IF is greater than 1 Bits 47: (b+1) do not exist and return 0 on a read.
43	Reserved and read as zero			
b:39	SRC_PER	R/W	0x0	Assigns a hardware handshaking interface (0 - DMAX_NUM_HS_IF-1) to the source of channel1 if the CH*x*_CFG.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. Reset Value = 2*(Channel Number - 1) Channel Number = 1 to 8 depending on the value of DMAX_NUM_CHANNELS. **Note 1:** For correct DW_axi_dmac operation, only one peripheral (source or destination) should be assigned to the same handshaking interface. **Note 2:** This field does not exist if the configuration parameter DMAX_NUM_HS_IF is set to 0. **Note 3:** b = 39 if DMAX_NUM_HS_IF is 1 b = ceil(log2(DMAX_NUM_HS_IF)) + 38 if DMAX_NUM_HS_IF is greater than 1 Bits 42: (b+1) do not exist and return 0 on a read.
38	DST_HWHS_POL	R/W	0	Destination Hardware Handshaking Interface Polarity ■ 0: ACTIVE HIGH ■ 1: ACTIVE LOW **Note:** Polarity selection is not supported in the first version. Only Active High polarity is supported in first version.

Table 6-19 Fields for Register: CH*x*_CFG (Continued)

Bits	Name	Memory Access	Reset Value	Description
37	SRC_HWHS_POL	R/W	0	Source Hardware Handshaking Interface Polarity. ■ 0: ACTIVE HIGH ■ 1: ACTIVE LOW **Note:** Polarity selection is not supported in the first version. Only Active High polarity is supported in first version.
36	HS_SEL_DST	R/W	1	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. ■ 0 = Hardware handshaking interface. Software-initiated transaction requests are ignored. ■ 1 = Software handshaking interface. Hardware-initiated transaction requests are ignored. If the destination peripheral is memory, then this bit is ignored.
35	HS_SEL_SRC	R/W	1	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. ■ 0 = Hardware handshaking interface. Software-initiated transaction requests are ignored. ■ 1 = Software handshaking interface. Hardware-initiated transaction requests are ignored. If the source peripheral is memory, then this bit is ignored.
34:32	TT_FC	R/W	011	Transfer Type and Flow Control The following transfer types are supported: ■ Memory to Memory ■ Memory to Peripheral ■ Peripheral to Memory ■ Peripheral to Peripheral Flow Control can be assigned to the DW_axi_dmac, the source peripheral, or the destination peripheral. Table 6-20 on page 161 lists the decoding for this field. **Dependencies:** ■ For multi-block transfers using linked list operation, TT_FC must be constant for all blocks of this multi-block transfer. ■ If the configuration parameter DMAX_CH*x*_TT_FC is set to any value other than NO_HARDCODE, then TT_FC[2:0] = DMAX_CH*x*_TT_FC.
31:4	Reserved and read as zero			

Table 6-19 Fields for Register: CH*x*_CFG (Continued)

Bits	Name	Memory Access	Reset Value	Description
3:2	DST_MULTBLK_ TYPE	R/W	See Desc.	Destination Multi Block Transfer Type. These bits define the type of multi-block transfer used for destination peripheral. ■ 00: Contiguous ■ 01: Reload ■ 10: Shadow Register ■ 11: Linked List If the type selected is Contiguous, the CH*x*_DAR register is loaded with the value of the end source address of previous block + 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Reload, the CH*x*_DAR register is reloaded from the initial value of DAR at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Shadow Register, the CH*x*_DAR register is loaded from the content of its shadow register if CH*x*_CTL.ShadowReg_Or_LLI_Valid bit is set to 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Linked List, the CH*x*_DAR register is loaded from the Linked List if CTL.ShadowReg_Or_LLI_Valid bit is set to 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. CH*x*_CTL and CH*x*_BLOCK_TS registers are loaded from their initial values or from the contents of their shadow registers (if CH*x*_CTL.ShadowReg_Or_LLI_Valid bit is set to 1) or from the linked list (if CTL.ShadowReg_Or_LLI_Valid bit is set to 1) at the end of every block for multi-block transfers based on the multi-block transfer type programmed for source and destination peripherals. Contiguous transfer on both source and destination peripheral is not a valid multi-block transfer configuration. This field does not exist if the configuration parameter DMAX_CH*x*_MULTI_BLK_EN is not selected; in that case, the read-back value is always 0. Reset Value: If DMAX_CHx_MULTI_BLK_TYPE is set to NO_HARDCODE then reset value is 00; otherwise it will take the hard coded value, Refer Table 3-3 on page 71 for details.

Table 6-19 Fields for Register: CH*x*_CFG (Continued)

Bits	Name	Memory Access	Reset Value	Description
1:0	SRC_MULTBLK_ TYPE	R/W	See Desc.	Source Multi Block Transfer Type. These bits define the type of multi-block transfer used for source peripheral. ■ 00: Contiguous ■ 01: Reload ■ 10: Shadow Register ■ 11: Linked List If the type selected is Contiguous, the CH*x*_SAR register is loaded with the value of the end source address of previous block + 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Reload, the CH*x*_SAR register is reloaded from the initial value of SAR at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Shadow Register, the CH*x*_SAR register is loaded from the content of its shadow register if CH*x*_CTL.ShadowReg_Or_LLI_Valid bit is set to 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. If the type selected is Linked List, the CH*x*_SAR register is loaded from the Linked List if CTL.ShadowReg_Or_LLI_Valid bit is set to 1 at the end of every block for multi-block transfers. A new block transfer is then initiated. CH*x*_CTL and CH*x*_BLOCK_TS registers are loaded from their initial values or from the contents of their shadow registers (if CH*x*_CTL.ShadowReg_Or_LLI_Valid bit is set to 1) or from the linked list (if CTL.ShadowReg_Or_LLI_Valid bit is set to 1) at the end of every block for multi-block transfers based on the multi-block transfer type programmed for source and destination peripherals. Contiguous transfer on both source and destination peripheral is not a valid multi-block transfer configuration. This field does not exist if the configuration parameter DMAX_CH*x*_MULTI_BLK_EN is not selected; in that case, the read-back value is always 0. Reset Value: If DMAX_CHx_MULTI_BLK_TYPE is set to NO_HARDCODE then reset value is 00; otherwise it will take the hard coded value, Refer Table 3-3 on page 71 for details.

Table 6-20 CH*x*_CFG.TT_FC Decoding

CHx_CFG.TT_FC	Transfer Type	Flow Controller
000	Memory to Memory	DW_axi_dmac
001	Memory to Peripheral	DW_axi_dmac
010	Peripheral to Memory	DW_axi_dmac
011	Peripheral to Peripheral	DW_axi_dmac
100	Peripheral to Memory	(Source) Peripheral
101	Peripheral to Peripheral	Source Peripheral
110	Memory to Peripheral	(Destination) Peripheral
111	Peripheral to Peripheral	Destination Peripheral

6.2.2.6 CH*x*_LLP

This is Linked List Pointer register. This register should be programmed to point to the first Linked List Item (LLI) in memory prior to enabling the channel if linked-list-based block chaining is enabled. This register will be updated with new value of linked list pointer during the LLI update stage of DMA transfer.

- **Name:** Linked List pointer register for Channel*x*
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_LLP: 0x128
  CH2_LLP: 0x228
  CH3_LLP: 0x328
  CH4_LLP: 0x428
  CH5_LLP: 0x528
  CH6_LLP: 0x628
  CH7_LLP: 0x728
  CH8_LLP: 0x828
- **Read/Write Access:** Read/Write

Table 6-21 Fields for Register: CH*x*_LLP

Bits	Name	Memory Access	Reset Value	Description
63:6	LOC	R/W	0	Starting Address In Memory of next LLI if block chaining is enabled. The six LSBs of the starting address are not stored because the address is assumed to be aligned to a 64-byte boundary. LLI access always uses the burst size (arsize/awsize) that is same as the data bus width and cannot be changed or programmed to anything other than this. Burst length (awlen/arlen) is chosen based on the data bus width so that the access does not cross one complete LLI structure of 64 bytes. DW_axi_dmac fetches the entire LLI (40 bytes) in one AXI burst if the burst length is not limited by other settings.
5:1	Reserved and read as zero			
0	LMS	R/W	0	LLI master Select Identifies the AXI layer/interface where the memory device that stores the next linked list item resides. ■ 0: AXI Master 1 ■ 1: AXI Master 2 This field does not exist if the configuration parameter DMAX_CH*x*_LMS 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 'DMAX_NUM_MASTER_IF-1'.

6.2.2.7 CH*x*_STATUSREG

Channel*x* Status Register contains fields that indicate the status of DMA transfers for Channel*x*.

- **Name:** Channel*x* Status Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_STATUSREG: 0x130
  CH2_STATUSREG: 0x230
  CH3_STATUSREG: 0x330
  CH4_STATUSREG: 0x430
  CH5_STATUSREG: 0x530
  CH6_STATUSREG: 0x630
  CH7_STATUSREG: 0x730
  CH8_STATUSREG: 0x830
- **Read/Write Access:** Read

Table 6-22 Fields for Register: CHx_STATUSREG

Bits	Name	Memory Access	Reset Value	Description
63:47	Reserved and read as zero			
46:32	DATA_LEFT_IN_FIFO	R	0	Data Left in FIFO This field indicates the total number of data left in DW_axi_dmac channel FIFO after completing the current block transfer. The width of the data in channel FIFO is equal to CH*x*_CTL.SRC_TR_WIDTH. For normal block transfer completion without errors, Data_Left_In_FIFO = 0. If any error occurs during the DMA transfer, the block transfer might be terminated early and in such a case, Data_Left_In_FIFO indicates the data remaining in channel FIFO which could not be transferred to destination peripheral. This field is cleared to zero on enabling the channel. **Note:** If CH*x*_CTL.DST_TR_WIDTH > CH*x*_CTL.SRC_TR_WIDTH, there may be residual data left in the FIFO which is not enough to form one CH*x*_CTL.SRC_TR_WIDTH of data and Data_Left_In_FIFO will return 0 in this case.
31:22	Reserved and read as zero.			

Table 6-22 Fields for Register: CHx_STATUSREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
21:0	CMPLTD_BLK_TFR_SIZE	R	0	Completed Block Transfer Size This field indicates the total number of data of width CH*x*_CTL.SRC_TR_WIDTH transferred for the previous block transfer. For normal block transfer completion without any errors, this value will be equal to the value programmed in BLOCK_TS field of CH*x*_BLOCK_TS register. If any error occurs during the DMA transfer, the block transfer might be terminated early and in such a case, this value indicates the actual data transferred without error in the current block. This field is cleared to zero on enabling the channel.

6.2.2.8 CH*x*_SWHSSRCREG

- **Name:** Channel*x* Software Handshake Source Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_SWHSSRCREG: 0x138
  CH2_SWHSSRCREG: 0x238
  CH3_SWHSSRCREG: 0x338
  CH4_SWHSSRCREG: 0x438
  CH5_SWHSSRCREG: 0x538
  CH6_SWHSSRCREG: 0x638
  CH7_SWHSSRCREG: 0x738
  CH8_SWHSSRCREG: 0x838
- **Read/Write Access:** Read/Write

Table 6-23 Fields for Register: CH*x*_SWHSSRCREG

Bits	Name	Memory Access	Reset Value	Description
63:6	Reserved and read zero			
5	SWHS_LST_SRC_WE	W	0	Write Enable bit for Software Handshake Last Request for Channel Source. **Note:** This bit always returns 0 on read back.
4	SWHS_LST_SRC	R/W	0	Software Handshake Last Request for Channel Source. This bit is used to request LAST DMA source data transfer if software handshaking method is selected for the source of the corresponding channel. This bit is ignored if software handshaking is not enabled for the source of the channel1 or if the source of channel1 is not the flow controller. CH1_SWHSSrcReg.SWHS_Req_Src bit must be set to 1 for DW_axi_dmac to treat it as a valid software handshaking request. If CH1_SWHSSrcReg.SWHS_SglReq_Src is set to 1, the LAST request is for SINGLE DMA transaction (AXI burst length = 1), else the request is treated as a BURST transaction request. DW_axi_dmac clears this bit to 0 once software reads CH1_SWHSSrcReg.SWHS_Ack_Src bit and sees it as 1. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit.

Table 6-23 Fields for Register: CH*x*_SWHSSRCREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
4 *Cont'd*	SWHS_LST_SRC	R/W	0	*Continued* **Note:** SWHS_Lst_Src bit is written only if the corresponding write enable bit, SWHS_Lst_Src_WE is asserted on the same register write operation and if the channel1 is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CH1_SWHSSrcReg register without performing a read-modified write operation.
3	SWHS_SGLREQ_SRC_WE	W	0	Write Enable bit for Software Handshake Single Request for Channel Source.
2	SWHS_SGLREQ_SRC	R/W	0	Software Handshake Single Request for Channel Source. This bit is used to request SINGLE (AXI burst length = 1) DMA source data transfer if software handshaking method is selected for the source of the corresponding channel. This bit is ignored if software handshaking is not enabled for the source of the Channel*x*. The functionality of this field depends on whether the peripheral is the flow controller or not. DW_axi_dmac clears this bit to 0 once software reads CH*x*_SWHSSrcReg.SWHS_Ack_Src bit and sees it as 1. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit. Note: SWHS_SglReq_Src bit is written only if the corresponding write enable bit, SWHS_SglReq_Src_WE is asserted on the same register write operation and if the channel1 is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CH*x*_SWHSSrcReg register without performing a read-modified write operation.
1	SWHS_REQ_SRC_WE	W	0	Write Enable bit for Software Handshake Request for Channel Source **Note:** This bit always returns 0 on a read back.
0	SWHS_REQ_SRC	R/W	0	Software Handshake Request for Channel Source. This bit is used to request DMA source data transfer if software handshaking method is selected for the source of the corresponding channel. This bit is ignored if software handshaking is not enabled for the source of the channel*x*. The functionality of this field depends on whether the peripheral is the flow controller. DW_axi_dmac clears this bit to 0 when the source transaction is completed. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit to.

Table 6-23 Fields for Register: CH*x*_SWHSSRCREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
0 *Cont'd*	SWHS_REQ_SRC	R/W	0	**Note:** SWHS_Req_Src bit is written only if the corresponding write enable bit, SWHS_Req_Src_WE is asserted on the same register write operation and if the channel is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CHx_SWHSSrcReg register without performing a read-modified write operation.

6.2.2.9 CH*x*_SWHSDSTREG

- **Name:** Channel*x* Software Handshake Destination Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_SWHSDSTREG: 0x140
  CH2_SWHSDSTREG: 0x240
  CH3_SWHSDSTREG: 0x340
  CH4_SWHSDSTREG: 0x440
  CH5_SWHSDSTREG: 0x540
  CH6_SWHSDSTREG: 0x640
  CH7_SWHSDSTREG: 0x740
  CH8_SWHSDSTREG: 0x840
- **Read/Write Access:** Read/Write

Table 6-24 Fields for Register: CH*x*_SWHSDSTREG

Bits	Name	Memory Access	Reset Value	Description
63:6	Reserved and read as zero			
5	SWHS_LST_DST_WE	W	0	Write Enable bit for Software Handshake Last Request for Channel Destination **Note:** This bit always returns 0 on a read back.
4	SWHS_LST_DST	R/W	0	Software Handshake Last Request for Channel Destination This bit is used to request LAST DMA destination data transfer if software handshaking method is selected for the destination of the corresponding channel. This bit is ignored if software handshaking is not enabled for the destination of the Channel*x* or if the destination of channelx is not the flow controller. CH*x*_SWHSDstReg.SWHS_Req_Dst bit must be set to 1 for DW_axi_dmac to treat it as a valid software handshaking request. If CH*x*_SWHSDstReg.SWHS_SglReq_Dst is set to 1, the LAST request is for SINGLE DMA transaction (AXI burst length = 1), else the request is treated as a BURST transaction request. DW_axi_dmac clears this bit to 0 when software the destination transaction is completed. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit.

Table 6-24 Fields for Register: CH*x*_SWHSDSTREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
4 *Cont'd*	SWHS_LST_DST	R/W	0	*Continued* **Note:** SWHS_Lst_Src bit is written only if the corresponding write enable bit, SWHS_Lst_Src_WE is asserted on the same register write operation and if the channel1 is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CH*x*_SWHSDstReg register without performing a read-modified write operation.
3	SWHS_SGLREQ_DST_WE	W	0	Write Enable bit for Software Handshake Single Request for Channel Destination **Note:** This bit always returns 0 on a read back.
2	SWHS_SGLREQ_DST	R/W	0	Software Handshake Single Request for Channel Destination. This bit is used to request SINGLE (AXI burst length = 1) DMA destination data transfer if software handshaking method is selected for the destination of the corresponding channel. This bit is ignored if software handshaking is not enabled for the destination of the channel*x*. The functionality of this field depends on whether the peripheral is the flow controller. DW_axi_dmac clears this bit to 0 when the destination transaction is completed. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit. **Note:** SWHS_SglReq_Dst bit is written only if the corresponding write enable bit, SWHS_SglReq_Dst_WE is asserted on the same register write operation and if the channel*x* is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CH1_SWHSDstReg register without performing a read-modified write operation.
1	SWHS_REQ_DST_WE	W	0	Write Enable bit for Software Handshake Request for Channel Destination **Note:** This bit always returns 0 on read back.

Table 6-24 Fields for Register: CH*x*_SWHSDSTREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
0	SWHS_REQ_DST	R/W	0	Software Handshake Request for Channel Destination This bit is used to request DMA destination data transfer if software handshaking method is selected for the destination of the corresponding channel. This bit is ignored if software handshaking is not enabled for the destination of the channel*x*. The functionality of this field depends on whether the peripheral is the flow controller. DW_axi_dmac clears this bit to 0 when the destination transaction is completed. Software can only set this bit to 1; it is not allowed to clear this bit to 0; only DW_axi_dmac can clear this bit. **Note:** SWHS_Req_Dst bit is written only if the corresponding write enable bit, SWHS_Req_Dst_WE is asserted on the same register write operation and if the channel1 is enabled in the DMAC_ChEnReg register. This allows software to set a bit in the CH1_SWHSDstReg register without performing a read-modified write operation.

6.2.2.10 CH*x*_BLK_TFR_RESUMEREQREG

This register is used during Linked-List or Shadow-Register-based multi-block transfer.

- For Linked-List-based multi-block transfer, ShadowReg_Or_LLI_Valid bit in LLI.CH1_CTL indicates whether the linked list item fetched from the memory is valid or not (0: LLI is invalid, 1: LLI is valid).

  On seeing this bit as 0, DW_axi_dmac discards the LLI and generates ShadowReg_Or_LLI_Invalid_ERR Interrupt if the corresponding channel error interrupt mask bit is set to 0. This error condition causes the DW_axi_dmac to halt the corresponding channel gracefully. DW_axi_dmac waits till software writes (any value) to CH1_BLK_TFR_ResumeReqReg to indicate valid LLI availability, before attempting another LLI read operation.

- For Shadow-Register-based multi-block transfer, ShadowReg_Or_LLI_Valid bit in CH1_CTL register indicates whether the shadow register contents are valid or not (0: Shadow Register contents are invalid, 1: Shadow Register contents are valid).

  On seeing this bit as 0 during shadow register fetch phase, DW_axi_dmac discards the Shadow Register contents and generates ShadowReg_Or_LLI_Invalid_ERR Interrupt. DW_axi_dmac waits till software writes (any value) to CH1_BLK_TFR_ResumeReqReg to indicate valid shadow register availability, before attempting another shadow register fetch operation and continue the next block transfer.

Note For Shadow-Register-based multi-block transfers, when ShadowReg_Or_LLI_Invalid_ERR is generated, the recommended flow to resume transfer is as follows:

1. Software programs CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS and CHx_CTL registers with appropriate values for the next block.
2. Clear the interrupt using interrupt register CHx_IntClearReg
3. Program block resume request CHx_BLK_TFR_ResumeReqReg.

- **Name:** Channel*x* Block Transfer Resume Request Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_BLK_TFR_RESUMEREQREG: 0x148
  CH2_BLK_TFR_RESUMEREQREG: 0x248
  CH3_BLK_TFR_RESUMEREQREG: 0x348
  CH4_BLK_TFR_RESUMEREQREG: 0x448
  CH5_BLK_TFR_RESUMEREQREG: 0x548
  CH6_BLK_TFR_RESUMEREQREG: 0x648
  CH7_BLK_TFR_RESUMEREQREG: 0x748
  CH8_BLK_TFR_RESUMEREQREG: 0x847

- **Read/Write Access:** Write

Table 6-25 Fields for Register: CH*x*_BLK_TFR_RESUMEREQREG

Bits	Name	Memory Access	Reset Value	Description
63:1	Reserved for future use.			
0	BLK_TFR_RESUMEREQ	W	0	Block Transfer Resume Request during Linked-List or Shadow-Register-based multiblock transfer.

6.2.2.11 CH*x*_AXI_IDREG

This register is allowed to update only when the channel is disabled, which means it remains fixed for the entire DMA transfer.

Note The presence of this register is determined by the DMAX_M_ID_WIDTH and DMAX_NUM_CHANNELS configuration parameters.

- If LLI is enabled for any of the channel then the register is present only when:

  DMAX_M_ID_WIDTH – (log2(DMAX_NUM_CHANNELS) +1)> 0
- Otherwise:

  DMAX_M_ID_WIDTH – log2(DMAX_NUM_CHANNELS)> 0

- **Name:** Channel*x* AXI ID Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_AXI_IDREG: 0x150
  CH2_AXI_IDREG: 0x250
  CH3_AXI_IDREG: 0x350
  CH4_AXI_IDREG: 0x450
  CH5_AXI_IDREG: 0x550
  CH6_AXI_IDREG: 0x650
  CH7_AXI_IDREG: 0x750
  CH8_AXI_IDREG: 0x850
- **Read/Write Access:** Read/Write

Table 6-26 Fields for Register: CH*x*_AXI_IDREG

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as zero			
31:(16+IDW-L2NC-1)	Reserved and read as zero			

Table 6-26 Fields for Register: CH*x*_AXI_IDREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
(16+IDW-L2NC–2):16	AXI_WRITE_ID_SUFFIX	R/W	0	AXI Write ID Suffix These bits form part of the AWID output of AXI3/AXI4 master interface. IDW = DMAX_M_ID_WIDTH L2NC = log2(DMAX_NUM_CHANNELS) The upper L2NC+1 bits of awid_m*N* is derived from the channel number which is currently accessing the master interface. This varies for LLI fetch and source data transfer. For source data transfer, awid_m*N* for channel1 4'b0000, awid_m*N* for channel8 4'b0111, and so on. For LLI fetch access, awid_m*N* for channel1 4'b1000, awid_m*N* for channel8 4'b1111 and so on. Lower bits are same as the value programmed in CH*x*_AXI_IDReg.AXI_Write_ID_Suffix filed. **Note:** For AXI3 master interface, the same value is used for AXI Write ID.
15:(IDW-L2NC-1)	Reserved for future use.			
(IDW-L2NC–2):0	AXI_READ_ID_SUFFIX	R/W	0	AXI Read ID Suffix These bits form part of the ARID output of AXI3/AXI4 master interface. IDW = DMAX_M_ID_WIDTH L2NC = log2(DMAX_NUM_CHANNELS) The upper L2NC+1 bits of arid_m*N* is derived from the channel number which is currently accessing the master interface. This varies for LLI fetch and source data transfer. For source data transfer, arid_m*N* for channel1 4'b0000, arid_m*N* for channel8 4'b0111 and so on. For LLI fetch access, arid_m*N* for channel1 4'b1000, arid_m*N* for channel8 4'b1111 and so on. Lower bits are the same as the value programmed in CH*x*_AXI_IDReg.AXI_Read_ID_Suffix filed.

6.2.2.12 CH*x*_AXI_QOSREG

This register is allowed to update only when channel is disabled which means it remains fixed for the entire DMA transfer.

- **Name:** Channel*x* AXI QOS Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_AXI_QOSREG: 0x158
  CH2_AXI_QOSREG: 0x258
  CH3_AXI_QOSREG: 0x358
  CH4_AXI_QOSREG: 0x458
  CH5_AXI_QOSREG: 0x558
  CH6_AXI_QOSREG: 0x658
  CH7_AXI_QOSREG: 0x758
  CH8_AXI_QOSREG: 0x858
- **Read/Write Access:** Read/Write

Table 6-27 Fields for Register: CH*x*_AXI_QOSREG

Bits	Name	Memory Access	Reset Value	Description
63:8	Reserved and read as zero			
7:4	AXI_ARQOS	R/W	0	AXI ARQOS.These bits form the arqos output of AXI4 master interface.
3:0	AXI_AWQOS	R/W	0	AXI AWQOS.These bits form the awqos output of AXI4 master interface.

6.2.2.13 CH*x*_SSTAT

After each block transfer completes, hardware can retrieve the source status information from the address pointed to by the contents of the CH1_SSTATAR register. This status information is then stored in the CH1_SSTAT register and written out to the CH1_SSTAT register location of the LLI before the start of the next block.

Source status write-back to the CH1_SSTAT register location of the LLI is performed only if DMAX_CH1_LLI_WB_EN = 1 and linked-list-based multi-block transfer is enabled for either source or destination peripheral of the channel.

This register does not exist if DMAX_CHx_SRC_STAT_EN is set to False; in this case, the read-back value is always 0.

- **Name:** Channel*x* Source Status Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_SSTAT: 0x160
  CH2_SSTAT: 0x260
  CH3_SSTAT: 0x360
  CH4_SSTAT: 0x460
  CH5_SSTAT: 0x560
  CH6_SSTAT: 0x660
  CH7_SSTAT: 0x760
  CH8_SSTAT: 0x860
- **Read/Write Access:** Read

Table 6-28 Fields for Register: CH*x*_SSTAT

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as zero			
31:0	SSTAT	R	0	Source Status Source status information retrieved by hardware from the address pointed to by the contents of the CH*x*_SSTATAR register. Source peripheral should update the source status information, if any, at the location pointed to by CH*x*_SSTATAR to use this feature.This status is not related to any internal status of DW_axi_dmac. This status is not related to any internal status of DW_axi_dmac.

6.2.2.14 CH*x*_DSTAT

After each block transfer completes, hardware can retrieve the destination status information from the address pointed to by the contents of the CH*x*_DSTATAR register. This status information is then stored in the CH*x*_DSTAT register and written out to the CH1_DSTAT register location of the LLI before the start of the next block.

Destination status write-back to the CH*x*_DSTAT register location of the LLI is performed only if DMAX_CH*x*_LLI_WB_EN = 1 and linked-list-based multiblock transfer is enabled for either source or destination peripheral of the channel.

This register does not exist if DMAX_CHx_DST_STAT_EN is set to False; in this case, the read-back value is always 0.

- **Name:** Channel*x* Destination Status Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_DSTAT: 0x168
  CH2_DSTAT: 0x268
  CH3_DSTAT: 0x368
  CH4_DSTAT: 0x488
  CH5_DSTAT: 0x568
  CH6_DSTAT: 0x668
  CH7_DSTAT: 0x768
  CH8_DSTAT: 0x868
- **Read/Write Access:** Read

Table 6-29 Fields for Register: CH*x*_DSTAT

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as zero			
31:0	DSTAT	R	0	Destination Status Destination status information retrieved by hardware from the address pointed to by the contents of the CH1_DSTATAR register. Destination peripheral should update the destination status information, if any, at the location pointed to by CH1_DSTATAR to utilize this feature.This status is not related to any internal status of DW_axi_dmac. This status is not related to any internal status of DW_axi_dmac.

6.2.2.15 CH*x*_SSTATAR

After completion of each block transfer, hardware can retrieve the source status information from the user-defined address to which the contents of the CHx_SSTATAR register point. You can select any location in system memory that provides a 64-bit value to indicate the status of the source transfer.

This register does not exist if DMAX_CHx_SRC_STAT_EN is set to False; in this case, the read-back value is always 0.

- **Name:** Channel*x* Source Status Fetch Address Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_SSTATAR: 0x170
  CH2_SSTATAR: 0x270
  CH3_SSTATAR: 0x370
  CH4_SSTATAR: 0x470
  CH5_SSTATAR: 0x570
  CH6_SSTATAR: 0x670
  CH7_SSTATAR: 0x770
  CH8_SSTATAR: 0x870
- **Read/Write Access**: Read/Write

Table 6-30 Fields for Register: CH*x*_SSTATAR

Bits	Name	Memory Access	Reset Value	Description
63:0	SSTATAR	R/W	0	Source Status Fetch Address Pointer from where hardware can fetch the source status information, which is registered in the CH*x*_SSTAT register and written out to the CH*x*_SSTAT register location of the LLI before the start of the next block if DMAX_CH*x*_LLI_WB_EN = 1 and linked-list-based multi-block transfer is enabled for either source or destination peripheral of the channel. Source peripheral should update the source status information, if any, at the location pointed to by CH*x*_SSTATAR to utilize this feature. This status is not related to any internal status of DW_axi_dmac. This status is not related to the internal status of the DW_axi_dmac.

6.2.2.16 CH*x*_DSTATAR

After completion of each block transfer, hardware can retrieve the destination status information from the user-defined address to which the contents of the CH*x*_DSTATAR register point. You can select any location in system memory that would provide a 64-bit value to indicate the status of the destination transfer.

This register does not exist if DMAX_CHx_DST_STAT_EN is set to False; in this case, the read-back value is always 0.

- **Name:** Channel*x* Destination Status Fetch Address Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_DSTATAR: 0x178
  CH2_DSTATAR: 0x278
  CH3_DSTATAR: 0x378
  CH4_DSTATAR: 0x478
  CH5_DSTATAR: 0x578
  CH6_DSTATAR: 0x678
  CH7_DSTATAR: 0x778
  CH8_DSTATAR: 0x878
- **Read/Write Access**: Read/Write

Table 6-31 Fields for Register: CH*x*_DSTATAR

Bits	Name	Memory Access	Reset Value	Description
63:0	DSTATAR	R/W	0	Destination Status Fetch Address Pointer from where hardware can fetch the Destination status information, which is registered in the CH*x*_DSTAT register and written out to the CH*x*_DSTAT register location of the LLI before the start of the next block if DMAX_CH*x*_LLI_WB_EN = 1 and linked-list-based multiblock transfer is enabled for either source or destination peripheral of the channel. Destination peripheral updates the destination status information, if any, at the location pointed to by CH*x*_DSTATAR to utilize this feature. This status is not related to any internal status of DW_axi_dmac.

6.2.2.17 CH*x*_INTSTATUS_ENABLEREG

Writing 1 to the specific field enables the corresponding interrupt status generation in Channel*x* Interrupt Status Register (CH*x*_IntStatusReg).

- **Name:** Channel*x* Interrupt Status Enable Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_INTSTATUS_ENABLEREG: 0x180
  CH2_INTSTATUS_ENABLEREG: 0x280
  CH3_INTSTATUS_ENABLEREG: 0x380
  CH4_INTSTATUS_ENABLEREG: 0x480
  CH5_INTSTATUS_ENABLEREG: 0x580
  CH6_INTSTATUS_ENABLEREG: 0x680
  CH7_INTSTATUS_ENABLEREG: 0x780
  CH8_INTSTATUS_ENABLEREG: 0x880
- **Read/Write Access:** Read/Write

Table 6-32 Fields for Register: CH*x*_INTSTATUS_ENABLEREG

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as one			
31	Enable_CH_ABORTED_IntStat	R/W	1	Channel Aborted Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
30	Enable_CH_DISABLED_IntStat	R/W	1	Channel Disabled Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
29	Enable_CH_SUSPENDED_IntStat	R/W	1	Channel Suspended Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.

Table 6-32 Fields for Register: CH*x*_INTSTATUS_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
28	Enable_CH_SRC_SUSPENDED_ IntStat	R/W	1	Channel Source Suspended Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
27	Enable_ChLock_Cleared_IntStat	R/W	1	Channel Lock Cleared Interrupt Status Enable Bit Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted due to this interrupt.
26:22	Reserved and read as one			
21	Enable_SLVIF_WrOnHold_ERR_ IntStat	R/W	1	Slave Interface Write On Hold Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
20	Enable_SLVIF_ShadowReg_ WrOn_Valid_ERR_IntStat	R/W	1	Shadow Register Write On Valid Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
19	Enable_SLVIF_WrOnChEn_ERR_ IntStat	R/W	1	Slave Interface Write On Channel Enabled Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
18	Enable_SLVIF_RD2RWO_ERR_ IntStat	R/W	1	Slave Interface Read to write Only Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.

Table 6-32 Fields for Register: CH*x*_INTSTATUS_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
17	Enable_SLVIF_WR2RO_ERR_IntStat	R/W	1	Slave Interface Write to Read Only Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
16	Enable_SLVIF_DEC_ERR_IntStat	R/W	1	Slave Interface Decode Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
15	Reserved and read as one			
14	Enable_SLVIF_MultiBlkType_ERR_IntStat	R/W	1	Slave Interface Multi Block type Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
13	Enable_ShadowReg_Or_LLI_Invalid_ERR_IntStat	R/W	1	Shadow register or LLI Invalid Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
12	Enable_LLI_WR_SLV_ERR_IntStat	R/W	1	LLI WRITE Slave Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
11	Enable_LLI_RD_SLV_ERR_IntStat	R/W	1	LLI Read Slave Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.

Table 6-32 Fields for Register: CH*x*_INTSTATUS_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
10	Enable_LLI_WR_DEC_ERR_IntStat	R/W	1	LLI WRITE Decode Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
9	Enable_LLI_RD_DEC_ERR_IntStat	R/W	1	LLI Read Decode Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
8	Enable_DST_SLV_ERR_IntStat	R/W	1	Destination Slave Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
7	Enable_SRC_SLV_ERR_IntStat	R/W	1	Source Slave Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
6	Enable_DST_DEC_ERR_IntStat	R/W	1	Destination Decode Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
5	Enable_SRC_DEC_ERR_IntStat	R/W	1	Source Decode Error Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
4	Enable_DST_TransComp_IntStat	R/W	1	Destination Transaction Completed Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.

Table 6-32 Fields for Register: CH*x*_INTSTATUS_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
3	Enable_SRC_TransComp_IntStat	R/W	1	Source Transaction Completed Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
2	Reserved and read as one			
1	Enable_DMA_TFR_DONE_IntStat	R/W	1	DMA Transfer Done Interrupt Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.
0	Enable_BLOCK_TFR_DONE_IntStat	R/W	1	Block Transfer Done Interrupt Status Enable Writing a 1 to this register field enables the corresponding interrupt status bit in the CHx_IntStatusReg and the corresponding channel interrupt status bit in the DMAC_IntStatusReg to be asserted.

6.2.2.18 CH*x*_INTSTATUSREG

This register captures the Channel*x* specific interrupts.

- **Name:** Channel*x* Interrupt Status Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_INTSTATUSREG: 0x188
  CH2_INTSTATUSREG: 0x288
  CH3_INTSTATUSREG: 0x388
  CH4_INTSTATUSREG: 0x488
  CH5_INTSTATUSREG: 0x588
  CH6_INTSTATUSREG: 0x688
  CH7_INTSTATUSREG: 0x788
  CH8_INTSTATUSREG: 0x888
- **Read/Write Access:** Read

Table 6-33 CH*x*_INTSTATUSREG

Bits	Name	R/W	Reset Value	Description
63:32	Reserved and read as zero			
31	CH_ABORTED_IntStat	R	0	Channel Aborted Interrupt Status Bit This bit is set when Channel Abort request has been serviced by DW_axi_dmac. For details, refer to "Abnormal Channel Abort" on page 80. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
30	CH_DISABLED_IntStat	R	0	Channel Disabled Interrupt Status Bit This bit is set when Channel Disable request has been serviced by DW_axi_dmac. For details, refer to "Channel Suspend and Disable Prior to Transfer Completion" on page 78 and "Channel Disable Prior to Transfer Completion without Suspend" on page 79. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in the CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
29	CH_SUSPENDED_IntStat	R	0	Channel Suspended Interrupt Status Bit This bit is set when Channel Suspend request has been serviced by DW_axi_dmac. For details, refer to "Channel Suspend" on page 77. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).

Table 6-33 CH*x*_INTSTATUSREG *(Continued)*

Bits	Name	R/W	Reset Value	Description
28	CH_SRC_SUSPENDED_ IntStat	R	0	Channel Source Suspended Interrupt Status Bit This bit is set when the Source Peripheral has been suspended as a result of Suspend request from software. For more information, see “Channel Suspend” on page 77. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
27	ChLock_Cleared_IntStat	R	0	Channel Lock Cleared Interrupt Status Bit This bit indicates that channel locking is cleared. ■ 1 = Channel locking is cleared ■ 0 = Channel locking is not cleared This field has significance only if software has enabled channel locking at some transfer hierarchy. Channel locking is cleared by DW_axi_dmac during the following situations. ■ Channel locking is cleared and the channel locking settings in CHx_CFG register is reset if DW_axi_dmac disables the channel upon request from software. ■ Channel locking is cleared and the channel locking settings in CHx_CFG register is reset if DW_axi_dmac disables the channel upon receiving error response on the master interface. ■ Channel locking is cleared at end of each block if block level locking is enabled. ■ Channel locking is cleared at end of DMA transfer if transfer level locking is enabled. This bit is cleared to 0 on enabling the channel.
26:22	Reserved and read as zero			
21	SLVIF_WrOnHold_ERR_ IntStat	R	0	Slave Interface Write On Hold Error Interrupt Status Bit This bit is set when a Write is attempted to any register (except DMAC_RstReg) while DW_axi_dmac is on HOLD. This bit has significance only when parameter DMAX_HOLD_IO _EN is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).

Table 6-33 CH*x*_INTSTATUSREG *(Continued)*

Bits	Name	R/W	Reset Value	Description
20	SLVIF_ShadowReg_WrOnValid_ERR_IntStat	R	0	Slave Interface Shadow Register Write On Valid Error Interrupt Status Bit This bit it set when S/W tries to program the shadow registers when CH*x*_CTL.ShadowReg_Or_LLI_Valid bit is set to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
19	SLVIF_WrOnChEn_ERR_IntStat	R	0	Slave Interface Write On Channel Enabled Error Interrupt Status Bit This bit is set to when a write to the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, CH*x*_CTL, or CHx_LLP register addresses occur when the channel is enabled and if Shadow-Register-based multiblock transfer is not enabled. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
18	SLVIF_RD2WO_ERR_IntStat	R	0	Slave Interface Read to Write Only Error Interrupt Status Bit This bit is set when S/W performs a read on Write Only registers. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
17	SLVIF_WR2RO_ERR_IntStat	R	0	Slave Interface Write to Read Only Error Interrupt Status Bit This bit is set when S/W performs a write on Read only registers. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
16	SLVIF_DEC_ERR_IntStat	R	0	Slave Interface Decode Error Interrupt Status Bit This bit is set when S/W performs a Write/Read to an undefined address. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
15	Reserved and read as zero			
14	SLVIF_MultiBlkType_ERR_IntStat	R	0	Slave Interface Multi Block Type Error Interrupt Status Bit This bit is set when Software programs an illegal combination of Multiblock type in CHx_CFG register. For more information, see "Multiblock Transfer" on page 70. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).

Table 6-33 CH*x*_INTSTATUSREG *(Continued)*

Bits	Name	R/W	Reset Value	Description
13	ShadowReg_Or_LLI_ Invalid_ERR_IntStat	R	0	Shadow Register or LLI Invalid Error Interrupt Status Bit For Shadow-Register-based multi-block transfer, ShadowReg_Or_LLI_Valid bit in CH*x*_CTL register indicates whether the shadow register contents are valid or not (0: Shadow Register contents are invalid, 1: Shadow Register contents are valid). On seeing this bit as 0 during shadow register fetch phase, DW_axi_dmac discards the Shadow Register contents and sets ShadowReg_Or_LLI_Invalid_ERR_IntStat to 1. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
12	LLI_WR_SLV_ERR_IntStat	R	0	LLI Write Slave Error Interrupt Status Bit This bit is set to 1 when a LLI Write Back operation gets Slave Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
11	LLI_RD_SLV_ERR_IntStat	R	0	LLI Read Slave Error Interrupt Status Bit This bit is set to 1 when a LLI Read operation gets Slave Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
10	LLI_WR_DEC_ERR_ IntStat	R	0	LLI Write Decode Error Interrupt Status Bit This bit is set to 1 when a LLI Write operation gets Decode Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
9	LLI_RD_DEC_ERR_IntStat	R	0	LLI Read Decode Error Interrupt Status Bit This bit is set to 1 when a LLI Read operation gets Decode Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
8	DST_SLV_ERR_IntStat	R	0	Destination Slave Error Interrupt Status Bit This bit is set to 1 when write/read operation by destination FSM returns with Slave Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).

Table 6-33 CH*x*_INTSTATUSREG *(Continued)*

Bits	Name	R/W	Reset Value	Description
7	SRC_SLV_ERR_IntStat	R	0	Source Slave Error Interrupt Status Bit This bit is set to 1 when Read operation by Source FSM returns with Slave Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
6	DST_DEC_ERR_IntStat	R	0	Destination Decode Error Interrupt Status Bit This bit is set to 1 when write/read operation by destination FSM returns with Decode Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
5	SRC_DEC_ERR_IntStat	R	0	Source Decode Error Interrupt Status Bit This bit is set to 1 when Read operation by Source FSM returns with Decode Error response. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg.
4	DST_TransComp_IntStat	R	0	Destination Transaction Completed Interrupt Status Bit This bit is set when Destination DMA transaction is completed. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
3	SRC_TransComp_IntStat	R	0	Source Transaction Completed Interrupt Status Bit This bit is set when Source DMA transaction is completed. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
2	Undefined	NA	0	Reserved
1	DMA_TFR_DONE_IntStat	R	0	DMA Transfer Done Interrupt Status Bit This bit is set when DMA Transfer is complete for the channel. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).
0	BLOCK_TFR_DONE_ IntStat	R	0	Block Transfer Done Interrupt Status Bit This bit is set when Block transfer is complete for the channel. This bit is cleared to 0 on writing 1 to the corresponding channel interrupt clear bit in CHx_IntClearReg register or on enabling the channel (needed when interrupt is not enabled).

6.2.2.19 CH*x*_INTSIGNAL_ENABLEREG

This register contains fields that are used to enable the generation of port level interrupt at the channel level.

- **Name:** Channel*x* Interrupt Signal Enable Register
- **Size:** 64 bits
- **Offset:** for x = 1 to 8

  CH1_INTSIGNAL_ENABLEREG: 0x190
  CH2_INTSIGNAL_ENABLEREG: 0x290
  CH3_INTSIGNAL_ENABLEREG: 0x390
  CH4_INTSIGNAL_ENABLEREG: 0x490
  CH5_INTSIGNAL_ENABLEREG: 0x590
  CH6_INTSIGNAL_ENABLEREG: 0x690
  CH7_INTSIGNAL_ENABLEREG: 0x790
  CH8_INTSIGNAL_ENABLEREG: 0x890
- **Read/Write Access:** Read/Write

Table 6-34 Fields for Register: CH*x*_INTSIGNAL_ENABLEREG

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as one			
31	Enable_CH_ABORTED_IntSignal	R/W	1	Channel Aborted Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
30	Enable_CH_DISABLED_IntSignal	R/W	1	Channel Disabled Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
29	Enable_CH_SUSPENDED_ IntSignal	R/W	1	Channel Suspended Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
28	Enable_CH_SRC_SUSPENDED_ IntSignal	R/W	1	Channel Source Suspended Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.

Table 6-34 Fields for Register: CH*x*_INTSIGNAL_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
27	Enable_ChLock_Cleared_ IntSignal	R/W	1	Channel Lock Cleared Interrupt Signal Enable Bit Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port-level interrupt.
26:22	Reserved and read as one			
21	Enable_SLVIF_WrOnHold_ ERR_IntSignal	R/W	1	Slave Interface Write On Hold Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
20	Enable_SLVIF_ShadowReg_ WrOn_Valid_ERR_IntSignal	R/W	1	Shadow Register Write On Valid Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
19	Enable_SLVIF_WrOnChEn_ ERR_IntSignal	R/W	1	Slave Interface Write On Channel Enabled Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
18	Enable_SLVIF_RD2RWO_ERR_ IntSignal	R/W	1	Slave Interface Read to write Only Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
17	Enable_SLVIF_WR2RO_ERR_ IntSignal	R/W	1	Slave Interface Write to Read Only Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
16	Enable_SLVIF_DEC_ERR_ IntSignal	R/W	1	Slave Interface Decode Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.

Table 6-34 Fields for Register: CH*x*_INTSIGNAL_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
15	Reserved and read as one			
14	Enable_SLVIF_MultiBlkType_ERR_IntSignal	R/W	1	Slave Interface Multi Block type Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
13	Enable_ShadowReg_OR_LLI_Invalid_ERR_IntSignal	R/W	1	Shadow register or LLI Invalid Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
12	Enable_LLI_WR_SLV_ERR_IntSignal	R/W	1	LLI WRITE Slave Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
11	Enable_LLI_RD_SLV_ERR_IntSignal	R/W	1	LLI Read Slave Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
10	Enable_LLI_WR_DEC_ERR_IntSignal	R/W	1	LLI WRITE Decode Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
9	Enable_LLI_RD_DEC_ERR_IntSignal	R/W	1	LLI Read Decode Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
8	Enable_DST_SLV_ERR_IntSignal	R/W	1	Destination Slave Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.

Table 6-34 Fields for Register: CH*x*_INTSIGNAL_ENABLEREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
7	Enable_SRC_SLV_ERR_IntSignal	R/W	1	Source Slave Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
6	Enable_DST_DEC_ERR_ IntSignal	R/W	1	Destination Decode Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
5	Enable_SRC_DEC_ERR_ IntSignal	R/W	1	Source Decode Error Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
4	Enable_DST_TransComp_ IntSignal	R/W	1	Destination Transaction Completed Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
3	Enable_SRC_TransComp_ IntSignal	R/W	1	Source Transaction Completed Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
2	Reserved and read as one			
1	Enable_DMA_TFR_DONE_ IntSignal	R/W	1	DMA Transfer Done Interrupt Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.
0	Enable_BLOCK_TFR_DONE_ IntSignal	R/W	1	Block Transfer Done Interrupt Signal Enable. Writing 1 to this register field propagates the corresponding interrupt status in the CHx_IntStatusReg register to generate a port level interrupt.

6.2.2.20 CH*x*_INTCLEARREG

Writing 1 to specific field will clear the corresponding field in Channel*x* Interrupt Status Register (CH*x*_IntStatusReg).

- **Name:** Channel*x* Interrupt Clear Register
- **Size:** 64 bits
- **Offset:** 0x198
- **Read/Write Access**: Write

Table 6-35 Fields for Register: CH*x*_INTCLEARREG

Bits	Name	Memory Access	Reset Value	Description
63:32	Reserved and read as zero			
31	Clear_CH_ABORTED_IntStat	W	0	Channel Aborted Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
30	Clear_CH_DISABLED_IntStat	W	0	Channel Disabled Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
29	Clear_CH_SUSPENDED_IntStat	W	0	Channel Suspended Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
28	Clear_CH_SRC_SUSPENDED_IntStat	W	0	Channel Source Suspended Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
27	Clear_ChLock_Cleared_IntStat	W	0	Channel Lock Cleared Interrupt Status Clear Bit Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
26:22	Reserved and read as zero			
21	Clear_SLVIF_WRONHOLD_ERR_IntStat	W	0	Slave Interface Write On Hold Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.

Table 6-35 Fields for Register: CH*x*_INTCLEARREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
20	Clear_SLVIF_SHADOWREG_WRON_ VALID_ERR_IntStat	W	0	Shadow Register Write On Valid Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
19	Clear_SLVIF_WRONCHEN_ERR_IntStat	W	0	Slave Interface Write On Channel Enabled Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
18	Clear_SLVIF_RD2RWO_ERR_IntStat	W	0	Slave Interface Read to write Only Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
17	Clear_SLVIF_WR2RO_ERR_IntStat	W	0	Slave Interface Write to Read Only Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
16	Clear_SLVIF_DEC_ERR_IntStat	W	0	Slave Interface Decode Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
15	Reserved and read as zero			
14	Clear_SLVIF_MULTIBLKTYPE_ERR_ IntStat	W	0	Slave Interface Multi Block type Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
13	Clear_SHADOWREG_OR_LLI_INVALID_ ERR_IntStat	W	0	Shadow register or LLI Invalid Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
12	Clear_LLI_WR_SLV_ERR_IntStat	W	0	LLI WRITE Slave Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.

Table 6-35 Fields for Register: CH*x*_INTCLEARREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
11	Clear_LLI_RD_SLV_ERR_IntStat	W	0	LLI Read Slave Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
10	Clear_LLI_WR_DEC_ERR_IntStat	W	0	LLI WRITE Decode Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
9	Clear_LLI_RD_DEC_ERR_IntStat	W	0	LLI Read Decode Error Interrupt Clear Bit. This bit is used to clear the corresponding channel interrupt status bit in CH*x*_INTSTATUSREG.
8	Clear_DST_SLV_ERR_IntStat	W	0	Destination Slave Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
7	Clear_SRC_SLV_ERR_IntStat	W	0	Source Slave Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
6	Clear_DST_DEC_ERR_IntStat	W	0	Destination Decode Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
5	Clear_SRC_DEC_ERR_IntStat	W	0	Source Decode Error Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
4	Clear_DST_TRANSCOMP_IntStat	W	0	Destination Transaction Completed Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
3	Clear_SRC_TRANSCOMP_IntStat	W	0	Source Transaction Completed Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
2	Reserved and read as zero			

Table 6-35 Fields for Register: CH*x*_INTCLEARREG (Continued)

Bits	Name	Memory Access	Reset Value	Description
1	Clear_DMA_TFR_DONE_IntStat	W	0	DMA Transfer Done Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.
0	Clear_BLOCK_TFR_DONE	W	0	Block Transfer Done Interrupt Clear Bit. Writing a 1 to this register field clears the corresponding bit in the CHx_IntStatusReg register.

7 Programming the DW_axi_dmac

The DW_axi_dmac can be programmed through software registers or the DW_axi_dmac low-level software driver; software registers are described in more detail in "Registers" on page 123.

7.1 Programming Flow for Shadow-Register-Based Multi-Block Transfer

1. Software reads the DMAC channel enable register (DMAC_ChEnReg) to select an available (unused) channel.
2. Software programs the CHx_CFG register with appropriate values for the DMA transfer.

   The SRC_MLTBLK_TYPE and/or DST_MLTBLK_TYPE bits must be set to 2'b10.

Note

- The CHx_CFG register must be programmed before programming the CHx_SAR, CHx_DAR, CHx_BLOCK_TS, or CHx_CTL registers, as the value of the SRC_MLTBLK_TYPE and/or DST_MLTBLK_TYPE fields are used for accessing the shadow registers.
- If the slave interface data bus width or transfer size is less than 64 bits, CHx_CFG[7:0] should be updated in the first write to the CHx_CFG register.

3. Software programs the CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers with appropriate values for the first block.

   DW_axi_dmac loads the corresponding shadow registers with these values.

   The CHx_CTL register must be the last register to be programmed with the ShadowReg_Or_LLI_Valid bit set to 1 to indicate that the shadow register contents are valid. If the slave interface data bus width or transfer size is less than 64 bits, CHx_CTL[63:56] must be updated last.
4. Software enables the channel by writing 1 to the appropriate bit location in the DMAC_ChEnReg register.

Note

It is possible to swap the sequence of step 3 and step 4. However, if step 4 is performed before step 3, DW_axi_dmac might generate a ShadowReg_Or_LLI_Invalid_ERR interrupt if the value of the ShadowReg_Or_LLI_Valid bit is 0 during the shadow register fetch phase.

5. DW_axi_dmac initiates the DMA block transfer operation based on the settings for the block transfer.
    a. The block transfer might start immediately or after the hardware or software handshaking request, depending on the value of the TT_FC field in the CHx_CFG register.
    b. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as '0', DW_axi_dmac waits till software writes (any value) to CHx_BLK_TFR_ResumeReqReg to indicate valid LLI availability, before attempting another Shadow Register fetch operation. DW_axi_dmac might generate 'ShadowReg_Or_LLI_Invalid_ERR' Interrupt in this case.
    c. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as ‘1,’ DW_axi_dmac copies the shadow register contents to the registers used for executing the DMA block transfer (CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS and CHx_CTL registers) and clears the ShadowReg_Or_LLI_Valid bit in CHx_CTL and CHx_CTL_ShadowReg registers to 0.
        i. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register as 1, it understands that the current block is the final block in the transfer and completes the DMA transfer operation at the end of current block transfer.
        ii. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register as 0, it understands that there are one or more blocks to be transferred and checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit again at the end of current block transfer.
6. Software polls the ShadowReg_Or_LLI_Valid bit in the CHx_CTL register till it is 0.
    a. DW_axi_dmac clears this bit to 0 only after copying the shadow register contents to the registers used for executing the DMA block transfer (that is, the CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers).
    b. Software must program the shadow registers with a new set of values only after the ShadowReg_Or_LLI_Valid bit is set to 0.
    c. If software tries to programs the shadow registers when the ShadowReg_Or_LLI_Valid bit is set to 1, DW_axi_dmac ignores this write operation, sets the SLVIF_ShadowReg_WrOnValid_ERR bit of the CHx_IntStatusReg register to 1, and generates an interrupt (if the corresponding interrupt generation is not masked off).
7. Software programs the CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers with appropriate values for the next block.
    a. The CHx_CTL register must be the last register to be programmed with the ShadowReg_Or_LLI_Valid bit set to 1 to indicate that the shadow register contents are valid.
    b. If current block is the final block in the transfer, S/W must set CHx_CTL.ShadowReg_Or_LLI_Last bit to ‘1.’
    c. The DMA block transfer corresponding to the previous shadow register contents may be in progress during this time.
    d. DW_axi_dmac loads the corresponding shadow registers with these new values.
8. DW_axi_dmac initiates the DMA block transfer operation based on the settings for the block transfer.
    a. Based on the settings of TT_FC field in CHx_CFG register, the block transfer might start immediately or after the hardware/software handshaking request.

b. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as '0,' DW_axi_dmac waits until software writes (any value) to CHx_BLK_TFR_ResumeReqReg to indicate valid LLI availability, before attempting another Shadow Register fetch operation. DW_axi_dmac might generate ShadowReg_Or_LLI_Invalid_ERR Interrupt in this case.

c. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as '1,' DW_axi_dmac copies the shadow register contents to the registers used for executing the DMA block transfer (CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS and CHx_CTL registers) and clears the ShadowReg_Or_LLI_Valid bit in CHx_CTL and CHx_CTL_ShadowReg registers to 0.

d. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register as 1, it understands that the current block is the final block in the transfer and completes the DMA transfer operation at the end of current block transfer.

e. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register as 0, it understands that there are one or more blocks to be transferred and checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit again at the end of current block transfer.

9. Software waits for the block transfer completion interrupt or polls the block transfer completion indication bit (BLOCK_TFR_DONE) of the CHx_IntStatusReg register until it is set to 1.

10. On block transfer completion:

    a. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as '0,' DW_axi_dmac waits until software writes (any value) to CHx_BLK_TFR_ResumeReqReg to indicate valid LLI availability, before attempting another Shadow Register fetch operation. DW_axi_dmac might generate a ShadowReg_Or_LLI_Invalid_ERR Interrupt in this case.

    b. DW_axi_dmac checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit and if it is seen as '1', DW_axi_dmac copies the shadow register contents to the registers used for executing the DMA block transfer (CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS and CHx_CTL registers) and clears ShadowReg_Or_LLI_Valid bit in CHx_CTL and CHx_CTL_ShadowReg registers to 0.

        - If CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register is 1, it understands that the current block is the final block in the transfer and completes the DMA transfer operation.
        - If CHx_CTL.ShadowReg_Or_LLI_Last bit of the copied Shadow Register is 0, it understands that there are one or more blocks to be transferred and checks CHx_CTL_ShadowReg.ShadowReg_Or_LLI_Valid bit again at the end of current block transfer.

    c. If there are one or more blocks to be transferred, software polls CHx_CTL.ShadowReg_Or_LLI_Valid bit until it is seen as 0 and go to step 7.

       One read operation is enough as DW_axi_dmac should have already copied the shadow register contents and cleared this bit to 0.

Note In case when 'ShadowReg_Or_LLI_Invalid_ERR' is generated the recommended flow to resume transfer is:

- Software programs CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS and CHx_CTL registers with appropriate values for the next block.
- Clear the interrupt using interrupt register CHx_IntClearReg.
- Program block resume request CHx_BLK_TFR_ResumeReqReg.

7.2 Programming Flow for Linked-List-Based Multi-Bock Transfer

1. Software reads the DMAC channel enable register (DMAC_ChEnReg) to select an available (unused) channel.
2. Software programs the CHx_CFG register with appropriate values for the DMA transfer.

   The SRC_MLTBLK_TYPE and/or DST_MLTBLK_TYPE bits must be set to 2'b11.
3. Software programs the base address of the first linked list item and the master interface on which the linked list item is available in the CHx_LLP register.
4. Software creates one or more linked list items in system memory. Software can create the entire linked list item in advance or dynamically extend the linked list using the CHx_CTL.ShadowReg_Or_LLI_Valid and CHx_CTL.LLI_Last fields of the LLI.
5. Software enables the channel by writing 1 to the appropriate bit location in DMAC_ChEnReg register.

Note It is possible to swap the sequence of step 4 and step 5. However, if step 5 is performed before step 4, or if the linked list item for the next block transfer is not available in system memory at any time during the multiblock transfer, as indicated by CHx_CTL.ShadowReg_Or_LLI_Valid bit of the fetched LLI being set to 0, DW_axi_dmac might generate a ShadowReg_Or_LLI_Invalid_ERR interrupt.

6. DW_axi_dmac initiates the DMA block transfer operation based on the settings for the block transfer.

   The block transfer might start immediately or after the hardware or software handshaking request, depending on the settings of the TT_FC field in the CHx_CFG register.

   DW_axi_dmac copies the linked list contents to the registers used for executing the DMA block transfer (that is, the CHx_SAR and/or CHx_DAR, CHx_BLOCK_TS, and CHx_CTL registers) and initiates the DMA block transfer.
7. During the linked list fetch phase:
   a. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the fetched LLI as 0, it understands that the current block is the final block in the transfer and completes the DMA transfer operation at the end of current block transfer.
   b. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Last bit of the fetched LLI as 1, it understands that there are one or more blocks to be transferred and goes to step 6.
   c. If DW_axi_dmac sees CHx_CTL.ShadowReg_Or_LLI_Valid bit of the fetched LLI as 0, DW_axi_dmac might generate 'ShadowReg_Or_LLI_Invalid_ERR' Interrupt. DW_axi_dmac waits till software writes (any value) to CHx_BLK_TFR_ResumeReqReg to indicate valid LLI availability, before attempting another LLI read operation.

8 Verification

This chapter provides an overview of the testbench available for DW_axi_dmac verification. Once the DW_axi_dmac has been configured and the verification environment set up, simulations can be run automatically. For information on running simulations for DW_axi_dmac in coreAssembler or coreConsultant, see "Building and Verifying a Component or Subsystem" on page 25.

Note The DW_axi_dmac verification testbench is built with DesignWare Verification IP (VIP). Please make sure 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*.

8.1 Overview of SV-UVM Tests

The DW_axi_dmac verification testbench (SV-UVM) performs the following tests that have been written to exhaustively verify the functionality:

- A single block transfer
- An LLP block chaining transfer (static and dynamic LLP)
- A reloading transfer
- A shadow transfer

Within each of these transfers, all parameters are randomized. The testbench constantly verifies whether conditions including the following are met:

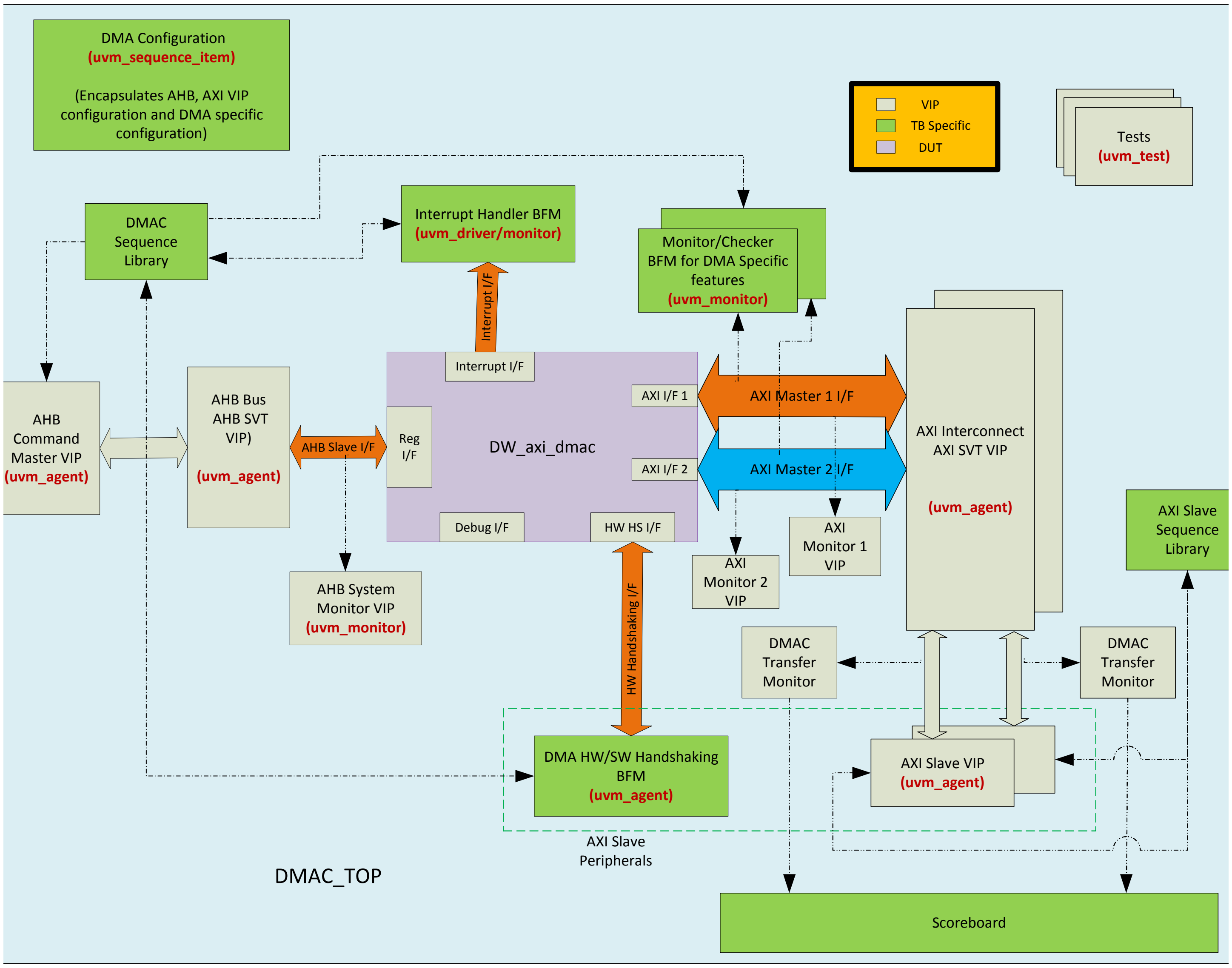
- Transfers are correct
- Registers are updated correctly
- Registers are written out correctly
- Interrupts are set correctly
- Flow control mode is not violated
- Bus and channel locking is correct
- Channel arbitration is correct

8.2 Overview of DW_axi_dmac Testbench

As illustrated in Figure 8-1, the DW_axi_dmac testbench is an SV-UVM testbench that includes:

- Verilog DUT (DW_axi_dmac)
- SV-UVM BFMs (Interrupt handler, HW/SW handshake, Transfer Monitors)
- VIPs (AHB Master, bus, monitor and AXI Slaves, interconnect, and monitors)

Figure 8-1 Verification Testbench Block Diagram



The file, dmac_top.sv, 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_axi_dmac. The “<workspace>/sim/<test_name>” directory contains the test_name.sv file.

9 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.

9.1 Performance

This section discusses the performance of DW_axi_dmac, and the software and hardware configuration parameters that affect the performance of the DW_axi_dmac.

9.1.1 Area

This section provides information to help you configure the area for your configuration.

9.1.1.1 Synthesis Results

Tables 9-1 includes synthesis results that have been generated using the tsmc28nm technology library for configurations where DMAX_CHx_MULTI_BLK_TYPE is set to 0.

Table 9-1 Area Results for DW_axi_dmac with tsmc28nm and DMAX_CHx_MULTI_BLK_TYPE set to 0

Configuration	Operating Frequency (in MHz)				Gate Count
	Core Clock	Master 1 Clock	Master 2 Clock	Slave Clock	
Number of masters = 1 Number of channels = 1 FIFO depth of each channel = 8 Slave clock synchronous to core Master clock synchronous to core (Default configuration)	200	-	-	-	21600 gates
Number of masters = 2 Number of channels = 4 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock asynchronous to core	200	400	250	100	93921 gates

Table 9-1 Area Results for DW_axi_dmac with tsmc28nm and DMAX_CHx_MULTI_BLK_TYPE set to 0

Configuration	Operating Frequency (in MHz)				Gate Count
	Core Clock	Master 1 Clock	Master 2 Clock	Slave Clock	
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock synchronous to core	200	-	-	100	272719 gates
Number of masters = 1 Number of channels = 4 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	76478 gates
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	164751 gates

Tables 9-2 includes synthesis results that have been generated using the tsmc28nm technology library for configuration in which DMAX_CHx_MULTI_BLK_TYPE is set to 1.

Table 9-2 Area Results for DW_axi_dmac with tsmc28nm and DMAX_CHx_MULTI_BLK_TYPE set to 1

Configuration	Operating Frequency (in MHz)				Gate Count
	Core Clock	Master 1 Clock	Master 2 Clock	Slave Clock	
Number of masters = 1 Number channels = 1 FIFO depth of each channel = 8 Slave clock synchronous to core Master clock synchronous to core (Default configuration)	200	-	-	-	24531gates
Number of masters = 2 Number of channels = 4 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock asynchronous to core	200	400	250	100	106633 gates
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock synchronous to core	200	-	-	100	297402 gates
Number of masters = 1 Number of channels = 4 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	795450 gates
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	188955 gates

9.1.2 Power Consumption

Tables 9-3 provides information about the power consumption of the DW_axi_dmac using the tsmc28nm technology library configurations where DMAX_CHx_MULTI_BLK_TYPE is set to 0.

Table 9-3 Power Results for DW_axi_dmac with tsmc28nm and DMAX_CHx_MULTI_BLK_TYPE set to 0

Configuration	Operating Frequency (in MHz)				Leakage Power consumption (mW)	Dynamic Power consumption (mW)
	Core Clock	Master 1 Clock	Master 2 Clock	Slave Clock		
Number of master = 1 Number channel = 1 FIFO depth of each channel = 8 Slave clock synchronous to core Master clock synchronous to core (Default configuration)	200	-	-	-	2.1765	1.9192
Number of masters = 2 Number of channels = 4 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock asynchronous to core	200	400	250	100	9.3262	8.7559
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock synchronous to core	200	-	-	100	26.1822	27.4017
Number of masters = 1 Number of channels = 4 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	7.6294	6.9414
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	16.5025	15.0463

Tables 9-4 provides information about the power consumption of the DW_axi_dmac using the tsmc28nm technology library configurations where DMAX_CHx_MULTI_BLK_TYPE is set to 1.

Table 9-4 Power Results for DW_axi_dmac with tsmc28nm and DMAX_CHx_MULTI_BLK_TYPE set to 0

Configuration	Operating Frequency (in MHz)				Leakage Power consumption (mW)	Dynamic Power consumption (mW)
	Core Clock	Master 1 Clock	Master 2 Clock	Slave Clock		
Number of masters = 1 Number channels = 1 FIFO depth of each channel = 8 Slave clock synchronous to core Master clock synchronous to core (Default configuration)	200	-	-	-	2.4654	2.2059
Number of masters = 2 Number of channels = 4 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock asynchronous to core	200	400	250	100	10.5294	9.9608
Number of masters = 1 Number of channels =8 FIFO depth of each channel = 64 Slave clock asynchronous to core Master clock synchronous to core	200	-	-	100	28.5589	29.7008
Number of masters = 1 Number of channels = 4 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	8.7763	8.0870
Number of masters = 1 Number of channels = 8 FIFO depth of each channel = 16 Slave clock synchronous to core Master clock asynchronous to core	200	400	-	-	18.8086	17.3357

9.2 4K Boundary Crossing

The AXI protocol requires that any AXI burst does not cross a 4KB address boundary. DW_axi_dmac handles this situation automatically. If a DMA transfer is set up by software such that during the transfer, an AXI transfer crosses a 4KB boundary, DW_axi_dmac will automatically set up the AXI transfers such that the end of the 4KB boundary completes one AXI transfer and the beginning of the 4KB boundary starts another AXI transfer.

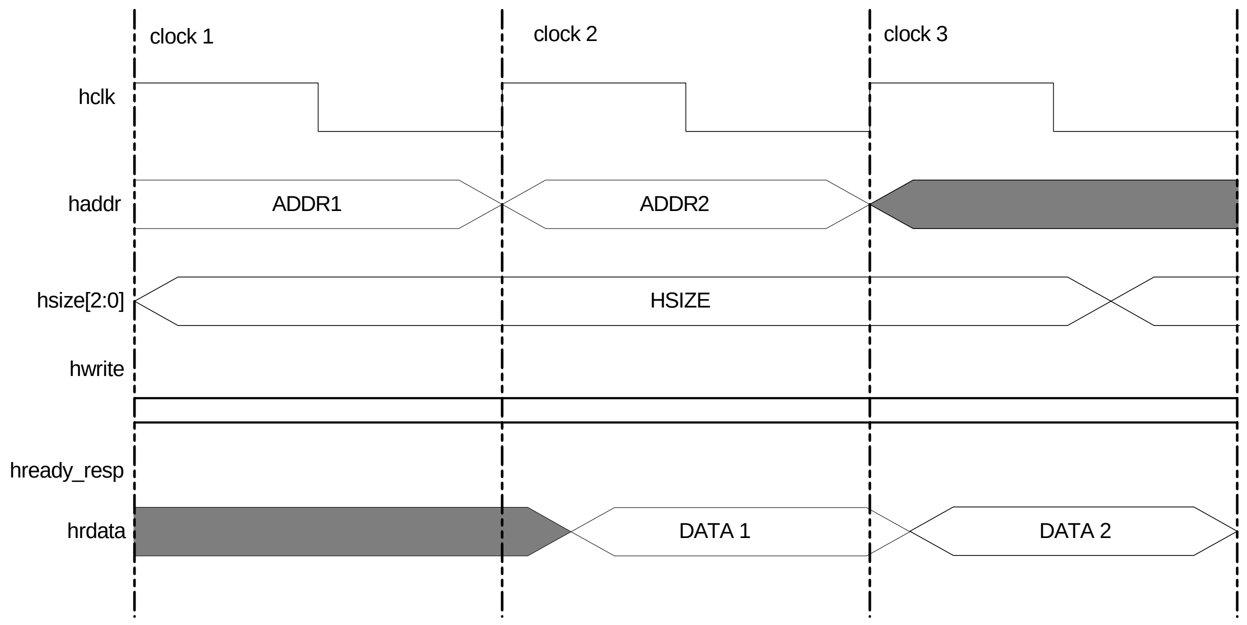
9.3 Read Accesses

For reads, registers less than the full access width return zeros in the unused upper bits. All registers in DW_axi_dmac are READ and WRITE in DW_axi_dmac core clock domain. Therefore, the time taken for the reads or writes to complete depends on the slave interface clock mode (parameter DMAX_SLVIF_CLOCK_MODE).

9.3.1 Slave Interface Clock is Synchronous to the DW_axi_dmac Core Clock

When a slave interface clock is synchronous to the core clock, synchronization is not required. In this case, an AHB read takes two hclk cycles. The two cycles can be a control and data cycle, respectively. As shown in Figure 9-1, 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.

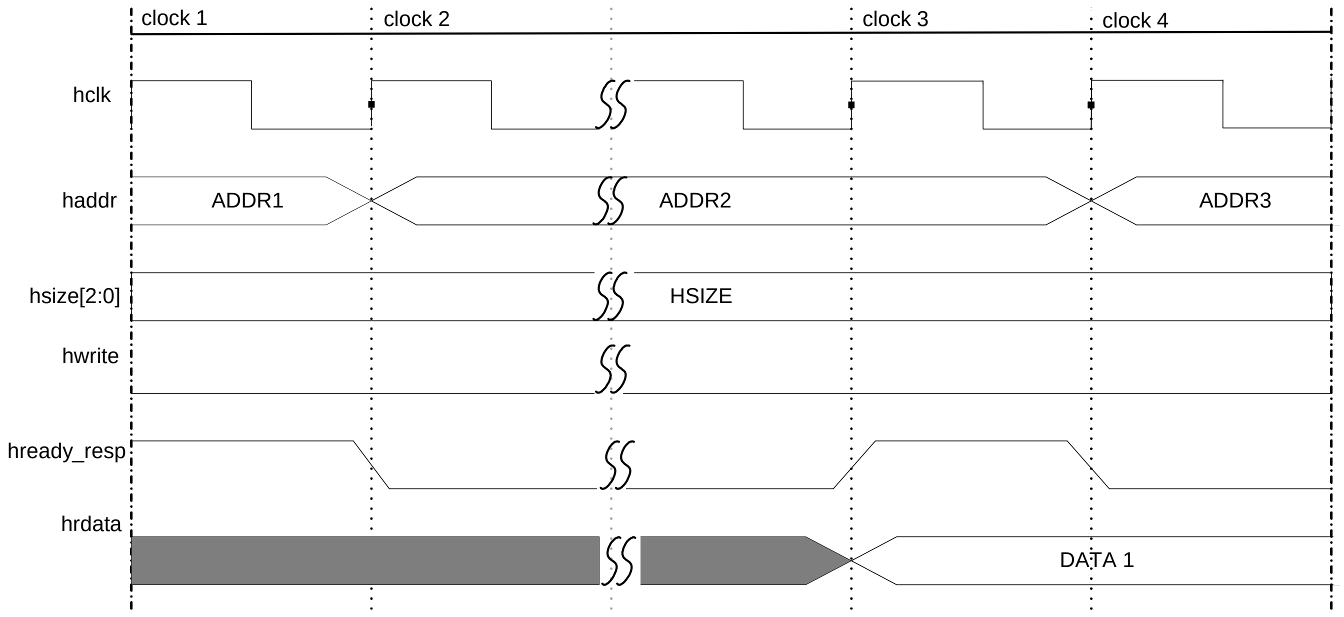
Figure 9-1 AHB Read When Slave Clock is Synchronous to DW_axi_dmac Core Clock



9.3.2 Slave Interface Clock is Asynchronous to DW_axi_dmac Core Clock

When a slave interface clock is asynchronous to DW_axi_dmac core clock, synchronization occurs in between each read operation. Once the read request is made on the slave interface, it is synchronized in DW_axi_dmac core clock domain and then response from register interface is synchronized back in the slave clock domain. Therefore, the data is driven on the slave interface only after the synchronization delay from slave interface clock to DW_axi_dmac core clock and the synchronization delay from DW_axi_dmac core clock to slave interface clock. Until then, hready_resp is driven low on AHB interface. As shown in Figure 9-2, the address and control for the first read is driven on clock 1; on clock 2 controls for next read is driven on the bus but hready_resp is pulled down by DW_axi_dmac. After synchronization delay on clock 3, data for first read is driven on the bus by DW_axi_dmac and in similar manner reads following that happens.

Figure 9-2 AHB Read When Slave Clock is Asynchronous to DW_axi_dmac Core Clock



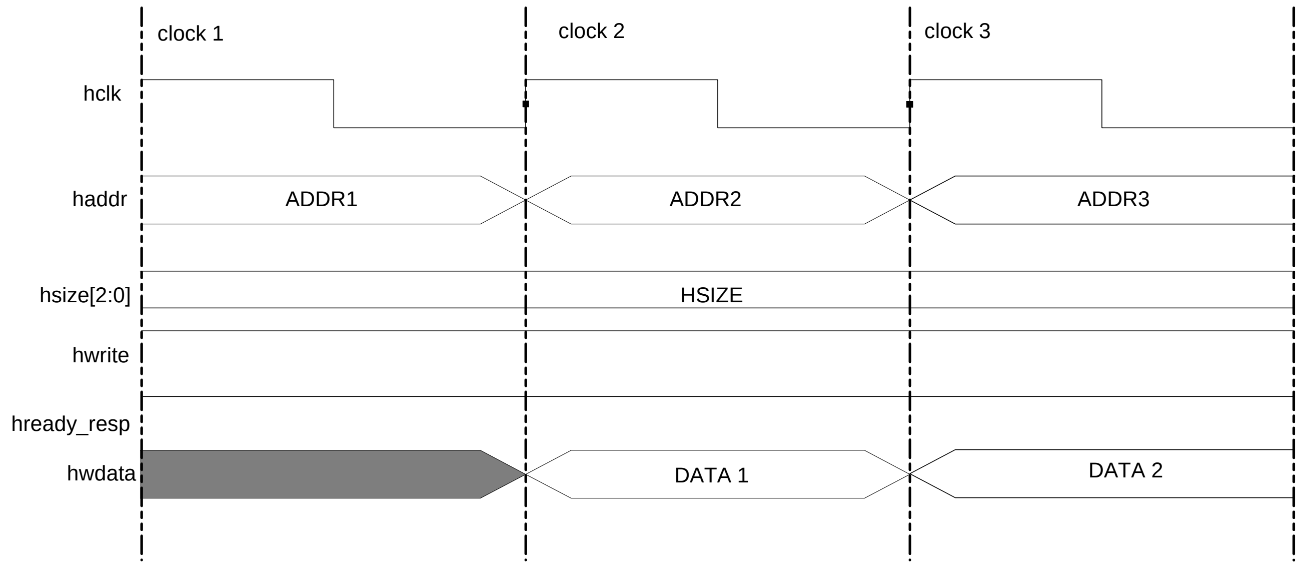
9.4 Write Accesses

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 read access, write access time is also dependent on the slave interface clock mode (parameter DMAX_SLVIF_CLOCK_MODE).

9.4.1 Slave Interface Clock is Synchronous to DW_axi_dmac Core Clock

Similar to read, a write access may be thought of as comprising a control and data cycle. As illustrated Figure 9-3, the address and control are 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.

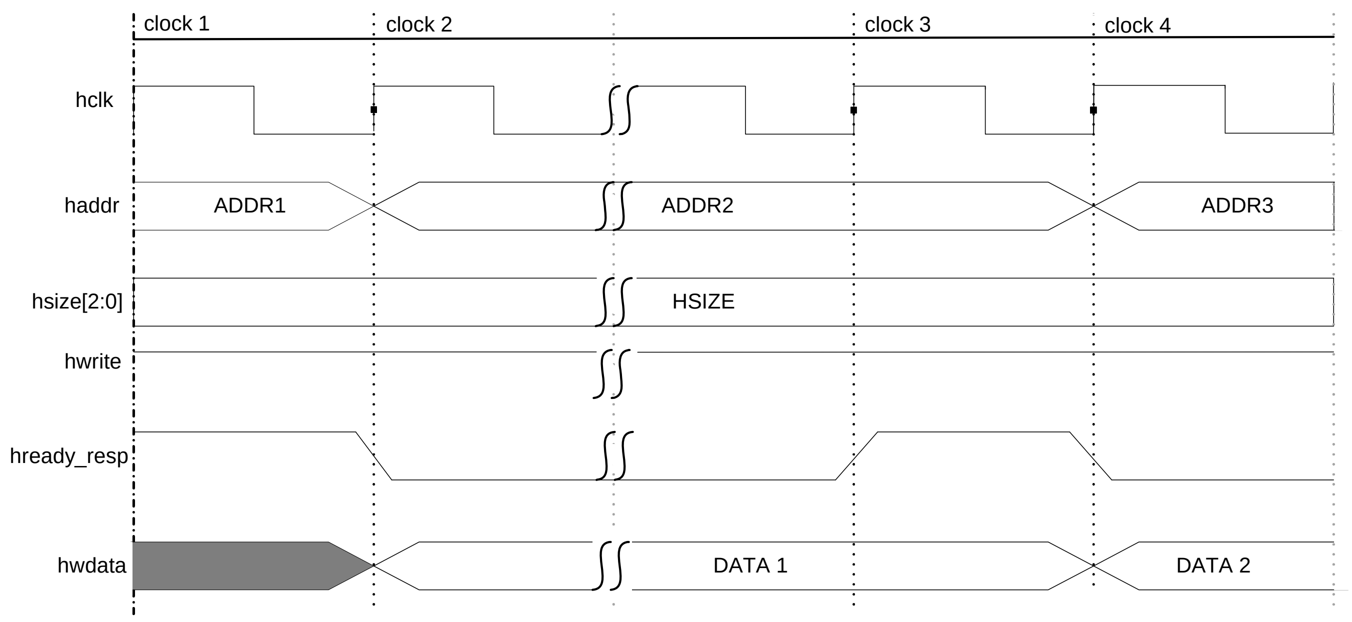
Figure 9-3 AHB Write When Slave Clock is Synchronous to DW_axi_dmac Core Clock



9.4.2 Slave Interface Clock is Asynchronous to DW_axi_dmac Core Clock

Similar to read, in this case also write requests are first synchronized in DW_axi_dmac core clock domain and then response is again synchronized to slave clock domain. Therefore, acceptance of next write and response only happens after synchronization delay from slave interface clock to DW_axi_dmac core clock and after synchronization delay from DW_axi_dmac core clock to slave interface clock. Until then, hready_resp signal is driven low by DW_axi_dmac. As shown in Figure 9-4, control is driven by AHB master on clock 1 and data along with control information for second write is driven on clock 2; at clock 2 DW_axi_dmac pulls down hready_resp to 0 indicating that current data and control has not yet been accepted. After synchronization delay on clock 3, hready_resp is asserted indicating that current data has been written. At clock 4, AHB master samples hread_resp and drives data from previous address and next control information.

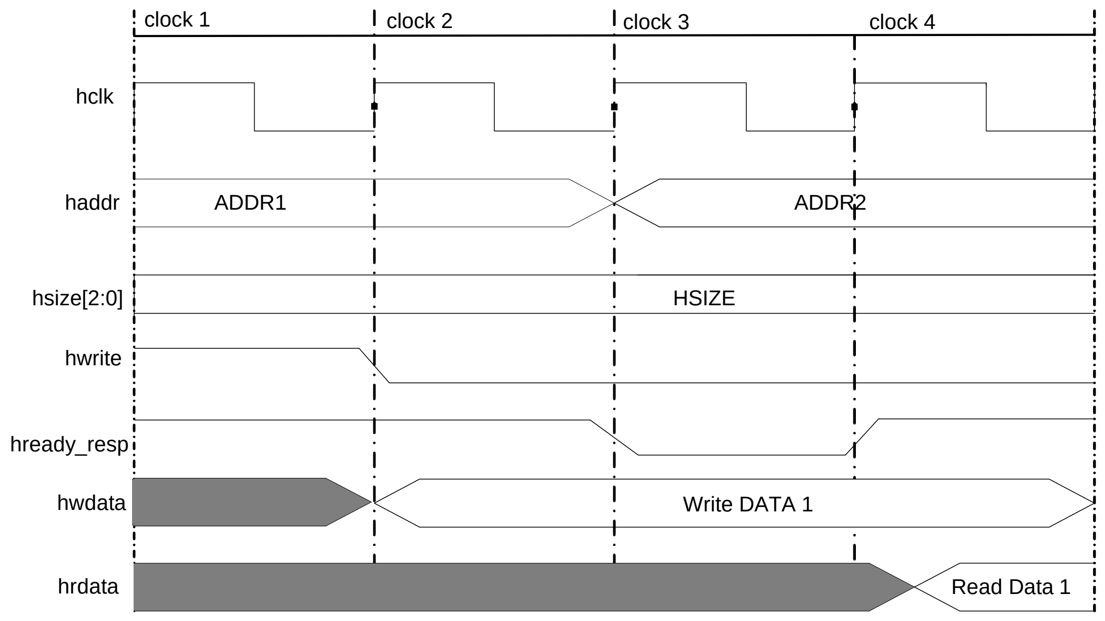
Figure 9-4 AHB Write When Slave Clock is Asynchronous to DW_axi_dmac Core Clock



9.5 Consecutive Write-Read

This is a specific case for the AHB slave interface (only applicable when slave interface clock is synchronous to DW_axi_dmac core clock). The AMBA specification mentions that for a read after a write to the same address, the newly written data must be read back, and not the old data. To comply with this, the slave interface in the DW_axi_dmac inserts a "wait state" when it detects a read immediately after a write to the same address. As shown in Figure 9-5, 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 9-5 AHB Wait State Read/Write



9.6 Accessing Top-Level Constraints

To get SDC constraints from coreConsultant (cC), 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>
```

A Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
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.
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.
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.
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.
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.
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.
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