PrimeCell® DMA Controller (PL330)

Revision: r0p0

Technical Reference Manual

ARM®

ARM DDI 0424A

PrimeCell DMA Controller (PL330)
Technical Reference Manual

Release Information

The *Change history* table lists the changes made to this manual.

Change history

Date	Issue	Confidentiality	Change
19 December 2007	A	Non-Confidential	First issue for the r0p0 release

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents

PrimeCell DMA Controller (PL330) Technical Reference Manual

Preface

About this manual x
Feedback xv

Chapter 1 Introduction

1.1 About the DMAC 1-2
1.2 Terminology 1-5

Chapter 2 Functional Overview

2.1 Overview 2-2
2.2 DMAC interfaces 2-4
2.3 Operating states 2-10
2.4 Initializing the DMAC 2-14
2.5 Using the APB slave interfaces 2-16
2.6 Peripheral request interface 2-18
2.7 Using events and interrupts 2-23
2.8 Aborts 2-25
2.9 Security usage 2-29
2.10 Constraints and limitations of use 2-34
2.11 Programming restrictions 2-35

Chapter 3 **Programmers Model**
3.1 About the programmers model 3-2
3.2 DMAC Register summary 3-4
3.3 DMAC Register descriptions 3-11

Chapter 4 **Instruction Set**
4.1 Instruction syntax conventions 4-2
4.2 Instruction set summary 4-3
4.3 Instructions 4-5
4.4 Assembler directives 4-25

Appendix A **Signal Descriptions**
A.1 Clocks and resets A-2
A.2 AXI signals A-3
A.3 APB signals A-7
A.4 Peripheral request interface A-9
A.5 Interrupt signals A-11
A.6 Tie-off signals A-12

Glossary

List of Tables
PrimeCell DMA Controller (PL330) Technical Reference Manual

	Change history	ii
Table 2-1	AXI characteristics for a DMA transfer	2-8
Table 2-2	Handshake rules	2-19
Table 3-1	DMAC Control Register summary	3-4
Table 3-2	DMA channel thread status register summary	3-6
Table 3-3	AXI status and loop counter register summary	3-7
Table 3-4	DMAC Debug Register summary	3-9
Table 3-5	DMAC Configuration Register summary	3-9
Table 3-6	Peripheral and PrimeCell Identification Register summary	3-10
Table 3-7	DS Register bit assignments	3-11
Table 3-8	DPC Register bit assignments	3-13
Table 3-9	INTEN Register bit assignments	3-14
Table 3-10	ES Register bit assignments	3-15
Table 3-11	INTSTATUS Register bit assignments	3-16
Table 3-12	INTCLR Register bit assignments	3-17
Table 3-13	FSM Register bit assignments	3-18
Table 3-14	FSC Register bit assignments	3-19
Table 3-15	FTM Register bit assignments	3-20
Table 3-16	FTC Register bit assignments	3-22
Table 3-17	CSn Register bit assignments	3-25
Table 3-18	CPCn Register bit assignments	3-27

Table 3-19 SA_n Register bit assignments 3-28
Table 3-20 DA_n Register bit assignments 3-29
Table 3-21 CC_n Register bit assignments 3-31
Table 3-22 Swap data 3-35
Table 3-23 LC0_n Register bit assignments 3-36
Table 3-24 LC1_n Register bit assignments 3-37
Table 3-25 DBGSTATUS Register bit assignments 3-37
Table 3-26 DBGCMD Register bit assignments 3-38
Table 3-27 DBGINST0 Register bit assignments 3-39
Table 3-28 DBGINST1 Register bit assignments 3-40
Table 3-29 CR0 Register bit assignments 3-41
Table 3-30 CR1 Registers bit assignments 3-43
Table 3-31 CR2 Register bit assignments 3-44
Table 3-32 CR3 Register bit assignments 3-44
Table 3-33 CR4 Register bit assignments 3-45
Table 3-34 CRDn Registers bit assignments 3-46
Table 3-35 periph_id Register bit assignments 3-48
Table 3-36 periph_id_0 Register bit assignments 3-49
Table 3-37 periph_id_1 Register bit assignments 3-49
Table 3-38 periph_id_2 Register bit assignments 3-49
Table 3-39 periph_id_3 Register bit assignments 3-50
Table 3-40 pcell_id Register bit assignments 3-50
Table 3-41 pcell_id_0 Register bit assignments 3-51
Table 3-42 pcell_id_1 Register bit assignments 3-52
Table 3-43 pcell_id_2 Register bit assignments 3-52
Table 3-44 pcell_id_3 Register bit assignments 3-52
Table 4-1 Instruction syntax summary 4-3
Table 4-2 DMAMOV CCR argument description and the default values 4-27
Table A-1 Clock and reset A-2
Table A-2 AXI-AW signals A-3
Table A-3 AXI-W signals A-4
Table A-4 AXI-B signals A-4
Table A-5 AXI-AR signals A-5
Table A-6 AXI-R signals A-5
Table A-7 Non-secure APB interface signals A-7
Table A-8 Secure APB interface signals A-7
Table A-9 Peripheral request interface A-9
Table A-10 Interrupt signals A-11
Table A-11 DMAC tie-off signals A-12
Table A-12 Interrupt and peripheral tie-off signals A-13

List of Figures
PrimeCell DMA Controller (PL330) Technical Reference Manual

	Key to timing diagram conventions	xii
Figure 1-1	Interfaces on the DMAC	1-2
Figure 1-2	Example system	1-3
Figure 2-1	DMAC block diagram	2-2
Figure 2-2	APB slave interfaces	2-4
Figure 2-3	AXI master interface connections	2-6
Figure 2-4	Peripheral request interface	2-9
Figure 2-5	Interrupt interface	2-9
Figure 2-6	Reset initialization interface	2-9
Figure 2-7	Thread operating states	2-10
Figure 2-8	Request and acknowledge buses on the peripheral request interface	2-18
Figure 2-9	Burst request signaling	2-20
Figure 2-10	Single and burst request signaling	2-21
Figure 2-11	Single transfers for a burst request	2-22
Figure 2-12	Abort process	2-27
Figure 2-13	Equation for the maximum number of consecutive DMALD	2-40
Figure 2-14	DMAC operating with four DMA channels	2-41
Figure 2-15	Equation for the maximum number of consecutive DMAST	2-41
Figure 3-1	DMAC summary register map	3-3
Figure 3-2	DS Register bit assignments	3-11
Figure 3-3	DPC Register bit assignments	3-13

Figure 3-4 INTEN Register bit assignments 3-13
Figure 3-5 ES Register bit assignments 3-14
Figure 3-6 INTSTATUS Register bit assignments 3-16
Figure 3-7 INTCLR Register bit assignments 3-17
Figure 3-8 FSM Register bit assignments 3-18
Figure 3-9 FSC Register bit assignments 3-19
Figure 3-10 FTM Register bit assignments 3-20
Figure 3-11 FTC Register bit assignments 3-22
Figure 3-12 CSn Register bit assignments and address offsets 3-24
Figure 3-13 CPCn Register bit assignments and address offsets 3-27
Figure 3-14 SA_n Register bit assignments and address offsets 3-28
Figure 3-15 DA_n Register bit assignments and address offsets 3-29
Figure 3-16 CC_n Register bit assignments 3-30
Figure 3-17 LC0_n Register bit assignments 3-36
Figure 3-18 LC1_n Register bit assignments 3-36
Figure 3-19 DBGSTATUS Register bit assignments 3-37
Figure 3-20 DBGCMD Register bit assignments 3-38
Figure 3-21 DBGINST0 Register bit assignments 3-38
Figure 3-22 DBGINST1 Register bit assignments 3-39
Figure 3-23 CR0 Register bit assignments 3-40
Figure 3-24 CR1 Registers bit assignments 3-42
Figure 3-25 CR2 Registers bit assignments 3-43
Figure 3-26 CR3 Registers bit assignments 3-44
Figure 3-27 CR4 Registers bit assignments 3-45
Figure 3-28 CRDn Register bit assignments 3-46
Figure 3-29 periph_id Register bit assignments 3-48
Figure 3-30 pcell_id Register bit assignments 3-51
Figure 4-1 DMAADDH encoding 4-5
Figure 4-2 DMAEND encoding 4-6
Figure 4-3 DMAFLUSHP encoding 4-6
Figure 4-4 DMAGO encoding 4-7
Figure 4-5 DMALD[S|B] encoding 4-8
Figure 4-6 DMALDP<S|B> encoding 4-10
Figure 4-7 DMALP encoding 4-11
Figure 4-8 DMALPEND[S|B] encoding 4-12
Figure 4-9 DMAKILL encoding 4-14
Figure 4-10 DMAMOV encoding 4-15
Figure 4-11 DMANOP encoding 4-17
Figure 4-12 DMARMB encoding 4-17
Figure 4-13 DMASEV encoding 4-18
Figure 4-14 DMAST[S|B] encoding 4-19
Figure 4-15 DMASTP<S|B> encoding 4-20
Figure 4-16 DMASTZ encoding 4-21
Figure 4-17 DMAWFE encoding 4-22
Figure 4-18 DMAWFP<S|B|P> encoding 4-22
Figure 4-19 DMAWMB encoding 4-23

Preface

This preface introduces the *PrimeCell DMA Controller (PL330) Technical Reference Manual*. It contains the following sections:

- *About this manual* on page x
- *Feedback* on page xv.

About this manual

This is the *Technical Reference Manual* (TRM) for the *PrimeCell DMA Controller (PL330)*.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, where:

r*n*	Identifies the major revision of the product.
p*n*	Identifies the minor revision or modification status of the product.

Intended audience

This manual is written for system designers, system integrators, and programmers who are designing or programming a *System-on-Chip* (SoC) device that uses the *DMA Controller* (DMAC).

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for a high-level view of the DMAC.

Chapter 2 *Functional Overview*

Read this chapter for a description of the major interfaces and components of the DMAC. The chapter also describes how they operate.

Chapter 3 *Programmers Model*

Read this chapter for a description of the DMAC memory map and registers.

Chapter 4 *Instruction Set*

Read this chapter for a description of the instruction set.

Appendix A *Signal Descriptions*

Read this appendix for a description of the DMAC input and output signals.

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams*
- *Signals* on page xii
- *Numbering* on page xiii.

Typographical

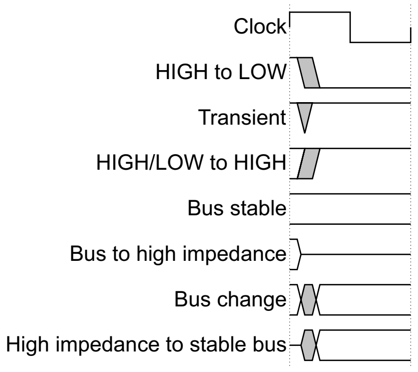
The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` (mono underlined)	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Timing diagrams

The figure named *Key to timing diagram conventions* on page xii explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means: • HIGH for active-HIGH signals • LOW for active-LOW signals.
Lower-case n	At the start or end of a signal name denotes an active-LOW signal.
Prefix A	Denotes global *Advanced eXtensible Interface* (AXI) signals.
Prefix AR	Denotes AXI read address channel signals.
Prefix AW	Denotes AXI write address channel signals.
Prefix B	Denotes AXI write response channel signals.
Prefix C	Denotes AXI low-power interface signals.
Prefix P	Denotes *Advanced Peripheral Bus* (APB) signals.
Prefix R	Denotes AXI read data channel signals.
Prefix W	Denotes AXI write data channel signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog® method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Additional reading

This section lists publications by ARM and by third parties. You can access ARM documentation at:

`http://infocenter.arm.com/help/index.jsp`

ARM publications

This manual contains information that is specific to the DMAC. See the following documents for other relevant information:

- *PrimeCell DMA Controller (PL330) Implementation Guide* (ARM DII 0192)
- *PrimeCell DMA Controller (PL330) Integration Manual* (ARM DII 0193)
- *AMBA® Designer (FD001) User Guide* (ARM DUI 0333)
- *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* (ARM DUI 0333 Supplement 6)
- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *AMBA AXI Protocol v1.0 Specification* (ARM IHI 0022)
- *AMBA 3 APB Protocol v1.0 Specification* (ARM IHI 0024).

Other publications

This section lists relevant documents published by third parties:

- JEDEC Solid State Technology Association, *JEP106, Standard Manufacturer's Identification Code*, obtainable at `http://www.jedec.org`.

Feedback

ARM welcomes feedback on the DMAC and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

If you have any comments on this manual, send an e-mail to `errata@arm.com`. Give:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the *DMA Controller* (DMAC). It contains the following sections:

- *About the DMAC* on page 1-2
- *Terminology* on page 1-5.

1.1 About the DMAC

The DMAC is an *Advanced Microcontroller Bus Architecture* (AMBA) compliant PrimeCell peripheral that is developed, tested, and licensed by ARM.

The DMAC provides an AXI interface to perform the DMA transfers and two APB interfaces that control its operation. The DMAC implements TrustZone® secure technology with one APB interface operating in the Secure state and the other operating in the Non-secure state. See the *ARM Architecture Reference Manual* for more information about TrustZone technology.

The DMAC includes a small instruction set that provides a flexible method of specifying the DMA operations. This enables it to provide greater flexibility than the fixed capabilities of a *Linked-List Item* (LLI) based DMA controller. To minimize the program memory requirements the DMAC uses variable-length instructions.

Figure 1-1 shows the interfaces that are available on the DMAC.

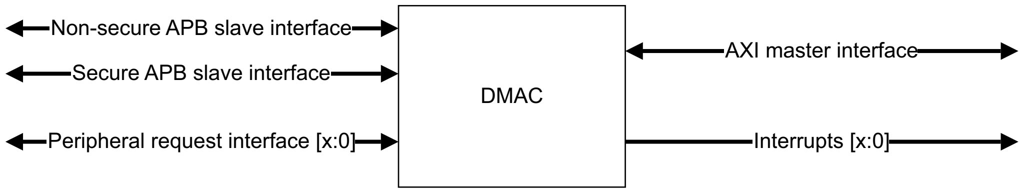

Non-secure APB slave interface
Secure APB slave interface
Peripheral request interface [x:0]
DMAC
AXI master interface
Interrupts [x:0]

Figure 1-1 Interfaces on the DMAC

Figure 1-2 on page 1-3 shows an example system that contains a DMAC.

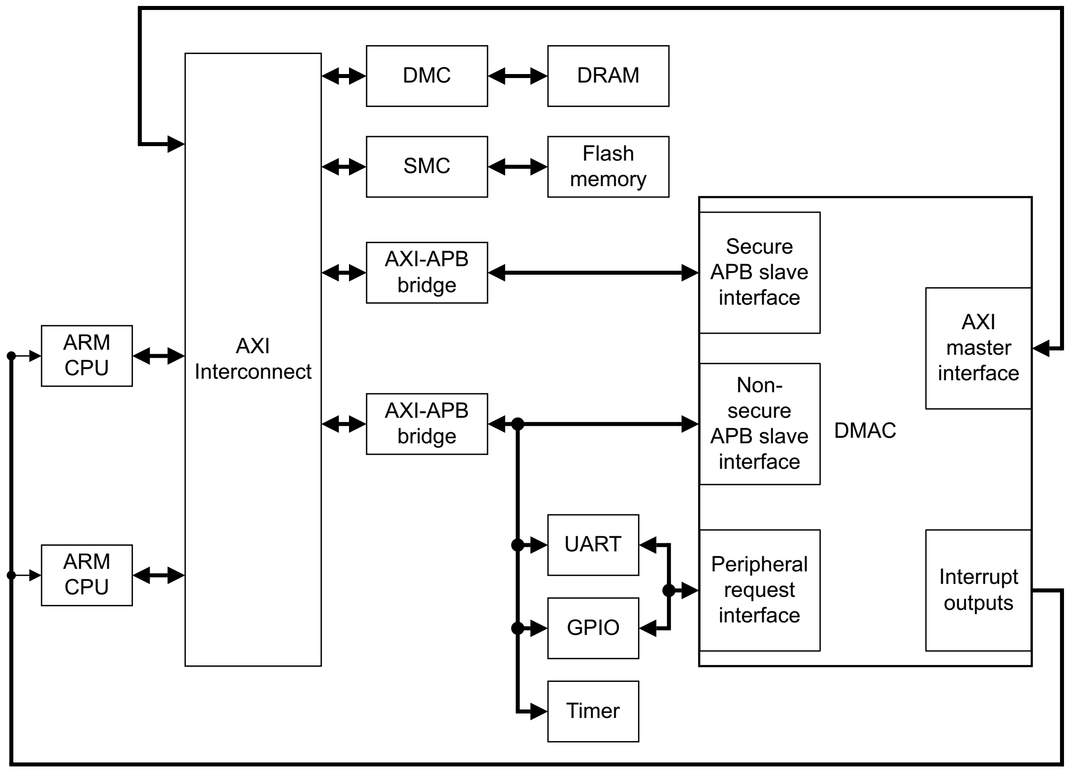


Figure 1-2 Example system

The example system contains:

- AXI bus masters:
  - — a DMAC
  - — two ARM processors.
- an AXI interconnect and two AMBA protocol bridge components
- PrimeCell slaves:
  - — a *Dynamic Memory Controller* (DMC)
  - — a *Static Memory Controller* (SMC)
  - — a Timer
  - — a *General Purpose Input-Output* (GPIO)
  - — a *Universal Asynchronous Receiver-Transmitter* (UART).

The AXI interconnect enables each bus master to access the slaves. The ARM processors can access the APB interfaces of the DMAC by using the appropriate AXI to APB bridge component.

1.1.1 Features of the DMAC

The DMAC provides the following features:

- an instruction set that provides flexibility for programming DMA transfers
- single AXI master interface that performs the DMA transfers
- dual APB slave interfaces, designated as secure and non-secure, for accessing registers in the DMAC
- supports TrustZone technology
- supports multiple transfer types:
  - memory-to-memory
  - memory-to-peripheral
  - peripheral-to-memory
  - scatter-gather
- configurable RTL that enables the DMAC to be optimized for the application
- programmable security state for each DMA channel
- signals the occurrence of various DMA events using the interrupt output signals.

1.1.2 Configurable features of the DMAC

When you implement the DMAC in your design you can configure the:

- AXI data bus width
- number of active AXI read transactions
- number of active AXI write transactions
- number of simultaneously active DMA channels
- depth of the internal data buffer
- number of lines in the instruction cache and how many words a line contains
- depth of the read instruction queue
- depth of the write instruction queue
- number of peripheral request interfaces
- number of interrupt output signals.

Note

See the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* for information about how to configure these features and the values that you can assign.

1.2 Terminology

This manual uses the following terminology:

Configurable

A parameter of the DMAC that you can only change, prior to the RTL being generated. See the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* for information about configuring the DMAC.

Programmable

A parameter of the DMAC that you can change, after the RTL is generated. See Chapter 3 *Programmers Model* for information about programming the DMAC.

Initialization

A feature of the DMAC that is initialized when it exits from reset, depending on the state of the *Tie-off signals* on page A-12. See *Initializing the DMAC* on page 2-14.

DMA channel

A section of the DMAC that controls a DMA cycle by executing its own program thread. You can configure the number of channels that the DMAC contains.

DMA cycle All the DMA transfers that the DMAC must perform, to transfer the programmed number of data packets.

DMA manager

A section of the DMAC that manages the operation of the DMAC by executing its own program thread.

DMA transfer

The action of transferring a single byte, halfword, or word.

Chapter 2
Functional Overview

This chapter describes the major interfaces and components of the DMAC, and how it operates. It contains the following sections:

- *Overview* on page 2-2
- *DMAC interfaces* on page 2-4
- *Operating states* on page 2-10
- *Initializing the DMAC* on page 2-14
- *Using the APB slave interfaces* on page 2-16
- *Peripheral request interface* on page 2-18
- *Using events and interrupts* on page 2-23
- *Security usage* on page 2-29
- *Aborts* on page 2-25
- *Constraints and limitations of use* on page 2-34
- *Programming restrictions* on page 2-35.

2.1 Overview

Figure 2-1 shows a diagram of the DMAC.

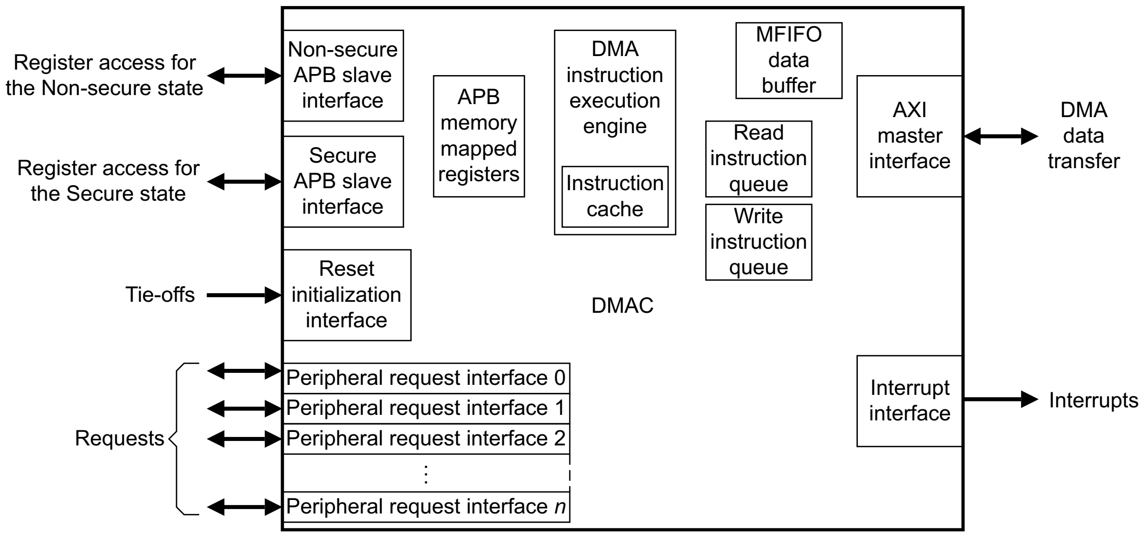

Register access for the Non-secure state
Register access for the Secure state
Tie-offs
Requests
Non-secure APB slave interface
Secure APB slave interface
Reset initialization interface
APB memory mapped registers
DMA instruction execution engine
Instruction cache
MFIFO data buffer
Read instruction queue
Write instruction queue
AXI master interface
DMA data transfer
DMAC
Peripheral request interface 0
Peripheral request interface 1
Peripheral request interface 2
Peripheral request interface *n*
Interrupt interface
Interrupts

Figure 2-1 DMAC block diagram

The DMAC contains an instruction processing block that enables it to process program code that controls a DMA transfer. The program code is stored in a region of system memory that the DMAC accesses using its AXI interface. The DMAC stores instructions temporarily in a cache whose line length and depth are configurable.

You can configure the DMAC with up to eight DMA channels, with each channel being capable of supporting a single concurrent thread of DMA operation. In addition there is a single DMA manager thread that you can use to initialize the DMA channel threads. The DMAC executes up to one instruction for each AXI clock cycle. To ensure that it regularly executes each active thread it alternates by processing the DMA manager thread and then a DMA channel thread. It uses a round-robin process when selecting the next active DMA channel thread to execute.

The DMAC uses variable-length instructions that consist of one to six bytes. It provides a separate *Program Counter* (PC) register for each DMA channel. When a thread requests an instruction from an address, the cache performs a look-up. If a cache hit occurs then the cache immediately provides the data, otherwise the thread is stalled while the DMAC uses the AXI interface to perform a cache line fill. If an instruction is greater than 4 bytes, or spans the end of a cache line, then it performs multiple cache accesses to fetch the instruction.

Note

When a cache line fill is in progress, the DMAC enables other threads to access the cache, but if another cache miss occurs, this stalls the pipeline until the first line fill is complete.

When a DMA channel thread executes a load or store instruction the DMAC adds the instruction to the relevant read queue or write queue. The DMAC uses these queues as an instruction storage buffer prior to it issuing the instructions on the AXI bus. The DMAC also contains a *Multi First-In-First-Out* (MFIFO) data buffer that it uses to store data that it reads, or writes, during a DMA transfer.

Note

To meet your system requirements you can configure the:

- depth of the read queue
- depth of the write queue
- depth of the MFIFO.

It provides multiple interrupt outputs to enable efficient communication of events to external microprocessors. The peripheral request interfaces support the connection of DMA-capable peripherals to enable memory-to-peripheral and peripheral-to-memory DMA transfers to occur, without intervention from a microprocessor.

Dual APB interfaces enable the operation of the DMAC to be partitioned into the Secure state and Non-secure state. You can use the APB interfaces to access status registers and also directly execute instructions in the DMAC.

2.2 DMAC interfaces

The DMAC contains the following interfaces:

- *APB slave interfaces*
- *AXI master interface* on page 2-5
- *Peripheral request interfaces* on page 2-8
- *Interrupt interface* on page 2-9
- *Reset initialization interface* on page 2-9.

2.2.1 APB slave interfaces

The DMAC provides the following APB interfaces:

- non-secure APB slave interface
- secure APB slave interface.

Using these interfaces you can access the registers that control the functionality of the DMAC. Figure 2-2 shows the signal connections for both interfaces.

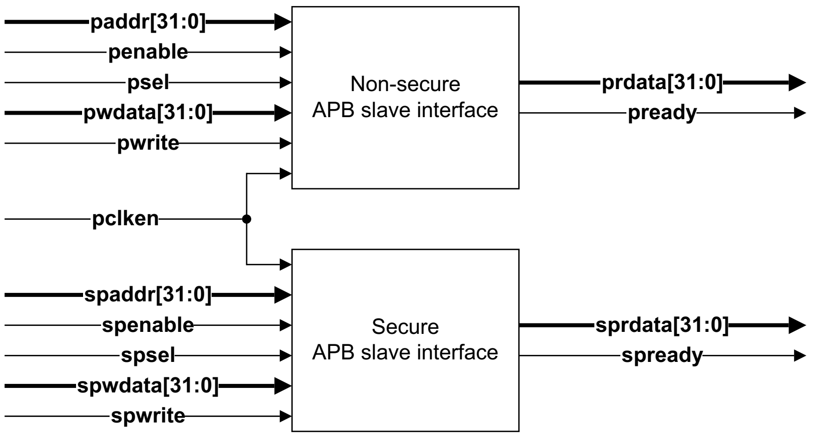


Figure 2-2 APB slave interfaces

The DMAC allocates 4KB of memory for each APB interface and implements the memory map that Chapter 3 *Programmers Model* describes.

The APB interfaces are clocked by the same clock as the AXI domain clock, **aclk**. However, the DMAC provides a clock enable signal, **pclken**, that enables both APB interfaces to operate at a slower clock rate. The clock enable signal must be an integer divisor of **aclk**.

2.2.2 AXI master interface

The DMAC contains a single AXI master interface that enables it to transfer data from a source AXI slave to a destination AXI slave.

The DMAC is compliant to the AMBA AXI protocol. For detailed information about the AXI interface, see the *AMBA AXI Protocol Specification*.

Figure 2-3 on page 2-6 shows the AXI master interface external connections.

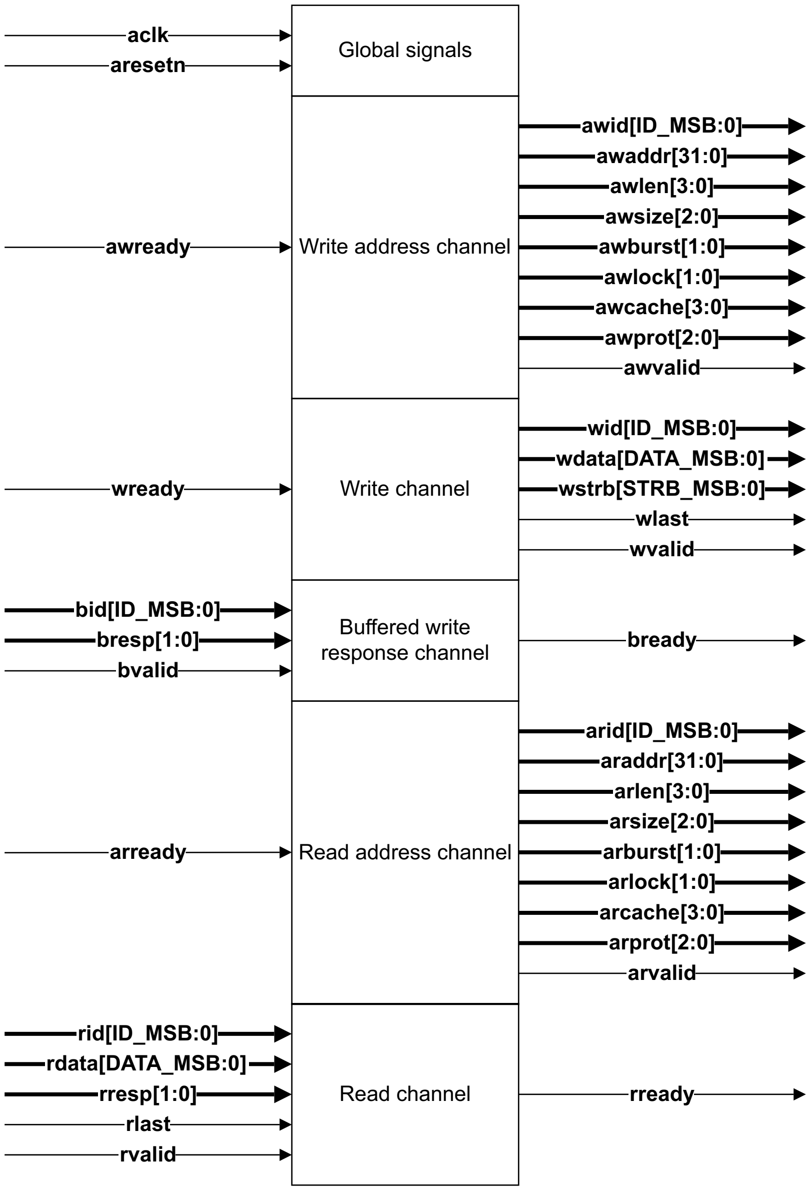


Figure 2-3 AXI master interface connections

Note

In Figure 2-3 on page 2-6:

- **awcache[2]** is tied LOW
- **arcache[3]** is tied LOW
- the DMAC does not support locked or exclusive accesses and therefore **arlock[1:0]** and **awlock[1:0]** are tied LOW
- the value of ID_MSB is dependent on the number of DMA channels in the configured DMAC
- the values of DATA_MSB and STRB_MSB are dependent on the data width of the configured DMAC.

When a DMA channel thread accesses the AXI interface the DMAC signals the AXI identification tag to be the same number as the DMA channel. For example, when the program thread for DMA channel 5 performs a DMA store operation then the DMAC sets **AWID[2:0]** and **WID[2:0]** to b101.

When the DMA manager thread accesses the AXI interface the DMAC signals the AXI identification tag to be the same number as the number of DMA channels that the DMAC provides. For example, if the DMAC is configured to provide eight DMA channels then when the DMA manager performs a read operation the DMAC sets **ARID[3:0]** to b1000.

AXI characteristics for a DMA transfer

Table 2-1 lists how the DMAC controls the setting of the AXI control signals, depending on the type of DMA access it performs.

Table 2-1 AXI characteristics for a DMA transfer

Access type	AxPROT	AxLEN	AxBURST	AxSIZE	AxCACHE
DMA channel load	*Channel Control Registers* on page 3-30				
DMA channel store	*Channel Control Registers* on page 3-30				
DMA manager instruction fetch	1. Privileged 2. Secure state from DNS[a] bit 3. Instruction	See *ARLEN and ARSIZE for instruction fetches*	INCR	See *ARLEN and ARSIZE for instruction fetches*	Cacheable write-through, allocate on reads only.
DMA channel instruction fetch	1. Privileged 2. Secure state from CNS[b] bit 3. Instruction				

a. The *DMA Status Register* on page 3-11 contains the DNS bit.
b. The *Channel Status Registers* on page 3-24, corresponding to the DMA channel, contains the CNS bit.

ARLEN and ARSIZE for instruction fetches

When performing an instruction fetch, the DMAC sets **ARLEN** and **ARSIZE** as follows:

Instruction cache length ≤ AXI data bus width

- **ARLEN** = 1.
- **ARSIZE** = length of instruction cache in bytes.

Instruction cache length > AXI data bus width

- **ARLEN** = ratio of the length of an instruction cache line in bytes to the width of the AXI data bus in bytes.
- **ARSIZE** = width of AXI data bus in bytes.

2.2.3 Peripheral request interfaces

Figure 2-4 on page 2-9 shows the signals that a single peripheral request interface provides.

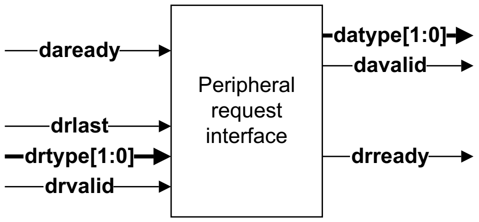


Figure 2-4 Peripheral request interface

The peripheral request interface supports the connection of DMA-capable peripherals. You must configure the number of peripheral request interfaces that you require, as the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* describes.

2.2.4 Interrupt interface

The interrupt interface enables efficient communications of events to an external microprocessor. Figure 2-5 shows the signal that this interface provides.

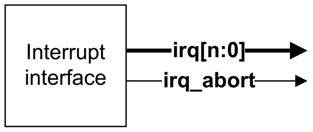


Figure 2-5 Interrupt interface

You must configure the number of interrupts that you require, as the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* describes.

2.2.5 Reset initialization interface

This interface enables you to initialize the operating state of the DMAC as it exits from reset. Figure 2-6 shows the tie-off signals that this interface provides.

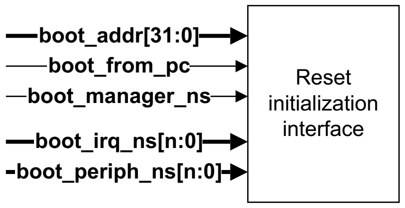


Figure 2-6 Reset initialization interface

2.3 Operating states

Figure 2-7 shows the operating states for the DMA manager thread and DMA channel threads. The DMAC provides a separate state machine for each thread.

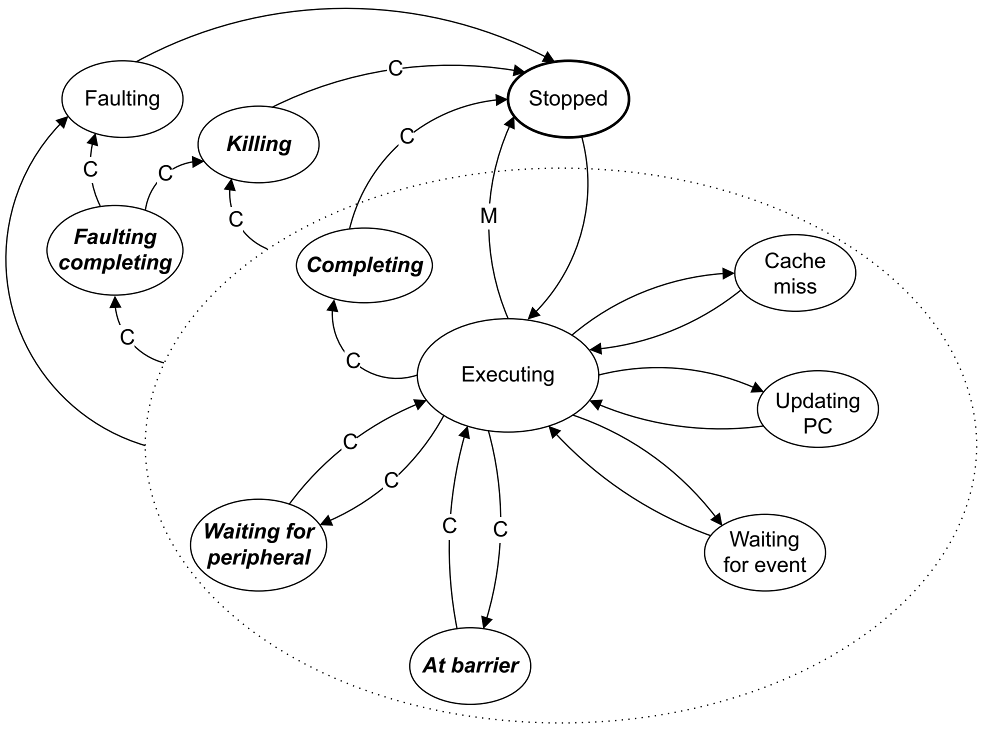


Figure 2-7 Thread operating states

——— **Note** ———

In Figure 2-7, the DMAC permits that:

- only DMA channel threads can use states in bold italics
- arcs with no letter designator indicate state transitions for the DMA manager and DMA channel threads, otherwise use is restricted as follows:

C	DMA channel threads only.
M	DMA manager thread only.

- states within the dotted line can transition to the Faulting completing, Faulting, or Killing states.

After the DMAC exits from reset it sets all DMA channel threads to the Stopped state and the status of **boot_from_pc** controls the DMA manager thread state:

boot_from_pc is LOW

DMA manager thread moves to the Stopped state.

boot_from_pc is HIGH

DMA manager thread moves to the Executing state.

The states are described in:
- *Stopped*
- *Executing*
- *Cache miss* on page 2-12
- *Updating PC* on page 2-12
- *Waiting for event* on page 2-13
- *At barrier* on page 2-13
- *Waiting for peripheral* on page 2-13
- *Faulting completing* on page 2-13
- *Faulting* on page 2-13
- *Killing* on page 2-13
- *Completing* on page 2-13.

2.3.1 Stopped

The thread has an invalid PC and it is not fetching instructions. Depending on the thread type, it can transition to the Executing state by:

DMA manager thread

Writing to the *Debug Command Register* on page 3-37.

DMA channel thread

Programming the DMA manager thread to execute `DMAGO` for a DMA channel thread in the Stopped state.

2.3.2 Executing

The thread has a valid PC and therefore the DMAC includes the thread when it arbitrates.

The DMAC can then change to one of the following states under the following conditions:

Stopped If the DMA manager thread executes `DMAEND`.

Cache miss For any thread, when the instruction cache does not contain the next instruction for that thread.

Updating PC

When the DMAC calculates the address of the next access in the cache.

Waiting for event

If any thread executes `DMAWFE`.

At barrier Either when:

- a DMA channel thread executes `DMARMB`, `DMAWMB`, or `DMAFLUSHP`
- the DMAC updates the MFIFO data buffer, or similar internal control function.

Waiting for peripheral

If a DMA channel thread executes `DMAWFP`.

Killing If a DMA channel thread executes `DMAKILL`.

Faulting completing

If an AXI bus error occurs when the DMAC is performing a DMA transfer. This state transition is only applicable to DMA channel threads.

Faulting Either:

- if an AXI bus error occurs when the DMAC fetches an instruction
- the DMAC executes an undefined instruction.

Completing When a DMA channel thread executes `DMAEND`.

2.3.3 Cache miss

The thread is stalled and the DMAC is performing a cache line fill. After it completes the cache fill the thread returns to the Executing state.

2.3.4 Updating PC

The DMAC is calculating the address of the next access in the cache. After it calculates the PC the thread returns to the Executing state.

2.3.5 Waiting for event

The thread is stalled and is waiting for the DMAC to execute `DMASEV` using the corresponding event number. After the corresponding event occurs the thread returns to the Executing state.

2.3.6 At barrier

A DMA channel thread is stalled and the DMAC is waiting for transactions on the AXI bus to complete. After the AXI transactions complete the thread returns to the Executing state.

2.3.7 Waiting for peripheral

A DMA channel thread is stalled and the DMAC is waiting for the peripheral to provide the requested data. After the peripheral provides the data then the thread returns to the Executing state.

2.3.8 Faulting completing

A DMA channel thread is waiting for the AXI interface to signal that the outstanding load or store transactions are complete. After the transactions complete then the thread moves to the Faulting state.

2.3.9 Faulting

The thread is stalled indefinitely. The thread moves to the Stopped state when you use the *Debug Command Register* on page 3-37 to instruct the DMAC to execute `DMAEND` or `DMAKILL` for that thread.

2.3.10 Killing

A DMA channel thread is waiting for the AXI interface to signal that the outstanding load or store transactions are complete. After the transactions complete then the thread moves to the Stopped state.

2.3.11 Completing

A DMA channel thread is waiting for the AXI interface to signal that the outstanding load or store transactions are complete. After the transactions complete then the thread moves to the Stopped state.

2.4 Initializing the DMAC

The DMAC provides several tie-off signals that initialize its operating state when it exits from reset. The initialization of the tie-offs is described in:

- *Setting the security state of the DMA manager*
- *Setting the location of the first instruction for the DMAC to execute*
- *Setting the security state for the interrupt outputs* on page 2-15
- *Setting the security state for a peripheral request interface* on page 2-15.

2.4.1 Setting the security state of the DMA manager

The **boot_manager_ns** signal is the only method of setting the security state of the DMA manager. When the DMAC exits from reset, it reads the status of **boot_manager_ns** and sets the security of the DMA manager as Table A-11 on page A-12 lists.

——— **Note** ———

Once set, the security state remains constant until the DMAC is reset using a state transition on **aresetn**.

———————————

See *DMA manager thread is in the Secure state* on page 2-29 and *DMA manager thread is in the Non-secure state* on page 2-29 for how the security state of the DMA manager affects how the DMAC operates.

2.4.2 Setting the location of the first instruction for the DMAC to execute

After the DMAC exits from reset, the status of the **boot_from_pc** signal controls if the DMAC:

- enters the Executing state and:
  - — updates the *DMA Program Counter Register* on page 3-12 using the address that **boot_addr[31:0]** provides
  - — fetches and executes the instruction from the address that the *DMA Program Counter Register* on page 3-12 contains.

  ——— **Note** ———

  - — You must ensure that the setting of the **boot_addr[31:0]** signals points to a region in system memory that contains the boot program for the DMAC.
  - — Also, if you set **boot_manager_ns** so that the DMA manager operates in the Non-secure state then the boot program must reside in a non-secure region of memory.

  ———————————

- enters the Stopped state. You must then provide the first instruction to the DMAC by using one of the slave APB interfaces.

Table A-11 on page A-12 lists the settings for **boot_from_pc**.

2.4.3 Setting the security state for the interrupt outputs

The DMAC provides the **boot_irq_ns[x:0]** signals to enable you to assign each **irq[x]** signal to a security state as Table A-12 on page A-13 lists.

Note

Once set, the security state of each **irq[x]** remains constant until the DMAC is reset using a state transition on **aresetn**.

See *Security usage* on page 2-29 for how the security state of the **irq[x]** signals affects how the DMAC executes the `DMAWFE` and `DMASEV` instructions.

2.4.4 Setting the security state for a peripheral request interface

The DMAC provides the **boot_periph_ns[x:0]** signals to enable you to assign each peripheral request interface to a security state as Table A-12 on page A-13 lists.

Note

Once set, the security state of each peripheral request interface remains constant until the DMAC is reset using a state transition on **aresetn**.

See *Security usage* on page 2-29 for how the security state of the peripheral request interfaces affects how a DMA channel thread executes the `DMAWFP`, `DMALDP`, `DMASTP`, or `DMAFLUSHP` instructions.

2.5 Using the APB slave interfaces

The APB slave interface connects the DMAC to the APB and enables a microprocessor to access the registers that Chapter 3 *Programmers Model* describes. Using these registers, a microprocessor can:

- access the status of the DMA manager thread
- access the status of the DMA channel threads
- enable or clear interrupts
- enable events
- issue an instruction for the DMAC to execute by programming the following debug registers:
  - *Debug Command Register* on page 3-37
  - *Debug Instruction-0 Register* on page 3-38
  - *Debug Instruction-1 Register* on page 3-39.

The following section describes:

- *Issuing instructions to the DMAC using an APB interface*.

2.5.1 Issuing instructions to the DMAC using an APB interface

When the DMAC is operating in real-time then you can only issue a limited subset of instructions as follows:

`DMAGO`	Starts a DMA transaction using a DMA channel that you specify.
`DMASEV`	Signals the occurrence of an event, or interrupt, using an event number that you specify.
`DMAKILL`	Terminates a thread.

You must ensure that you use the appropriate APB interface, depending on the security state that the **boot_manager_ns** initializes the DMAC to operate in. For example, if the DMAC is in the Secure state then you must issue the instruction using the secure APB interface, otherwise the DMAC ignores the instruction. You can use the secure APB interface, or the non-secure APB interface, to start or restart a DMA channel when the DMAC is in the Non-secure state.

Note

Before you can issue instructions using the debug instruction registers or the *Debug Command Register* on page 3-37 you must read the *Debug Status Register* on page 3-37 to ensure that debug is idle.

When the DMAC receives an instruction from an APB slave interface it can take several clock cycles before it can process the instruction, for example, if the pipeline is busy processing another instruction.

Note

Prior to issuing `DMAGO`, you must ensure that the system memory contains a suitable program for the DMAC to execute, starting at the address that the `DMAGO` specifies.

Example 2-1 lists the necessary steps to start a DMA channel thread using the debug instruction registers.

Example 2-1 Using `DMAGO` with the debug instruction registers

1. Create a program for the DMA channel.
2. Store the program in a region of system memory.

Using one of the APB interfaces on the DMAC, program a `DMAGO` instruction:

3. Poll the *Debug Status Register* on page 3-37 to ensure that debug is idle, that is, the dbgstatus bit is 0.
4. Write to the *Debug Instruction-0 Register* on page 3-38 setting the:
   - Instruction byte 0 encoding for `DMAGO`.
   - Instruction byte 1 encoding for `DMAGO`.
   - Debug thread bit to 0. This selects the DMA manager thread.
5. Write to the *Debug Instruction-1 Register* on page 3-39 with the `DMAGO` instruction byte [5:2] data. You must set these four bytes to the address of the first instruction in the program, that was written to system memory in step 2.

Instruct the DMAC to execute the instruction that the debug instruction registers contain by:

6. Writing zero to the *Debug Command Register* on page 3-37. The DMAC starts the DMA channel thread and sets the dbgstatus bit to 1.

   After the DMAC completes execution of the instruction it clears the dbgstatus bit to 0.

2.6 Peripheral request interface

Figure 2-8 shows that the peripheral request interface consists of a peripheral request bus and a DMAC acknowledge bus that use the prefixes:

dr	The peripheral request bus.
da	The DMAC acknowledge bus.

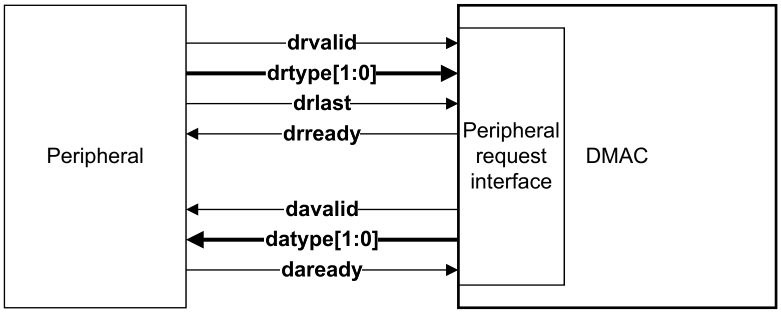


Figure 2-8 Request and acknowledge buses on the peripheral request interface

Both buses use the **valid/ready** handshake that the AXI protocol describes. For more information on the handshake process, see the *AMBA AXI Protocol v1.0 Specification*.

The peripheral uses **drtype[1:0]** to either:

- request a single transfer
- request a burst transfer
- acknowledge a flush request.

The DMAC uses **datype[1:0]** to either:

- signal when it completes the requested single transfer
- signal when it completes the requested burst transfer
- issue a flush request.

drlast enables the peripheral to notify the DMAC when the last DMA transfer commences.

Note

If you configure the DMAC to provide more than one peripheral request interface then each interface is assigned a unique identifier, _<x>. Where <x> represents the number of the interface. See *Peripheral request interface* on page A-9 for information about how the identifier is appended to the signal name.

The following sections describe:

- *Mapping to a DMA channel*
- *Handshake rules*
- *Peripheral length management* on page 2-20
- *DMAC length management* on page 2-20
- *Peripheral request interface timing diagrams* on page 2-20.

2.6.1 Mapping to a DMA channel

The DMAC enables you to assign a peripheral request interface to any of the DMA channels. When a DMA channel thread executes `DMAWFP` the value programmed in the peripheral [4:0] field specifies the peripheral that is associated with that DMA channel. See *DMAWFP<S|B|P>* on page 4-22.

2.6.2 Handshake rules

The DMAC uses the DMA handshake rules that Table 2-2 lists, when a DMA channel thread is active, that is, not in the Stopped state.

Table 2-2 Handshake rules

Rule	Description[a]
1	**drvalid** can change from LOW to HIGH on any **aclk** cycle but it must only change from HIGH to LOW when **drready** is HIGH.
2	**drtype** can only change when either: • **drready** is HIGH • **drvalid** is LOW.
3	**drlast** can only change when either: • **drready** is HIGH • **drvalid** is LOW.
4	**davalid** can change from LOW to HIGH on any **aclk** cycle but it must only change from HIGH to LOW when **daready** is HIGH.
5	**datype** can only change when either: • **daready** is HIGH • **davalid** is LOW.

a. All signals are only permitted to change state when **aclk** changes state.

2.6.3 Peripheral length management

The peripheral request interface enables a peripheral to control the quantity of data that a DMA cycle contains, without the DMAC being aware of how many data transfers that it contains. The peripheral controls the DMA cycle by using:

- **drtype[1:0]** to select a single or burst transfer
- **drlast** to notify the DMAC when the last transfer commences.

2.6.4 DMAC length management

If you are using the peripheral request interface and the DMAC is controlling the quantity of data in a DMA cycle then you must use the `DMALDP<S|B>` and `DMASTP<S|B>` instructions. Using these instructions enables the DMAC to update **datype[1:0]**, to notify the peripheral when it completes the final DMA transfer.

2.6.5 Peripheral request interface timing diagrams

The following sections provide examples of the functional operation of the peripheral request interface using the rules that *Handshake rules* on page 2-19 describe:

- *Burst request*
- *Single and burst request* on page 2-21
- *DMAC performs single transfers for a burst request* on page 2-22.

Burst request

Figure 2-9 shows the DMA request timing when a peripheral requests a burst transfer.

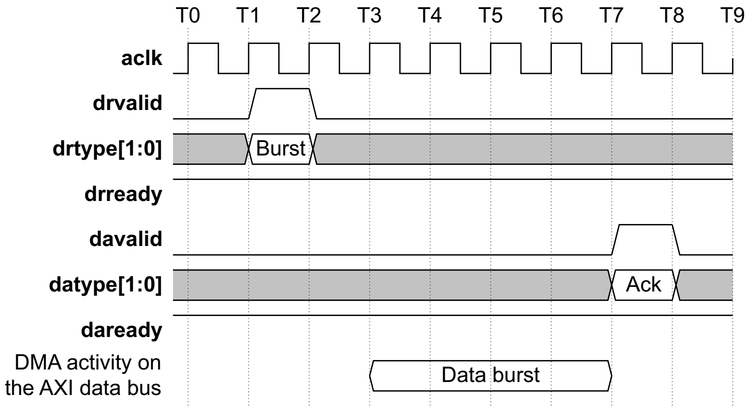


Figure 2-9 Burst request signaling

In Figure 2-9 on page 2-20:

T1	The DMAC detects a request for a burst transfer.
T3 - T6	The DMAC performs a burst transfer.
T7	The DMAC sets **davalid** HIGH and sets **datype[1:0]** to indicate that the burst transfer is complete.

Single and burst request

Figure 2-10 shows the DMA request timing when a peripheral requests a single and a burst transfer.

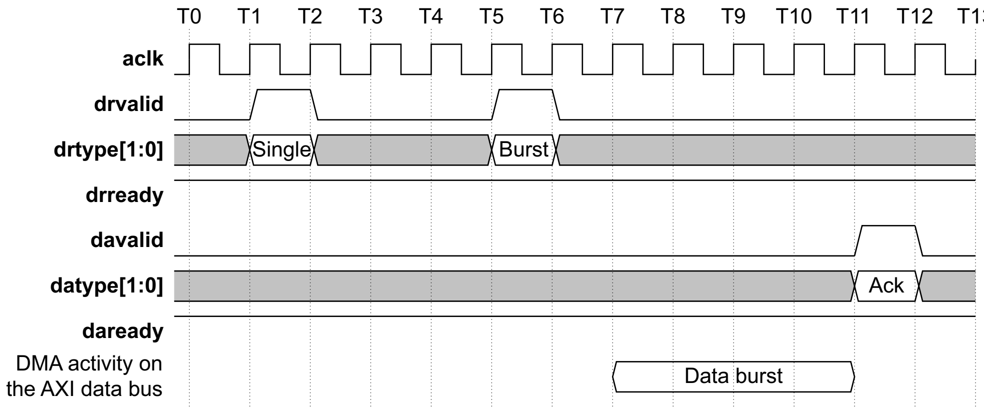


Figure 2-10 Single and burst request signaling

In Figure 2-10:

T1	The DMAC detects a request for a single transfer.
T3	The DMAC ignores the single transfer request because the DMA channel thread had executed a `DMAWFPB` instruction. See *DMAWFP<S|B|P>* on page 4-22.
T5	The DMAC detects a request for a burst transfer.
T7 - T10	The DMAC performs a burst transfer.
T11	The DMAC sets **davalid** HIGH and sets **datype[1:0]** to indicate that the burst transfer is complete.

DMAC performs single transfers for a burst request

Figure 2-11 shows the DMA request timing when a peripheral requests a burst transfer but the DMAC has insufficient data remaining in the MFIFO to generate a burst and therefore completes the request using single transfers.

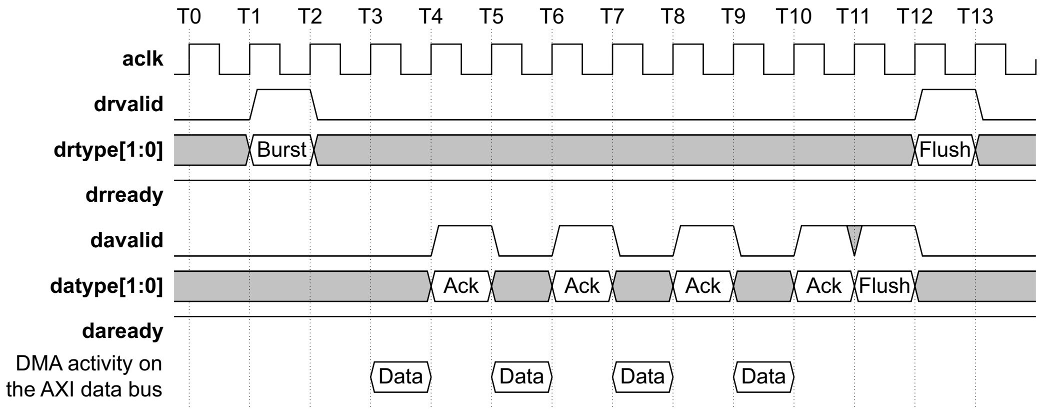


Figure 2-11 Single transfers for a burst request

In Figure 2-11:

T1	The DMAC detects a request for a burst transfer.
T3	The MFIFO contains insufficient data for the DMAC to generate a burst transfer and therefore the DMAC performs a single transfer.
T4	The DMAC signals **davalid** and **datype[1:0]** to indicate completion of a single transfer.
T5 - T10	The DMAC performs the remaining three single transfers.
T11	The DMAC signals **davalid** and **datype[1:0]** to request the peripheral to flush the contents of any control registers that are associated with the current DMA cycle.
T12	The peripheral signals **drvalid** and **drtype[1:0]** to acknowledge the flush request.

2.7 Using events and interrupts

By programming the *Interrupt Enable Register* on page 3-13, you can use the `DMASEV` instruction to either:

- generate an event, for the DMAC to process
- signal an interrupt using one of the **irq[x]** outputs.

Note

The number of events or interrupts that the DMAC can generate depends on its configuration. See the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement*.

The following sections describe:

- *Using an event to restart DMA channels*
- *Interrupting a microprocessor* on page 2-24.

2.7.1 Using an event to restart DMA channels

When you program the *Interrupt Enable Register* on page 3-13 to generate an event, you can use the `DMASEV` and `DMAWFE` instructions, to restart one or more DMA channels.

To restart a single DMA channel:

1. The first DMA channel executes `DMAWFE and then stalls while it waits for the event to occur.`
2. The other DMA channel executes `DMASEV` using the same event number. This generates an event and the first DMA channel restarts. The DMAC clears the event, one clock cycle after it executes `DMASEV`.

You can program multiple channels to wait for the same event. For example, if four DMA channels have all executed `DMAWFE` for event 12 then when another DMA channel executes `DMASEV` for event 12, the four DMA channels all restart at the same time. The DMAC clears the event, one clock cycle after it executes `DMASEV`.

Note

Because the event only lasts for a single clock cycle, if one of the DMA channels does not execute `DMAWFE` prior to the `DMASEV` occurring then that DMA channel must wait until the appropriate event is generated again.

An event only lasts longer than a single clock cycle when a DMA channel executes `DMASEV` and no other DMA channel has executed a `DMAWFE` using the same event number.

2.7.2 Interrupting a microprocessor

The DMAC provides the **irq[x]** signals for use as active-high level-sensitive interrupts to external microprocessors. When you program the *Interrupt Enable Register* on page 3-13 to generate an interrupt, then after the DMAC executes `DMASEV` it sets the corresponding **irq[x]** HIGH.

An external microprocessor can clear the interrupt by writing to the *Interrupt Clear Register* on page 3-17.

Note

Executing `DMAWFE` does not clear an interrupt.

2.8 Aborts

This section describes:

- *Abort types*
- *Abort sources*
- *Watchdog abort* on page 2-26
- *Abort handling* on page 2-26.

2.8.1 Abort types

An abort can be defined as either precise or imprecise depending on whether the DMAC provides an abort handler with the precise state of the DMAC when an abort occurs.

precise	The DMAC updates the PC Register with the address of the instruction that created the abort.
imprecise	The PC Register might contain the address of an instruction that did not cause the abort occur.

2.8.2 Abort sources

The DMAC signals a precise abort under the following conditions:

- A DMA channel thread in the Non-secure state attempts to program the *Channel Control Registers* on page 3-30 and generate a secure AXI transaction.
- A DMA channel thread in the Non-secure state executes `DMAWFE` or `DMASEV` for an event that is set as secure. The **boot_irq_ns** tie-offs initialize the security state for an event.

  Note

  For each event, the *Interrupt Enable Register* on page 3-13 controls if the DMAC generates an event or signals an interrupt.

- A DMA channel thread in the Non-secure state executes `DMAWFP`, `DMALDP`, `DMASTP`, or `DMAFLUSHP` for a peripheral request interface that is set as secure. The **boot_periph_ns** tie-offs initialize the security state for a peripheral request interface.
- A DMA manager thread in the Non-secure state executes `DMAGO` to attempt to start a secure DMA channel thread.
- The DMAC receives an ERROR response on the AXI master interface when it performs an instruction fetch.

- A thread executes an undefined instruction.
- A thread executes an instruction with an operand that is invalid for the configuration of the DMAC.

Note

When the DMAC signals a precise abort the instruction that triggers the abort is not executed, instead the DMAC executes a `DMANOP`.

The DMAC signals an imprecise abort under the following conditions:

- The DMAC receives an ERROR response on the AXI master interface when it performs a data load.
- The DMAC receives an ERROR response on the AXI master interface when it performs a data store.
- A DMA channel thread executes `DMALD` and the MFIFO is too small to store the data.
- A DMA channel thread executes `DMAST` and the MFIFO contains insufficient data to complete the data transfer.
- A DMA channel thread locks up due to resource starvation and this causes the internal watchdog timer to time out.

2.8.3 Watchdog abort

The DMAC contains logic to determine when one or more channels have locked up due to resource starvation. For example, if the MFIFO or load/store queues are full this might prevent the DMAC from executing instructions. The DMAC aborts the DMA channel thread if the lock-up condition persists for 1024 **aclk** cycles.

2.8.4 Abort handling

The architecture of the DMAC is not designed to recover from an abort and therefore you must use an external agent, such as a microprocessor, to terminate a thread when an abort occurs. Figure 2-12 on page 2-27 shows the operating states for the DMA channel and DMA manager threads after an abort occurs.

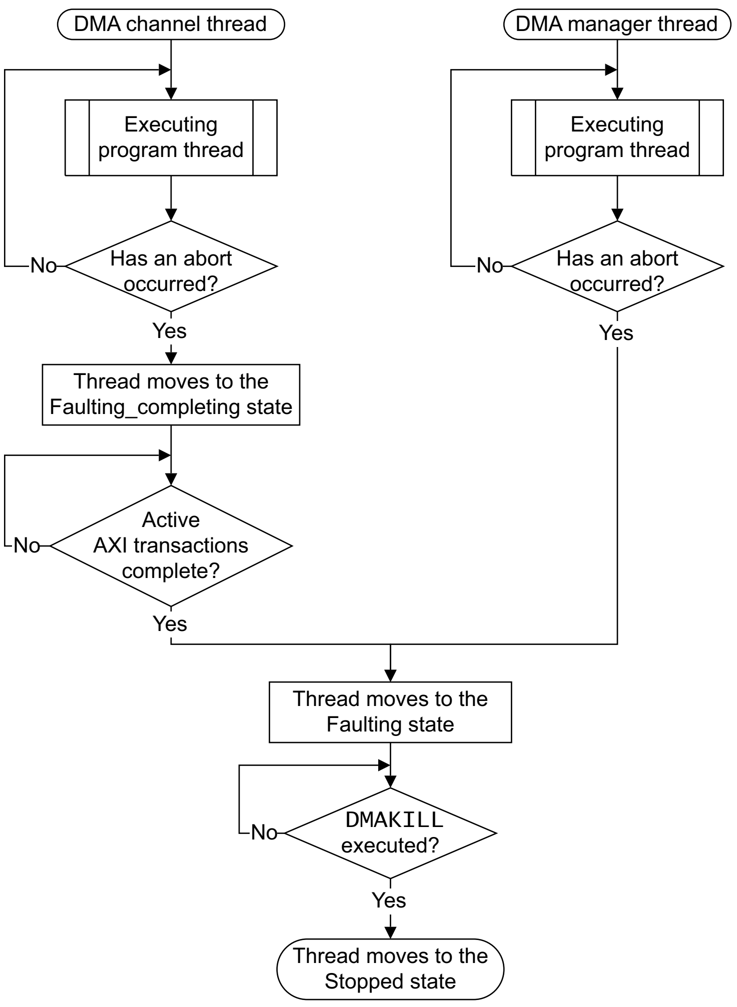


Figure 2-12 Abort process

After an abort occurs then the action the DMAC takes depends on the thread type:

DMA channel thread

The thread immediately moves to the Faulting completing state. In this state the DMAC:

- sets **irq_abort** HIGH
- stops executing instructions for the DMA channel
- invalidates all cache entries for the DMA channel

- updates the *Channel Program Counter Registers* on page 3-26 to contain the address of the aborted instruction provided that the abort was precise
- does not generate AXI accesses for any instructions remaining in the read queue and write queue
- permits currently active AXI transactions to complete.

  Note

  After the transactions for the DMA channel complete the thread moves to the Faulting state.

DMA manager thread

The thread immediately moves to the Faulting state and the DMAC sets **irq_abort** HIGH.

The external agent can respond to the assertion of **irq_abort** by:

- reading the status of *Fault Status DMA Manager Register* on page 3-18 to determine if the DMA manager is Faulting. When in the Faulting state the *Fault Status DMA Manager Register* on page 3-18 provides the cause of the abort.
- reading the status of *Fault Status DMA Channel Register* on page 3-19 to determine if a DMA channel is Faulting. When in the Faulting state the *Fault Type DMA Channel Registers* on page 3-21 provides the cause of the abort.

To enable a thread in the Faulting state to move to the Stopped state, the external agent must:

1. Program the *Debug Instruction-0 Register* on page 3-38 with the encoding for the `DMAKILL` instruction.
2. Write to the *Debug Command Register* on page 3-37.

   Note

   If the aborted thread is secure then you must use the secure APB interface to update these registers.

After a thread in the Faulting state executes `DMAKILL` it moves to the Stopped state.

2.9 Security usage

When the DMAC exits from reset, the status of the configuration signals that *Tie-off signals* on page A-12 describes, configures the security for the:

- DMA manager thread. The DNS bit in the *DMA Status Register* on page 3-11 provides the security state of the DMA manager thread.
- **irq[x]** signals. The INS bit in the *Configuration Register 3* on page 3-44 provides the security state of these signals.
- Peripheral request interfaces. The PNS bit in the *Configuration Register 4* on page 3-45 provides the security state of these interfaces.

Additionally, each DMA channel thread has a dynamic non-secure bit, CNS, that is valid when the channel is not in the Stopped state.

2.9.1 DMA manager thread is in the Secure state

If the DNS bit is 0, the DMA manager thread operates in the Secure state and it only performs secure instruction fetches. When a DMA manager thread in the Secure state processes:

`DMAGO` It uses the security setting that the ns bit provides, to set the security state of the DMA channel thread by writing to the CNS bit for that channel.

`DMAWFE` It halts execution of the thread until the event occurs. When the event occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding INS bit.

`DMASEV` It signals the appropriate **irq[x]**, irrespective of the security state of the corresponding INS bit.

2.9.2 DMA manager thread is in the Non-secure state

If the DNS bit is 1, the DMA manager thread operates in the Non-secure state and it only performs non-secure instruction fetches. When a DMA manager thread in the Non-secure state processes:

`DMAGO` It uses the security setting that the ns bit provides, to control if it starts a DMA channel thread:

ns = 0 The DMAC does not start a DMA channel thread and instead it:

1. Executes a `NOP`.

2. Sets the *Fault Status DMA Manager Register* on page 3-18.
3. Sets the dmago_err bit in the *Fault Type DMA Manager Register* on page 3-20.
4. Moves the DMA manager to the Faulting state.

ns = 1 The DMAC starts a DMA channel thread in the Non-secure state and programs the CNS bit to be non-secure.

`DMAWFE` It halts execution of the thread until the event occurs. When the event occurs, the DMAC only continues execution of the thread if the corresponding INS bit is in the Non-secure state. If the INS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the *Fault Status DMA Manager Register* on page 3-18.
3. Sets the mgr_evnt_err bit in the *Fault Type DMA Manager Register* on page 3-20.
4. Moves the DMA manager to the Faulting state.

`DMASEV` It only signals the appropriate **irq[x]** if the corresponding INS bit is in the Non-secure state. If the INS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the *Fault Status DMA Manager Register* on page 3-18.
3. Sets the mgr_evnt_err bit in the *Fault Type DMA Manager Register* on page 3-20.
4. Moves the DMA manager to the Faulting state.

2.9.3 DMA channel thread is in the Secure state

When the CNS bit is 0, the DMA channel thread is programmed to operate in the Secure state and it only performs secure instruction fetches.

When a DMA channel thread in the Secure state processes the following instructions:

`DMAWFE` It halts execution of the thread until the event occurs. When the event occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding INS bit.

`DMASEV` It signals the appropriate **irq[x]**, irrespective of the security state of the corresponding INS bit.

`DMAWFP` It halts execution of the thread until the peripheral signals a DMA request. When this occurs, the DMAC continues execution of the thread, irrespective of the security state of the corresponding PNS bit.

`DMALDP`, `DMASTP`

It sends a message to the peripheral to communicate that data transfer is complete, irrespective of the security state of the corresponding PNS bit.

`DMAFLUSHP` It clears the state of the peripheral and sends a message to the peripheral to resend its level status, irrespective of the security state of the corresponding PNS bit.

When a DMA channel thread is in the Secure state it enables the DMAC to perform secure and non-secure AXI accesses.

2.9.4 DMA channel thread is in the Non-secure state

When the CNS bit is 1, the DMA channel thread is programmed to operate in the Non-secure state and it only performs non-secure instruction fetches.

When a DMA channel thread in the Non-secure state processes the following instructions:

`DMAWFE` It halts execution of the thread until the event occurs. When the event occurs, the DMAC only continues execution of the thread if the corresponding INS bit is in the Non-secure state. If the INS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 corresponding to the DMA channel number.
3. Sets the ch_evnt_err bit in the *Fault Type DMA Channel Registers* on page 3-21.
4. Moves the DMA channel to the Faulting completing state.

`DMASEV` It only signals the appropriate **irq[x]** if the corresponding INS bit is in the Non-secure state. If the INS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 corresponding to the DMA channel number.
3. Sets the ch_evnt_err bit in the *Fault Type DMA Channel Registers* on page 3-21.
4. Moves the DMA channel to the Faulting completing state.

`DMAWFP` It halts execution of the thread until the peripheral signals a DMA request. When this occurs, the DMAC only continues execution of the thread if the corresponding PNS bit is in the Non-secure state. If the PNS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 corresponding to the DMA channel number.
3. Sets the ch_periph_err bit in the *Fault Type DMA Channel Registers* on page 3-21.
4. Moves the DMA channel to the Faulting completing state.

`DMALDP`, `DMASTP`

It only sends a message to the peripheral to communicate that data transfer is complete, if the corresponding PNS bit is in the Non-secure state. If the PNS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 corresponding to the DMA channel number.
3. Sets the ch_periph_err bit in the *Fault Type DMA Channel Registers* on page 3-21.
4. Moves the DMA channel to the Faulting completing state.

`DMAFLUSHP` It only clears the state of the peripheral and sends a message to the peripheral to resend its level status, if the corresponding PNS bit is in the Non-secure state. If the PNS bit is in the Secure state the DMAC:

1. Executes a `NOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 corresponding to the DMA channel number.
3. Sets the ch_periph_err bit in the *Fault Type DMA Channel Registers* on page 3-21.
4. Moves the DMA channel to the Faulting completing state.

When a DMA channel thread is in the Non-secure state and a `DMAMOV CCR` instruction attempts to perform a secure AXI transaction then the DMAC:

1. Executes a `DMANOP`.
2. Sets the appropriate bit in the *Fault Status DMA Channel Register* on page 3-19 that corresponds to the DMA channel number.
3. Sets the ch_rdwr_err bit in the *Fault Type DMA Channel Registers* on page 3-21.

4. Moves the DMA channel thread to the Faulting completing state.

2.10 Constraints and limitations of use

This section describes:

- *DMA channel arbitration*
- *DMA channel prioritization*
- *Instruction cache latency*.

2.10.1 DMA channel arbitration

The DMAC uses a round-robin scheme to service the active DMA channels. To ensure that the DMAC continues to service the DMA manager then it always services the DMA manager prior to it servicing the next DMA channel.

It is not possible to alter the arbitration process of the DMAC.

2.10.2 DMA channel prioritization

The DMAC responds to all active DMA channels with equal priority. It is not possible to increase the priority of a DMA channel over any other DMA channels.

2.10.3 Instruction cache latency

When a cache miss occurs, the latency to service the request is mainly dependent on the read latency of the AXI bus. The latency that the DMAC adds is minimal.

2.11 Programming restrictions

The following sections describe restrictions that apply when programming the DMAC:

- *Fixed unaligned bursts*
- *Endian swap size restrictions*
- *Updating DMA channel control registers during a DMA cycle* on page 2-36
- *Full MFIFO causes DMAC watchdog to abort a DMA channel* on page 2-36.

2.11.1 Fixed unaligned bursts

The DMAC does not support fixed unaligned bursts. It is a programming error if you program the following conditions:

Unaligned read

- src_inc field is 0 in the *Channel Control Registers* on page 3-30
- the *Source Address Registers* on page 3-27 contains an address that is not aligned to the size of data that the src_burst_size field contains.

Unaligned write

- dst_inc field is 0 in the *Channel Control Registers* on page 3-30
- the *Destination Address Registers* on page 3-29 contains an address that is not aligned to the size of data that the dst_burst_size field contains.

2.11.2 Endian swap size restrictions

If you program the endian_swap_size field in the *Channel Control Registers* on page 3-30 to enable a DMA channel to perform an endian swap then you must set the *Source Address Registers* on page 3-27 and *Destination Address Registers* on page 3-29, to contain an address that is aligned to the value that the endian_swap_size field contains.

Also, if you program the src_inc field in the *Channel Control Registers* on page 3-30 to use a fixed address then you must program the src_burst_size field to select a burst size that is equal or greater than the value that the endian_swap_size field specifies. Similarly, if you program the dst_inc field to select a fixed destination address then you must program the dst_burst_size field to select a burst size that is equal or greater than the value that the endian_swap_size field specifies.

2.11.3 Updating DMA channel control registers during a DMA cycle

Prior to a DMA cycle commencing, the values you program in to the *Channel Control Registers* on page 3-30, *Source Address Registers* on page 3-27, and *Destination Address Registers* on page 3-29 control the data byte lane manipulation that the DMAC performs when it transfers the data from the source address to the destination address.

If during a DMA cycle, you update the *Destination Address Registers* on page 3-29 or certain fields in the *Channel Control Registers* on page 3-30 then the DMAC creates a new entry in the MFIFO. To enable the DMAC to transfer the previous entry in the MFIFO, it waits for active AXI transactions for that DMA channel to complete. After the AXI transactions complete, the DMAC deletes the entry in the MFIFO and therefore, any remaining data associated with the entry is no longer accessible to the DMAC. The DMAC then uses the new entry in the MFIFO to store the AXI transactions for that DMA channel.

——— **Note** ———

The DMAC creates a new entry in the MFIFO when the destination data becomes non-contiguous because updates occur to:

- *Destination Address Registers* on page 3-29
- the following fields in the *Channel Control Registers* on page 3-30:
  - src_inc
  - dst_inc
  - dst_burst_size, when dst_inc is set to 0
  - endian_swap_size.

———————

2.11.4 Full MFIFO causes DMAC watchdog to abort a DMA channel

You must take care when programming the DMAC that you do not perform too many load instructions that might completely fill the MFIFO, prior to then issuing store instructions to empty the MFIFO. When the MFIFO is full and the current or next instruction that updates the MFIFO is a load then the DMAC cannot perform any more DMA transfers and the internal watchdog aborts the DMA channel.

The following sections describe the following lock-up situations:

- *Single DMA channel and lock-up situation*
- *Multiple DMA channels and lock-up situation* on page 2-38.

Single DMA channel and lock-up situation

Example 2-2 on page 2-37 shows a program that causes the DMAC to lock-up, when it is configured with a read queue depth of 2 and an MFIFO depth of 8.

Example 2-2 Program that causes the DMAC to lock-up

```
# Set up for 8-beat 32-bit transactions to both source and destination
   DMAMOV CCR, SS32 SB8 DS32 DB8
# Load data from source
   DMALD
   DMALD
   DMALD
   DMALD
# Store data to destination
   DMAST
   DMAST
   DMAST
   DMAST
   DMAEND
```

In Example 2-2:

1. The DMAC fetches the first `DMALD` into the read queue. It executes the instruction and completely fills the MFIFO. After the instruction completes it removes the `DMALD` from the read queue.
2. The DMAC fetches the second `DMALD` into the read queue. It executes the instruction but because the MFIFO is full the AXI read transaction does not complete and the instruction remains in the read queue.
3. The DMAC fetches the third `DMALD` into the read queue. It executes the instruction but because the MFIFO is full the AXI read transaction does not complete and the instruction remains in the read queue.
4. The read queue is now full and therefore the DMAC cannot fetch any more instructions. Because the DMAC cannot execute a `DMAST` instruction then it locks up and after the watchdog times out, the thread aborts.

Example 2-3 shows a rewritten version of Example 2-2 that prevents the DMAC from locking-up.

Example 2-3 Previous program rewritten to prevent the DMAC from locking-up

```
# Set up for 8-beat 32-bit transactions to both source and destination
   DMAMOV CCR, SS32 SB8 DS32 DB8
# Move data from source to destination
   DMALD
   DMAST
   DMALD
   DMAST
   DMALD
   DMAST
   DMALD
   DMAST
   DMAEND
```

In Example 2-3 on page 2-37, the load and store operations are interleaved and this enables the DMAC to empty the MFIFO and therefore it does not lock up.

Note

By using the loop instructions the program in Example 2-3 on page 2-37 can be written as:

```
# Set up transactions to both source and destination
DMAMOV CCR, SS32 SB8 DS32 DB16
# Move data from source to destination
DMALP 4
    DMALD
    DMAST
DMALPEND
DMAEND
```

Multiple DMA channels and lock-up situation

Example 2-4 shows a program that functions correctly when only a single DMA channel is active but causes the DMAC to lock-up when several DMA channels are active. The DMAC is configured with a read queue depth of 2 and an MFIFO depth of 8.

Example 2-4 Program that causes the DMAC to lock-up with multiple active DMA channels

```
# Set up transactions to both source and destination
DMAMOV CCR, SS32 SB8 DS32 DB16
# Move data from source to destination
DMALP 4
    DMALD(1)
    DMALD(2)
    DMAST
DMALPEND
DMAEND
```

When only a single DMA channel is active and executes the program in Example 2-4 on page 2-38 then:

1. The DMAC fetches the `DMALD`(1) into the read queue. It executes the instruction and half fills the MFIFO. After the instruction completes it removes the `DMALD`(1) from the read queue.

2. The DMAC fetches the `DMALD`(2) into the read queue. It executes the instruction and completely fills the MFIFO. After the instruction completes it removes the `DMALD`(2) from the read queue.

3. The DMAC fetches the `DMAST` into the write queue. It executes the instruction and empties the MFIFO. After the instruction completes it removes the `DMAST` from the write queue.

4. The DMAC repeats step 1 to step 3 for three more times until the loop count terminates and the DMA transaction completes successfully.

When multiple DMA channels are active and executing similar program code to that shown in Example 2-4 on page 2-38 then:

1. The DMAC fetches the `DMALD`(1) into the read queue. It executes the instruction and half fills the MFIFO. After the instruction completes it removes the `DMALD`(1) from the read queue.

2. The DMAC fetches the `DMALD`(2) into the read queue. It executes the instruction and completely fills the MFIFO. After the instruction completes it removes the `DMALD`(2) from the read queue.

If the DMAC arbitrates and another DMA channel starts to load data:

3. The DMAC fetches the `DMALD`(1) into the read queue. It executes the instruction but because the MFIFO is full the AXI read transaction does not complete and the instruction remains in the read queue.

4. The DMAC fetches the `DMALD`(2) into the read queue. It executes the instruction but because the MFIFO is full the AXI read transaction does not complete and the instruction remains in the read queue.

If the DMAC arbitrates and another DMA channel starts to load data:

5. The DMAC cannot fetch any more `DMALD` instructions because the read queue is full and the MFIFO remains full. The DMAC is locked-up and any outstanding DMA transactions fail to complete. The DMAC aborts the thread after the watchdog times out.

To prevent the DMAC from filling the MFIFO and causing a lock-up to occur you must ensure that the number of `DMALD` instructions that can be fetched before the DMAC fetches and executes a `DMAST` instruction is less than, or equal to, the read queue depth + number of complete bursts that the MFIFO can store. Figure 2-13 shows this as an equation.

$$\text{Number of consecutive DMALD instructions} \leq \text{Read queue depth} + \frac{\text{MFIFO depth}}{\text{source burst size}}$$

Figure 2-13 Equation for the maximum number of consecutive `DMALD`

For a DMAC that contains four active DMA channels that are executing similar program code to that shown in Example 2-4 on page 2-38 then using the equation in Figure 2-13 the read queue depth required to prevent lock-up occurring can be calculated as follows:

Consecutive `DMALD`	8
MFIFO depth	8
Source burst size	8, and therefore:
	Read queue depth 7, or larger.

Figure 2-14 on page 2-41 shows a DMAC operating with four active DMA channels that are each executing similar code to that shown in Example 2-4 on page 2-38. Using a read queue depth of 7 and an MFIFO depth of 8, the DMAC can execute the program without a lock-up condition occurring.

Note

- Figure 2-14 on page 2-41 shows a worst-case scenario where the DMAC arbitrates and selects a different DMA channel after executing every instruction.
- The scheduler uses a round-robin arbitration scheme but can bypass a DMA channel when the channel is not ready. Figure 2-14 on page 2-41 shows DMA channel 0 being bypassed after the DMAC executes the final `DMALD (1)` instruction.

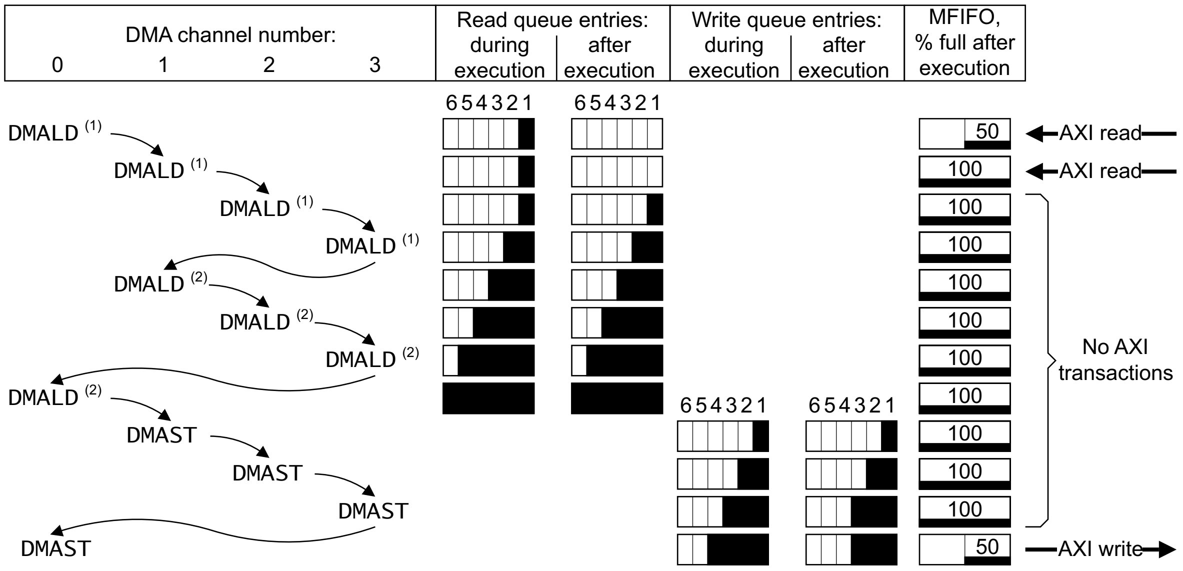


Figure 2-14 DMAC operating with four DMA channels

In Figure 2-14 it can be seen that the write queue depth must be configured to be 4 or greater. The write queue depth can be calculated using the equation shown in Figure 2-15.

$$\text{Number of consecutive DMAST instructions} \leq \text{Write LSQ depth} + \frac{\text{MFIFO depth}}{\text{destination burst size}}$$

Figure 2-15 Equation for the maximum number of consecutive `DMAST`

Note

In Figure 2-14, when the DMAC executes `DMAST` for DMA channel 0, it writes the destination data but it can only provide half of the intended data. This occurs because the DMAC could not complete the `DMALD (2)` as the MFIFO was full.

Chapter 3
Programmers Model

This chapter describes the memory map and registers of the DMAC. It contains the following sections:

- *About the programmers model* on page 3-2
- *DMAC Register summary* on page 3-4
- *DMAC Register descriptions* on page 3-11.

3.1 About the programmers model

The DMAC has 4KB of memory allocated to it from a base address of `0x000` to a maximum address of `0xFFF`. Figure 3-1 on page 3-3 shows that the register map address range is split into the following regions:

Control registers

Use these registers to control the DMAC.

DMA channel thread status registers

These registers provide the status of the DMA channel threads.

AXI and loop counter status registers

These registers provide the AXI bus transfer status and the loop counter status, for each DMA channel thread.

Debug registers

These registers enable:

- you to send instructions to a thread when debugging the program code
- system firmware to send instructions to the DMA manager thread as *Issuing instructions to the DMAC using an APB interface* on page 2-16 describes.

Configuration registers

These registers enable system firmware to discover the configuration of the DMAC.

PrimeCell ID registers

These registers enable system firmware to identify a PrimeCell component.

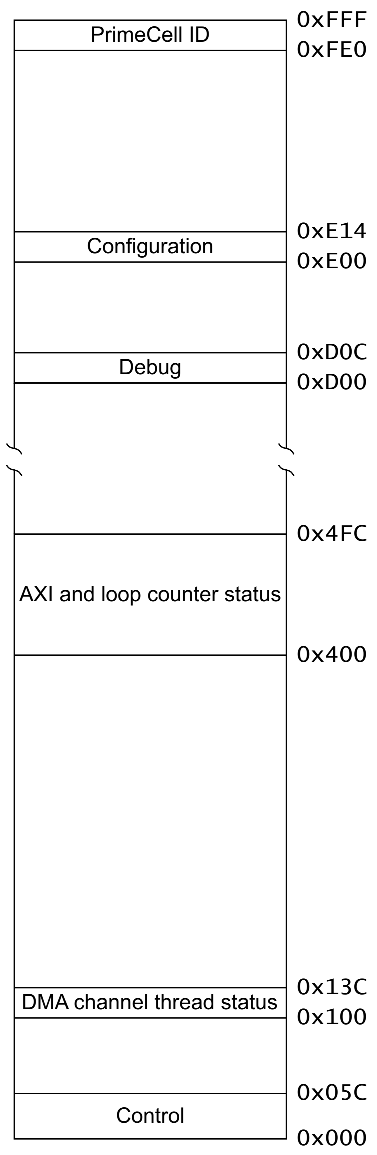


Figure 3-1 DMAC summary register map

3.2 DMAC Register summary

Table 3-1 lists the control registers and provides information about their address offsets, access permissions when using the secure and non-secure APB interfaces, and a brief description.

Table 3-1 DMAC Control Register summary

Name	Base offset	Secure R/W	Non-secure R/W when: thread is secure[a]	Non-secure R/W when: thread is non-secure[a]	Reset value	Description
DS	`0x000`	RO	*Read As Zero* (RAZ)	RO	`0x0`	*DMA Status Register* on page 3-11
DPC	`0x004`	RO	RAZ	RO	`0x0`	*DMA Program Counter Register* on page 3-12
-	`0x008` - `0x01C`	-	-	-	-	Reserved
INTEN	`0x020`	R/W	RAZ	R/W	`0x0`	*Interrupt Enable Register* on page 3-13
ES	`0x024`	RO	RAZ	RO	`0x0`	*Event Status Register* on page 3-14
INTSTATUS	`0x028`	RO	RAZ	RO	`0x0`	*Interrupt Status Register* on page 3-16
INTCLR	`0x02C`	WO	RAZ	WO	`0x0`	*Interrupt Clear Register* on page 3-17
FSM	`0x030`	RO	RAZ	RO	`0x0`	*Fault Status DMA Manager Register* on page 3-18
FSC	`0x034`	RO	RAZ	RO	`0x0`	*Fault Status DMA Channel Register* on page 3-19
FTM	`0x038`	RO	RAZ	RO	`0x0`	*Fault Type DMA Manager Register* on page 3-20
-	`0x03C`	-	-	-	-	Reserved

Table 3-1 DMAC Control Register summary (continued)

Name	Base offset	Secure R/W	Non-secure R/W when: thread is secure[a]	Non-secure R/W when: thread is non-secure[a]	Reset value	Description
***Fault Type DMA Channel Registers* on page 3-21**						
FTC0	`0x040`	RO	RAZ	RO	`0x0`	Fault type for DMA channel 0
FTC1	`0x044`					Fault type for DMA channel 1
FTC2	`0x048`					Fault type for DMA channel 2
FTC3	`0x04C`					Fault type for DMA channel 3
FTC4	`0x050`					Fault type for DMA channel 4
FTC5	`0x054`					Fault type for DMA channel 5
FTC6	`0x058`					Fault type for DMA channel 6
FTC7	`0x05C`					Fault type for DMA channel 7
-	`0x060` - `0x0FC`	-	-	-	-	Reserved

a. You must use the **boot_manager_ns** signal to set the security state for the DMA manager thread. See the *DMA Status Register* on page 3-11 for information about the security state of the DMA manager thread.

Table 3-2 lists the DMA channel thread status registers and provides information about their address offsets, access permissions when using the secure and non-secure APB interfaces, and a brief description.

Table 3-2 DMA channel thread status register summary

Name	Base offset	Secure R/W	Non-secure R/W when: channel is secure[a]	Non-secure R/W when: channel is non-secure[a]	Reset value	Description
***Channel Status Registers* on page 3-24**						
CS0	`0x100`	RO	RAZ	RO	`0x0`	Channel status for DMA channel 0
CS1	`0x108`					Channel status for DMA channel 1
CS2	`0x110`					Channel status for DMA channel 2
CS3	`0x118`					Channel status for DMA channel 3
CS4	`0x120`					Channel status for DMA channel 4
CS5	`0x128`					Channel status for DMA channel 5
CS6	`0x130`					Channel status for DMA channel 6
CS7	`0x138`					Channel status for DMA channel 7
***Channel Program Counter Registers* on page 3-26**						
CPC0	`0x104`	RO	RAZ	RO	`0x0`	Channel PC for DMA channel 0
CPC1	`0x10C`					Channel PC for DMA channel 1
CPC2	`0x114`					Channel PC for DMA channel 2
CPC3	`0x11C`					Channel PC for DMA channel 3
CPC4	`0x124`					Channel PC for DMA channel 4
CPC5	`0x12C`					Channel PC for DMA channel 5
CPC6	`0x134`					Channel PC for DMA channel 6
CPC7	`0x13C`					Channel PC for DMA channel 7
-	`0x140` - `0x3FC`	-	-	-	-	Reserved

a. The security setting for the channel is set by the security of the `DMAGO` instruction and the security state of the DMA manager thread. See the relevant *Channel Status Registers* on page 3-24 for information about the security state of the channel.

Table 3-3 lists the AXI status and loop counter registers and provides information about their address offsets, access permissions when using the secure and non-secure APB interfaces, and a brief description.

Table 3-3 AXI status and loop counter register summary

Name	Base offset	Secure R/W	Non-secure R/W when: channel is secure[a]	Non-secure R/W when: channel is non-secure[a]	Reset value	Description
***Source Address Registers* on page 3-27**						
SA_0	`0x400`	RO	RAZ	RO	`0x0`	Source address for DMA channel 0
SA_1	`0x420`					Source address for DMA channel 1
SA_2	`0x440`					Source address for DMA channel 2
SA_3	`0x460`					Source address for DMA channel 3
SA_4	`0x480`					Source address for DMA channel 4
SA_5	`0x4A0`					Source address for DMA channel 5
SA_6	`0x4C0`					Source address for DMA channel 6
SA_7	`0x4E0`					Source address for DMA channel 7
***Destination Address Registers* on page 3-29**						
DA_0	`0x404`	RO	RAZ	RO	`0x0`	Destination address for DMA channel 0
DA_1	`0x424`					Destination address for DMA channel 1
DA_2	`0x444`					Destination address for DMA channel 2
DA_3	`0x464`					Destination address for DMA channel 3
DA_4	`0x484`					Destination address for DMA channel 4
DA_5	`0x4A4`					Destination address for DMA channel 5
DA_6	`0x4C4`					Destination address for DMA channel 6
DA_7	`0x4E4`					Destination address for DMA channel 7
***Channel Control Registers* on page 3-30**						
CC_0	`0x408`	RO	RAZ	RO	`0x0`	Channel control for DMA channel 0
CC_1	`0x428`					Channel control for DMA channel 1
CC_2	`0x448`					Channel control for DMA channel 2
CC_3	`0x468`					Channel control for DMA channel 3
CC_4	`0x488`					Channel control for DMA channel 4
CC_5	`0x4A8`					Channel control for DMA channel 5
CC_6	`0x4C8`					Channel control for DMA channel 6
CC_7	`0x4E8`					Channel control for DMA channel 7

Table 3-3 AXI status and loop counter register summary (continued)

Name	Base offset	Secure R/W	Non-secure R/W when: channel is secure[a]	Non-secure R/W when: channel is non-secure[a]	Reset value	Description
***Loop Counter 0 Registers* on page 3-35**						
LC0_0	`0x40C`	RO	RAZ	RO	`0x0`	Loop counter 0 for DMA channel 0
LC0_1	`0x42C`					Loop counter 0 for DMA channel 1
LC0_2	`0x44C`					Loop counter 0 for DMA channel 2
LC0_3	`0x46C`					Loop counter 0 for DMA channel 3
LC0_4	`0x48C`					Loop counter 0 for DMA channel 4
LC0_5	`0x4AC`					Loop counter 0 for DMA channel 5
LC0_6	`0x4CC`					Loop counter 0 for DMA channel 6
LC0_7	`0x4EC`					Loop counter 0 for DMA channel 7
***Loop Counter 1 Registers* on page 3-36**						
LC1_0	`0x410`	RO	RAZ	RO	`0x0`	Loop counter 1 for DMA channel 0
LC1_1	`0x430`					Loop counter 1 for DMA channel 1
LC1_2	`0x450`					Loop counter 1 for DMA channel 2
LC1_3	`0x470`					Loop counter 1 for DMA channel 3
LC1_4	`0x490`					Loop counter 1 for DMA channel 4
LC1_5	`0x4B0`					Loop counter 1 for DMA channel 5
LC1_6	`0x4D0`					Loop counter 1 for DMA channel 6
LC1_7	`0x4F0`					Loop counter 1 for DMA channel 7
-	`0x414-0x41C`	-	-	-	-	Reserved
-	`0x434-0x43C`	-	-	-	-	Reserved
-	`0x454-0x45C`	-	-	-	-	Reserved
-	`0x474-0x47C`	-	-	-	-	Reserved
-	`0x494-0x49C`	-	-	-	-	Reserved
-	`0x4B4-0x4BC`	-	-	-	-	Reserved
-	`0x4D4-0x4DC`	-	-	-	-	Reserved
-	`0x4F4-0xCFC`	-	-	-	-	Reserved

a. The security setting for the channel is set by the security of the `DMAGO` instruction and the security state of the DMA manager thread. See the relevant *Channel Status Registers* on page 3-24 for information about the security state of the channel.

Table 3-4 lists the debug registers and provides information about their address offsets, access permissions when using the secure and non-secure APB interfaces, and a brief description.

Table 3-4 DMAC Debug Register summary

Name	Base offset	Secure R/W	Non-secure R/W when:		Reset value	Description
			thread is secure[a]	thread is non-secure[a]		
DBGSTATUS	`0xD00`	RO	RAZ	RO	`0x0`	*Debug Status Register* on page 3-37
DBGCMD	`0xD04`	WO	RAZ	WO	-	*Debug Command Register* on page 3-37
DBGINST0	`0xD08`	WO	RAZ	WO	-	*Debug Instruction-0 Register* on page 3-38
DBGINST1	`0xD0C`	WO	RAZ	WO	-	*Debug Instruction-1 Register* on page 3-39

a. You must use the **boot_manager_ns** signal to set the security state for the DMA manager thread. See the *DMA Status Register* on page 3-11 for information about the security state of the DMA manager thread.

Table 3-5 lists the configuration registers and provides information about their address offsets, access permissions when using the secure and non-secure APB interfaces, and a brief description.

Table 3-5 DMAC Configuration Register summary

Name	Base offset	Secure R/W	Non-secure R/W when:		Reset value	Description
			thread is secure[a]	thread is non-secure[a]		
CR0	`0xE00`	RO	RAZ	RO	-[b]	*Configuration Register 0* on page 3-40
CR1	`0xE04`	RO	RAZ	RO	-[b]	*Configuration Register 1* on page 3-42
CR2	`0xE08`	RO	RAZ	RO	-[b]	*Configuration Register 2* on page 3-43
CR3	`0xE0C`	RO	RAZ	RO	-[b]	*Configuration Register 3* on page 3-44
CR4	`0xE10`	RO	RAZ	RO	-[b]	*Configuration Register 4* on page 3-45
CRDn	`0xE14`	RO	RAZ	RO	-[b]	*Configuration Register Dn* on page 3-46

a. You must use the **boot_manager_ns** signal to set the security state for the DMA manager thread. See the *DMA Status Register* on page 3-11 for information about the security state of the DMA manager thread.

b. Configuration-dependent.

Table 3-6 lists the Peripheral Identification Registers and PrimeCell Identification Registers.

Table 3-6 Peripheral and PrimeCell Identification Register summary

Name	Base offset	Type	Reset value	Description
periph_id_n	`0xFE0-0xFEC`	RO	Configuration-dependent	*Peripheral Identification Registers 0-3* on page 3-48
pcell_id_n	`0xFF0-0xFFC`	RO	Configuration-dependent	*PrimeCell Identification Registers 0-3* on page 3-50

3.3 DMAC Register descriptions

This section describes the registers that the DMAC contains.

3.3.1 DMA Status Register

The DS Register provides information about the configuration and current state of the DMAC. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-2 shows the register bit assignments.

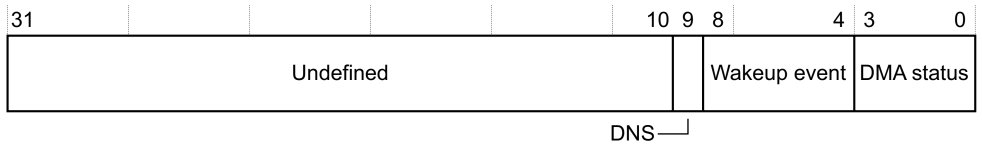


Figure 3-2 DS Register bit assignments

Table 3-7 lists the register bit assignments.

Table 3-7 DS Register bit assignments

Bits	Name	Function
[31:10]	-	Read undefined.

Table 3-7 DS Register bit assignments (continued)

Bits	Name	Function
[9]	DNS	Provides the secure state of the DMA manager thread: 0 = DMA manager operates in the Secure state 1 = DMA manager operates in the Non-secure state. **Note** You must use the **boot_manager_ns** signal to set the secure state of the DMA manager thread.

[8:4]	Wakeup_event	When the DMAC executes a `DMAWFE` instruction it waits for the following event to occur:
		b00000 = event[0]
		b00001 = event[1]
		b00010 = event[2]
		.
		.
		.
		b11111 = event[31].

[3:0]	DMA status	The operating state of the DMA manager:
		b0000 = Stopped
		b0001 = Executing
		b0010 = Cache miss
		b0011 = Updating PC
		b0100 = Waiting for event
		b0101-b1110 = reserved
		b1111 = Faulting.
		See *Operating states* on page 2-10 for more information.

3.3.2 DMA Program Counter Register

The DPC Register provides the value of the program counter for the DMA manager thread. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-3 on page 3-13 shows the register bit assignments.

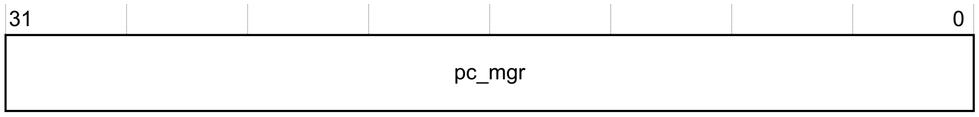


Figure 3-3 DPC Register bit assignments

Table 3-8 lists the register bit assignments.

Table 3-8 DPC Register bit assignments

Bits	Name	Function
[31:0]	pc_mgr	Program counter for the DMA manager thread

3.3.3 Interrupt Enable Register

When the DMAC executes a `DMASEV` instruction, each bit of the INTEN Register controls if the DMAC signals:

- the specified event to all of the threads
- an interrupt using the corresponding **irq**.

Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-4 shows the register bit assignments.

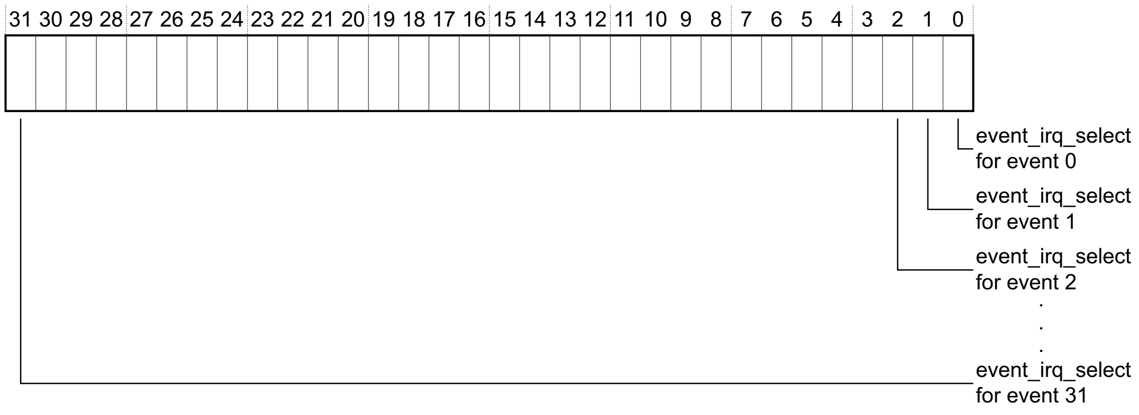

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
event_irq_select for event 0
event_irq_select for event 1
event_irq_select for event 2
event_irq_select for event 31

Figure 3-4 INTEN Register bit assignments

Table 3-9 lists the register bit assignments.

Table 3-9 INTEN Register bit assignments

Bits	Name	Function
[31:0]	event_irq_select	Program the appropriate bit to control how the DMAC responds when it executes `DMASEV`: **Bit [*N*] = 0** If executing `DMASEV` for event *N* then the DMAC signals event *N* to all of the threads. **Bit [*N*] = 1** If executing `DMASEV` for event *N* then the DMAC sets **irq[N]** HIGH. **Note** See *DMASEV* on page 4-18 for information about selecting an event number.

3.3.4 Event Status Register

The ES Register provides the status of the event/interrupt requests that are active in the DMAC. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Note

The DMAC only generates an event request when a thread executes a `DMASEV` instruction.

Figure 3-5 shows the register bit assignments.

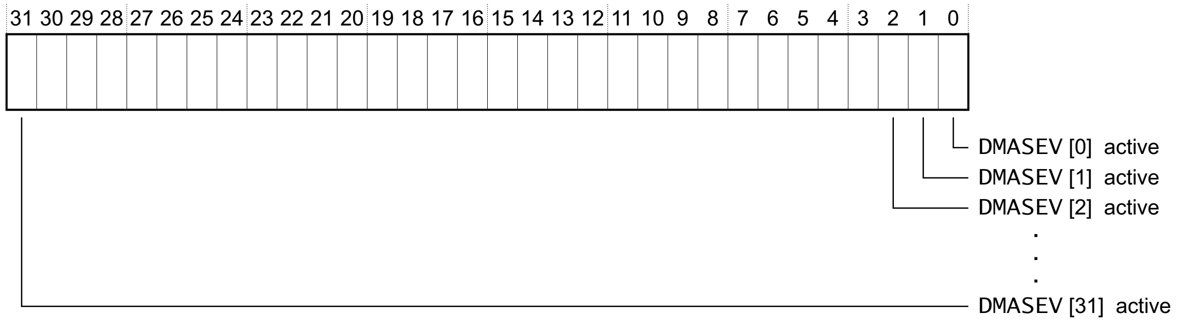


Figure 3-5 ES Register bit assignments

Table 3-10 on page 3-15 lists the register bit assignments.

Table 3-10 ES Register bit assignments

Bits	Name	Function
[31:0]	`DMASEV` active	Provides the status of the events and interrupts that are active in the DMAC: **Bit [*N*] = 0** Event *N* is inactive or **irq[N]** is LOW. **Bit [*N*] = 1** Event *N* is active or **irq[N]** is HIGH. **Note** When the DMAC receives an event request, the *Interrupt Enable Register* on page 3-13 controls if the DMAC signals: • an interrupt using the appropriate **irq** • the event to all of the threads. **Note** The DMAC clears bit [*N*] when either: • the INTEN Register is programmed to process the event and the DMAC executes a `DMAWFE` instruction for that event • the INTEN Register is programmed to signal an interrupt and you write to the corresponding bit in the *Interrupt Clear Register* on page 3-17.

3.3.5 Interrupt Status Register

The INTSTATUS Register provides the status of the active interrupts in the DMAC. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-6 shows the register bit assignments.

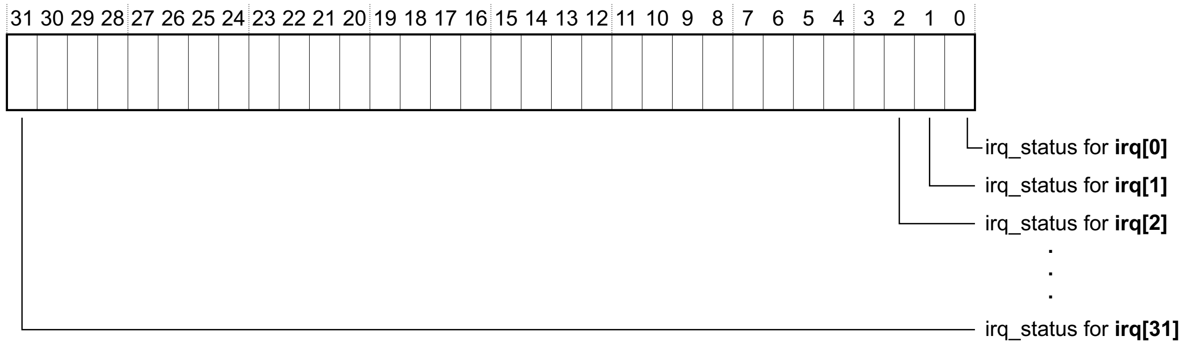

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
irq_status for **irq[0]**
irq_status for **irq[1]**
irq_status for **irq[2]**
irq_status for **irq[31]**

Figure 3-6 INTSTATUS Register bit assignments

Table 3-11 lists the register bit assignments.

Table 3-11 INTSTATUS Register bit assignments

Bits	Name	Function
[31:0]	irq_status	Provides the status of the interrupts that are active in the DMAC: **Bit [*N*] = 0** Interrupt *N* is inactive and therefore **irq[N]** is LOW. **Bit [*N*] = 1** Interrupt *N* is active and therefore **irq[N]** is HIGH. **Note** You must use the *Interrupt Clear Register* on page 3-17 to set bit [*N*] to 0. **Note** Bit [*N*] is 0 if the *Interrupt Enable Register* on page 3-13 programs `DMASEV` to signal an event.

3.3.6 Interrupt Clear Register

Each bit in the INTCLR Register controls the clearing of an interrupt. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-7 shows the register bit assignments.

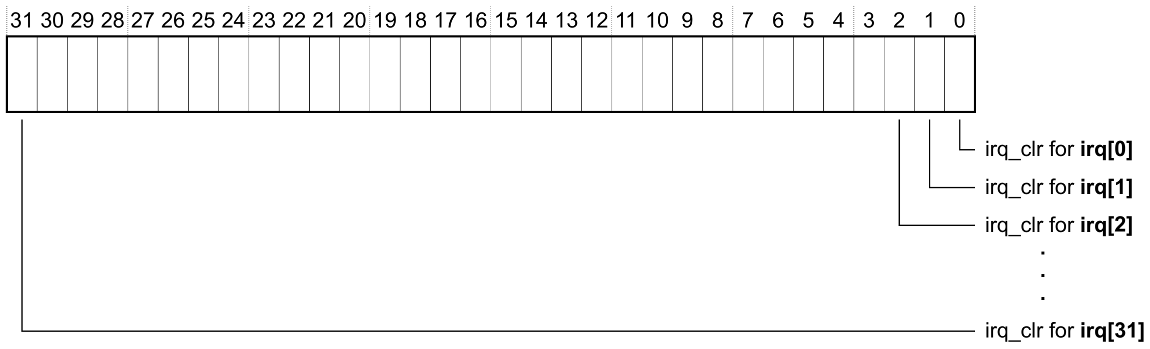


Figure 3-7 INTCLR Register bit assignments

Table 3-12 lists the register bit assignments.

Table 3-12 INTCLR Register bit assignments

Bits	Name	Function
[31:0]	irq_clr	Controls the clearing of the **irq** outputs: **Bit [*N*] = 0** The status of **irq[N]** does not change. **Bit [*N*] = 1** The DMAC sets **irq[N]** LOW if *Interrupt Enable Register* on page 3-13 programs the DMAC to signal an interrupt. Otherwise the status of **irq[N]** does not change.

3.3.7 Fault Status DMA Manager Register

The FSM Register provides the fault status of the DMA manager. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-8 shows the register bit assignments.

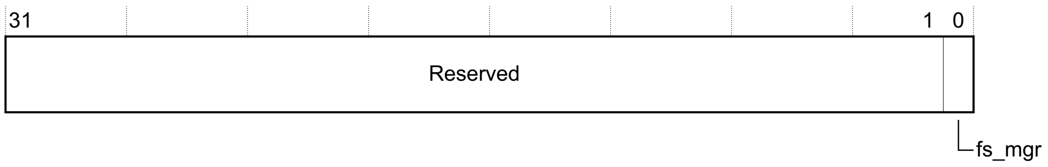


Figure 3-8 FSM Register bit assignments

Table 3-13 lists the register bit assignments.

Table 3-13 FSM Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined.
{0]	fs_mgr	Provides the fault status of the DMA manager. Read as: 0 = the DMA manager thread is not in the Faulting state 1 = the DMA manager thread is in the Faulting state. See *Fault Type DMA Manager Register* on page 3-20 for information about the type of fault that occurred.

3.3.8 Fault Status DMA Channel Register

The FSC Register provides the fault status for the DMA channels. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-9 shows the register bit assignments.

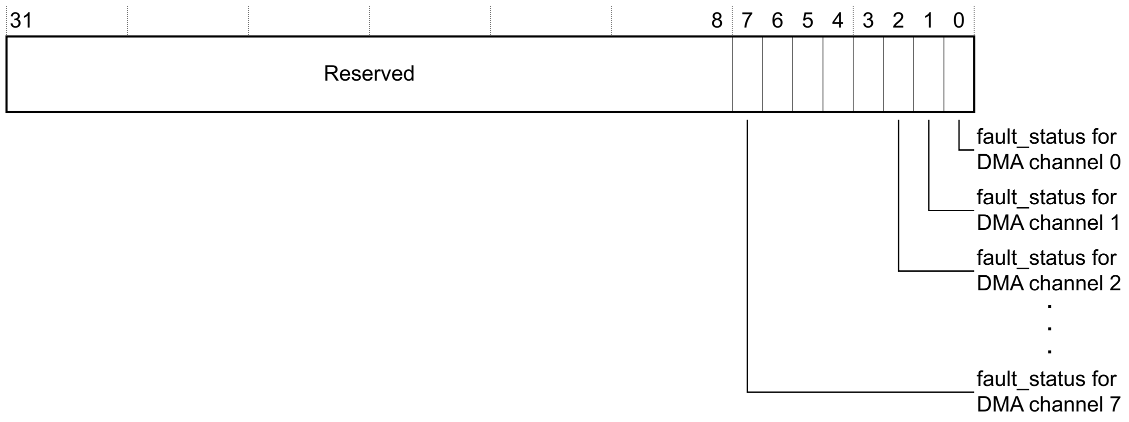

31
8
7
6
5
4
3
2
1
0
Reserved
fault_status for DMA channel 0
fault_status for DMA channel 1
fault_status for DMA channel 2
fault_status for DMA channel 7

Figure 3-9 FSC Register bit assignments

Table 3-14 lists the register bit assignments.

Table 3-14 FSC Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined.
[7:0]	fault_status	Each bit provides the fault status of the corresponding channel. Read as: **Bit [*N*] = 0** No fault is present on DMA channel *N*. **Bit [*N*] = 1** DMA channel *N* is in the Faulting or Faulting completing state. See *Fault Type DMA Channel Registers* on page 3-21 for information about the type of fault that occurred.

3.3.9 Fault Type DMA Manager Register

The FTM Register provides the type of fault that occurred to move the DMA manager to the Faulting state. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-10 shows the register bit assignments.

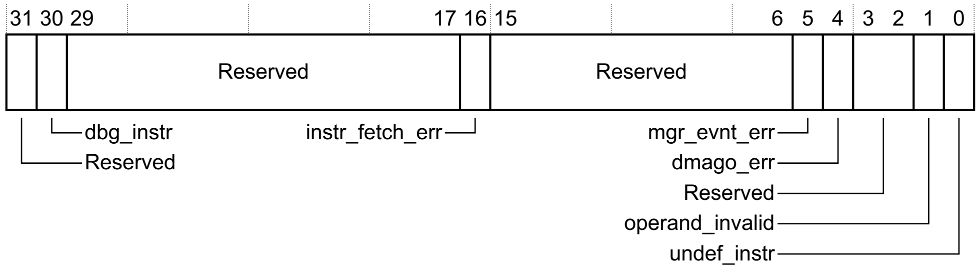


Figure 3-10 FTM Register bit assignments

Table 3-15 lists the register bit assignments.

Table 3-15 FTM Register bit assignments

Bits	Name	Function
[31]	-	Read undefined.
[30]	dbg_instr	If the DMA manager aborts then this bit indicates if the erroneous instruction was read from the debug interface: 0 = instruction that generated an abort was read from system memory 1 = instruction that generated an abort was read from the debug interface.
[29:17]	-	Read undefined.
[16]	instr_fetch_err	Indicates the AXI response that the DMAC receives on the **RRESP** or **BRESP** buses, after the DMA manager performs an instruction fetch: 0 = OKAY response 1 = EXOKAY, SLVERR, or DECERR response.
[15:6]	-	Read undefined.

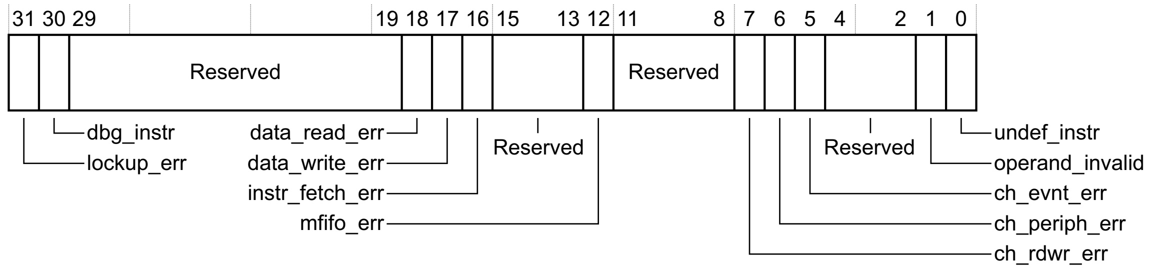
Table 3-15 FTM Register bit assignments (continued)

Bits	Name	Function
[5]	mgr_evnt_err	Indicates if the DMA manager was attempting to execute `DMAWFE` or `DMASEV` with inappropriate security permissions: 0 = DMA manager has appropriate security to execute `DMAWFE` or `DMASEV` 1 = a DMA manager thread in the Non-secure state attempted to execute either: • `DMAWFE` to wait for a secure event • `DMASEV` to create a secure event or secure interrupt.
[4]	dmago_err	Indicates if the DMA manager was attempting to execute `DMAGO` with inappropriate security permissions: 0 = DMA manager has appropriate security to execute `DMAGO` 1 = a DMA manager thread in the Non-secure state attempted to execute `DMAGO` to create a DMA channel operating in the Secure state.
[3:2]	-	Read undefined.
[1]	operand_invalid	Indicates if the DMA manager was attempting to execute an instruction operand that was not valid for the configuration of the DMAC: 0 = valid operand 1 = invalid operand.
[0]	undef_instr	Indicates if the DMA manager was attempting to execute an undefined instruction: 0 = defined instruction 1 = undefined instruction.

3.3.10 Fault Type DMA Channel Registers

An FTC Register provides the type of fault that occurred to move a DMA channel to the Faulting state. The DMAC provides a FTC*n* Register for each DMA channel that it contains. Table 3-1 on page 3-4 lists the address base offset, reset value, and access type for this register.

Figure 3-11 on page 3-22 shows the register bit assignments and the address base offsets for each FTC*n* Register.



FTC<n> Register address mapping:

Channel <n>	0	1	2	3	4	5	6	7
Address offset	0x040	0x044	0x048	0x04C	0x050	0x054	0x058	0x05C

Figure 3-11 FTC Register bit assignments

Table 3-16 lists the register bit assignments.

Table 3-16 FTC Register bit assignments

Bits	Name	Function
[31]	lockup_err	Indicates if the DMA channel has locked-up due to resource starvation: 0 = DMA channel has adequate resources 1 = DMA channel has locked-up due to insufficient resources.
[30]	dbg_instr	If the DMA channel aborts then this bit indicates if the erroneous instruction was read from the debug interface: 0 = instruction that generated an abort was read from system memory 1 = instruction that generated an abort was read from the debug interface.
[29:19]	-	Reserved, read undefined.
[18]	data_read_err	Indicates the AXI response that the DMAC receives on the **RRESP** or **BRESP** buses, after the DMA channel thread performs a data read: 0 = OKAY response 1 = EXOKAY, SLVERR, or DECERR response.
[17]	data_write_err	Indicates the AXI response that the DMAC receives on the **RRESP** or **BRESP** buses, after the DMA channel thread performs a data write: 0 = OKAY response 1 = EXOKAY, SLVERR, or DECERR response.

Table 3-16 FTC Register bit assignments (continued)

Bits	Name	Function
[16]	instr_fetch_err	Indicates the AXI response that the DMAC receives on the **RRESP** or **BRESP** buses, after the DMA channel thread performs an instruction fetch: 0 = OKAY response 1 = EXOKAY, SLVERR, or DECERR response.
[15:13]	-	Reserved, read undefined.
[12]	mfifo_err	Indicates if the MFIFO prevented the DMA channel thread from executing `DMALD` or `DMAST`. Depending on the instruction: `DMALD` 0 = MFIFO has sufficient space 1 = MFIFO has insufficient space to store the date that `DMALD` requires. `DMAST` 0 = MFIFO contains sufficient data 1 = MFIFO is too small to store the data to enable `DMAST` to complete.
[11:8]	-	Reserved, read undefined.
[7]	ch_rdwr_err	Indicates if a DMA channel thread in the Non-secure state was attempting to perform a secure read or secure write: 0 = a DMA channel thread in the Non-secure state is not violating the security permissions 1 = a DMA channel thread in the Non-secure state attempted to perform a secure read or secure write.
[6]	ch_periph_err	Indicates if a DMA channel thread in the Non-secure state was attempting to execute `DMAWFP`, `DMALDP`, `DMASTP`, or `DMAFLUSHP` with inappropriate security permissions: 0 = a DMA channel thread in the Non-secure state is not violating the security permissions 1 = a DMA channel thread in the Non-secure state attempted to execute either: • `DMAWFP` to wait for a secure peripheral • `DMALDP` or `DMASTP` to notify a secure peripheral • `DMAFLUSHP` to flush a secure peripheral.
[5]	ch_evnt_err	Indicates if the DMA channel thread was attempting to execute `DMAWFE` or `DMASEV` with inappropriate security permissions: 0 = a DMA channel thread in the Non-secure state is not violating the security permissions 1 = a DMA channel thread in the Non-secure state attempted to execute either: • `DMAWFE` to wait for a secure event • `DMASEV` to create a secure event or secure interrupt.

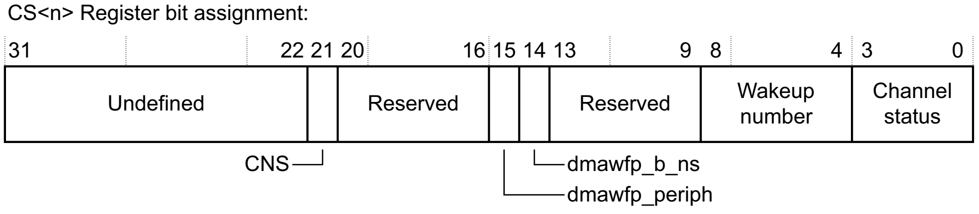
Table 3-16 FTC Register bit assignments (continued)

Bits	Name	Function
[4:2]	-	Reserved, read undefined.
[1]	operand_invalid	Indicates if the DMA channel thread was attempting to execute an instruction operand that was not valid for the configuration of the DMAC: 0 = valid operand 1 = invalid operand.
[0]	undef_instr	Indicates if the DMA channel thread was attempting to execute an undefined instruction: 0 = defined instruction 1 = undefined instruction.

3.3.11 Channel Status Registers

A CS Register provides the status of the DMA program on a DMA channel. The DMAC provides a CS*n* Register for each DMA channel that it contains. Table 3-2 on page 3-6 lists the reset value and access type for this register.

Figure 3-12 shows the register bit assignments and the address base offsets for each CS*n* Register.



CS<n> Register address mapping:

Channel <n>	0	1	2	3	4	5	6	7
Address offset	0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138

Figure 3-12 CS*n* Register bit assignments and address offsets

Table 3-17 lists the register bit assignments.

Table 3-17 CS*n* Register bit assignments

Bits	Name	Function
[31:22]	-	Reserved, read undefined.
[21]	CNS	The channel non-secure bit provides the security of the DMA channel: 0 = DMA channel operates in the Secure state 1 = DMA channel operates in the Non-secure state. **Note** The `DMAGO` instruction determines the security state of a DMA channel. See *DMAGO* on page 4-6.
[20:16]	-	Reserved, read undefined.
[15]	dmawfp_periph	When the DMA channel thread executes *DMAWFP<S|B|P>* on page 4-22 this bit indicates if the p operand was set: 0 = `DMAWFP` executed with the p operand not set 1 = `DMAWFP` executed with the p operand set.
[14]	dmawfp_b_ns	When the DMA channel thread executes *DMAWFP<S|B|P>* on page 4-22 this bit indicates if the b or s operand were set: 0 = `DMAWFP` executed with the s operand set 1 = `DMAWFP` executed with the b operand set.

Table 3-17 CS*n* Register bit assignments (continued)

Bits	Name	Function
[13:9]	-	Reserved, read undefined.
[8:4]	Wakeup number	If the DMA channel is in the Waiting for event state or the Waiting for peripheral state then these bits indicate the event or peripheral number that the channel is waiting for: b00000 = DMA channel is waiting for event, or peripheral, 0 b00001 = DMA channel is waiting for event, or peripheral, 1 b00010 = DMA channel is waiting for event, or peripheral, 2 . . . b11111 = DMA channel is waiting for event, or peripheral, 31.

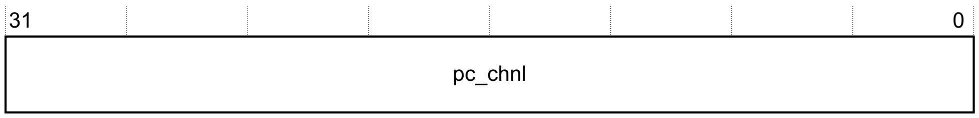
[3:0]	Channel status	The channel status encoding is:
		b0000 = Stopped
		b0001 = Executing
		b0010 = Cache miss
		b0011 = Updating PC
		b0100 = Waiting for event
		b0101 = At barrier
		b0110 = Queue busy
		b0111 = Waiting for peripheral
		b1000 = Killing
		b1001 = Completing
		b1010-b1101 = reserved
		b1110 = Faulting completing
		b1111 = Faulting.
		See *Operating states* on page 2-10 for more information.

3.3.12 Channel Program Counter Registers

A CPC Register provides the value of the program counter for the DMA channel thread. The DMAC provides a CPC*n* Register for each DMA channel that it contains. Table 3-2 on page 3-6 lists the reset value and access type for this register.

Figure 3-13 on page 3-27 shows the register bit assignments and the address base offsets for each CPC*n* Register.

CPC*n* Register bit assignment:



CPC*n* Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Address offset	0x104	0x10C	0x114	0x11C	0x124	0x12C	0x134	0x13C

Figure 3-13 CPC*n* Register bit assignments and address offsets

Table 3-18 lists the register bit assignments.

Table 3-18 CPC*n* Register bit assignments

Bits	Name	Function
[31:0]	pc_chnl	Program counter for the DMA channel *n* thread, where *n* depends on the address of the register as Figure 3-13 shows.

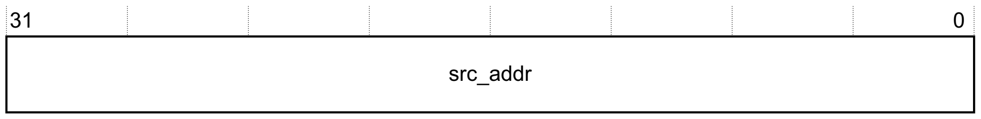
3.3.13 Source Address Registers

An SA Register provides the address of the source data for a DMA channel. The DMAC provides a SA_*n* Register for each DMA channel that it contains. Table 3-3 on page 3-7 lists the reset value and access type for this register.

The DMAC writes the initial source address value to the SA Register when the DMA channel thread executes a `DMAMOV SAR` instruction. If a subsequent `DMAMOV CCR` instruction programs the source address to increment then each time the DMA channel executes `DMALD` it updates the value to indicate the address that the next `DMALD` must use. See *DMAMOV* on page 4-15 for more information.

Figure 3-14 on page 3-28 shows the register bit assignments and the address base offsets for each SA_*n* Register.

SA_*n* Register bit assignments:



Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Register name	SA_0	SA_1	SA_2	SA_3	SA_4	SA_5	SA_6	SA_7
Address offset	0x400	0x420	0x440	0x460	0x480	0x4A0	0x4C0	0x4E0

Figure 3-14 SA_*n* Register bit assignments and address offsets

Table 3-19 lists the register bit assignments.

Table 3-19 SA_*n* Register bit assignments

Bits	Name	Function
[31:0]	src_addr	Address of the source data for DMA channel *n*, where *n* depends on the address of the register as Figure 3-14 shows.

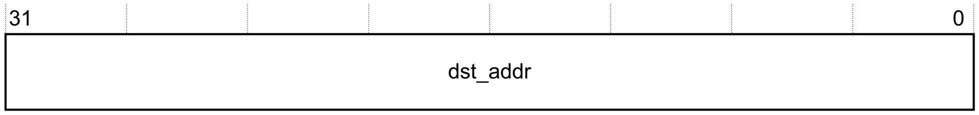
3.3.14 Destination Address Registers

A DA Register provides the address for the destination data for a DMA channel. The DMAC provides a DA_*n* Register for each DMA channel that it contains. Table 3-3 on page 3-7 lists the reset value and access type for this register.

The DMAC writes the initial destination address value to the DA Register when the DMA channel thread executes a `DMAMOV DAR` instruction. If a subsequent `DMAMOV CCR` instruction programs the destination address to increment then each time the DMA channel executes `DMAST` it updates the value to indicate the address that the next `DMAST` must use. See *DMAMOV* on page 4-15 for more information.

Figure 3-15 shows the register bit assignments and the address base offsets for each DA_*n* Register.

DA_*n* Register bit assignments:



Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Register name	DA_0	DA_1	DA_2	DA_3	DA_4	DA_5	DA_6	DA_7
Address offset	`0x404`	`0x424`	`0x444`	`0x464`	`0x484`	`0x4A4`	`0x4C4`	`0x4E4`

Figure 3-15 DA_*n* Register bit assignments and address offsets

Table 3-20 lists the register bit assignments.

Table 3-20 DA_*n* Register bit assignments

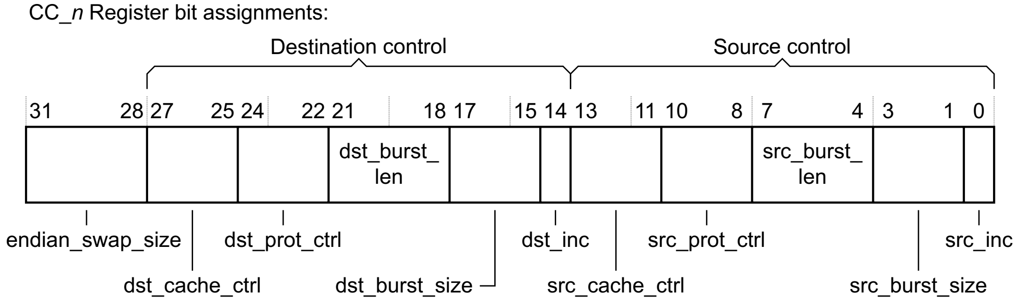
Bits	Name	Function
[31:0]	dst_addr	Address for the destination data for DMA channel *n*, where *n* depends on the address of the register as Figure 3-15 shows.

3.3.15 Channel Control Registers

A CC Register controls the AXI transactions that the DMAC uses for a DMA channel. The DMAC provides a CC_*n* Register for each DMA channel that it contains. Table 3-3 on page 3-7 lists the reset value and access type for this register.

The DMAC writes to the corresponding CC Register when a DMA channel thread executes a `DMAMOV CCR` instruction.

Figure 3-16 shows the register bit assignments and the address base offsets for each CC_*n* Register.



Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Register name	CC_0	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
Address offset	0x408	0x428	0x448	0x468	0x488	0x4A8	0x4C8	0x4E8

Figure 3-16 CC_*n* Register bit assignments

Table 3-21 lists the register bit assignments.

Table 3-21 CC_*n* Register bit assignments

Bits	Name	Function
[31:28]	endian_swap_size	See *Endian swap size* on page 3-35.
[27:25]	dst_cache_ctrl	Programs the state of **AWCACHE[3,1:0]**[a] when the DMAC writes the destination data. **Bit [27]** 0 = **AWCACHE[3]** is LOW 1 = **AWCACHE[3]** is HIGH. **Bit [26]** 0 = **AWCACHE[1]** is LOW 1 = **AWCACHE[1]** is HIGH. **Bit [25]** 0 = **AWCACHE[0]** is LOW 1 = **AWCACHE[0]** is HIGH. ——— **Note** ——— **AWCACHE[2]** is tied LOW by the DMAC. Setting **AWCACHE[3,1]**=b10 violates the AXI protocol. See the *AMBA AXI Protocol Specification*. ———————
[24:22]	dst_prot_ctrl	Programs the state of **AWPROT[2:0]**[a] when the DMAC writes the destination data. **Bit [24]** 0 = **AWPROT[2]** is LOW 1 = **AWPROT[2]** is HIGH. **Bit [23]** 0 = **AWPROT[1]** is LOW 1 = **AWPROT[1]** is HIGH. **Bit [22]** 0 = **AWPROT[0]** is LOW 1 = **AWPROT[0]** is HIGH. ——— **Note** ——— Only DMA channels in the Secure state can program **AWPROT[1]** LOW, that is a secure access. If a DMA channel in the Non-secure state attempts to set **AWPROT[1]** LOW then the DMA channel aborts. ———————

Table 3-21 CC_*n* Register bit assignments (continued)

Bits	Name	Function
[21:18]	dst_burst_len	For each burst, these bits program the number of data transfers that the DMAC performs when it writes the destination data: b0000 = 1 data transfer b0001 = 2 data transfers b0010 = 3 data transfers . . . b1111 = 16 data transfers. **Note** These bits control the status of **AWLEN[3:0]**.

[17:15]	dst_burst_size	Programs the burst size that the DMAC uses when it writes the destination data:
		b000 = 1 byte
		b001 = 2 bytes
		b010 = 4 bytes
		b011 = 8 bytes
		b100 = 16 bytes
		b101 = 32 bytes
		b110 = 64 bytes
		b111 = 128 bytes.
		Note
		These bits control the status of **AWSIZE[2:0]**.

[14]	dst_inc	Programs the burst type that the DMAC performs when it writes the destination data: 0 = fixed-address burst. The DMAC signals **AWBURST[0]** LOW. 1 = incrementing-address burst. The DMAC signals **AWBURST[0]** HIGH.

Table 3-21 CC_*n* Register bit assignments (continued)

Bits	Name	Function
[13:11]	src_cache_ctrl	Set the bits to control the state of **ARCACHE[2:0]**[a] when the DMAC reads the source data. **Bit [13]** 0 = **ARCACHE[2]** is LOW 1 = **ARCACHE[2]** is HIGH. **Bit [12]** 0 = **ARCACHE[1]** is LOW 1 = **ARCACHE[1]** is HIGH. **Bit [11]** 0 = **ARCACHE[0]** is LOW 1 = **ARCACHE[0]** is HIGH. **Note** **ARCACHE[3]** is tied LOW by the DMAC. Setting **ARCACHE[2:1]**=b10 violates the AXI protocol.
[10:8]	src_prot_ctrl	Programs the state of **ARPROT[2:0]**[a] when the DMAC reads the source data. **Bit [24]** 0 = **ARPROT[2]** is LOW 1 = **ARPROT[2]** is HIGH. **Bit [23]** 0 = **ARPROT[1]** is LOW 1 = **ARPROT[1]** is HIGH. **Bit [22]** 0 = **ARPROT[0]** is LOW 1 = **ARPROT[0]** is HIGH. **Note** Only DMA channels in the Secure state can program **ARPROT[1]** LOW, that is a secure access. If a DMA channel in the Non-secure state attempts to set **ARPROT[1]** LOW then the DMA channel aborts.

Table 3-21 CC_*n* Register bit assignments (continued)

Bits	Name	Function
[7:4]	src_burst_len	For each burst, these bits program the number of data transfers that the DMAC performs when it reads the source data: b0000 = 1 data transfer b0001 = 2 data transfers b0010 = 3 data transfers . . . b1111 = 16 data transfers. **Note** These bits control the status of **ARLEN[3:0]**.

[3:1]	src_burst_size	Programs the burst size that the DMAC uses when it reads the source data:
		b000 = 1 byte
		b001 = 2 bytes
		b010 = 4 bytes
		b011 = 8 bytes
		b100 = 16 bytes
		b101 = 32 bytes
		b110 = 64 bytes
		b111 = 128 bytes.
		Note
		These bits control the status of **ARSIZE[2:0]**.

[0]	src_inc	Programs the burst type that the DMAC performs when it reads the source data: 0 = fixed-address burst. The DMAC signals **ARBURST[0]** LOW. 1 = incrementing-address burst. The DMAC signals **ARBURST[0]** HIGH.

a. See the *AMBA AXI Protocol Specification* for information about this AXI signal.

Note

The DMAC does not generate:

- locked or exclusive accesses.
- WRAP transfers. Therefore, **ARBURST[1]** and **AWBURST[1]** are always LOW.

Endian swap size

Table 3-22 defines whether data can be swapped between *little-endian* (LE) and byte-invariant *big-endian* (BE-8) formats, and if so, also defines the natural width of the data independently of the source and destination transaction sizes. This enables unaligned data streams to use the full bus-width and to be correctly transformed irrespective of the source and destination address alignments. The format is identical to **AxSIZE**, except that b000 indicates that no swap must occur.

Table 3-22 Swap data

Endian swap size	Description
b000	No swap, 8-bit data
b001	Swap bytes within 16-bit data
b010	Swap bytes within 32-bit data
b011	Swap bytes within 64-bit data
b100	Swap bytes within 128-bit data
b101	Reserved
b110	Reserved
b111	Reserved

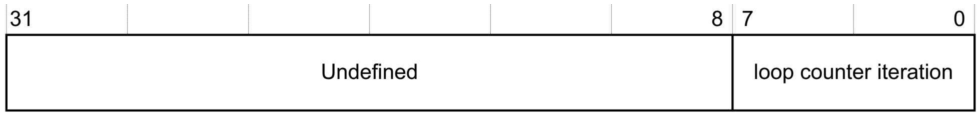
Note

See *Endian swap size restrictions* on page 2-35 for information about some restrictions that apply when you use this feature.

3.3.16 Loop Counter 0 Registers

An LC0 Register provides the status of loop counter zero for the DMA channel. The DMAC updates this register when it executes *DMALPEND[S|B]* on page 4-11 and the DMA channel thread is programmed to use loop counter zero. The DMAC provides a LC0_*n* Register for each DMA channel that it contains. Table 3-3 on page 3-7 lists the reset value and access type for this register.

Figure 3-17 on page 3-36 shows the register bit assignments and the address base offsets for each LC0_*n* Register.



Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Register name	LC0_0	LC0_1	LC0_2	LC0_3	LC0_4	LC0_5	LC0_6	LC0_7
Address offset	0x40C	0x42C	0x44C	0x46C	0x4CC	0x4AC	0x4CC	0x4EC

Figure 3-17 LC0_*n* Register bit assignments

Table 3-23 lists the register bit assignments.

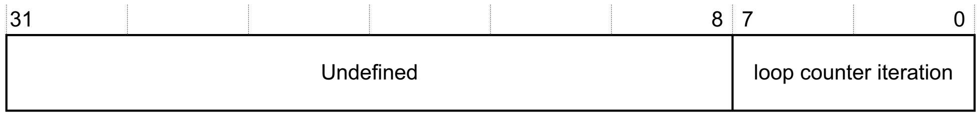
Table 3-23 LC0_*n* Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	Loop counter iterations	Loop counter iterations

3.3.17 Loop Counter 1 Registers

An LC1 Register provides the status of loop counter one for the DMA channel. The DMAC updates this register when it executes *DMALPEND[S|B]* on page 4-11 and the DMA channel thread is programmed to use loop counter one. The DMAC provides a LC1_*n* Register for each DMA channel that it contains. Table 3-3 on page 3-7 lists the reset value and access type for this register.

Figure 3-18 shows the register bit assignments and the address base offsets for each LC1_*n* Register.



Register address mapping:

Channel *n*	0	1	2	3	4	5	6	7
Register name	LC1_0	LC1_1	LC1_2	LC1_3	LC1_4	LC1_5	LC1_6	LC1_7
Address offset	0x410	0x430	0x450	0x470	0x490	0x4B0	0x4D0	0x4F0

Figure 3-18 LC1_*n* Register bit assignments

Table 3-24 lists the register bit assignments.

Table 3-24 LC1_*n* Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	Loop counter iterations	Loop counter iterations

3.3.18 Debug Status Register

The DBGSTATUS Register provides the debug status of the DMAC. Table 3-4 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-19 shows the register bit assignments.

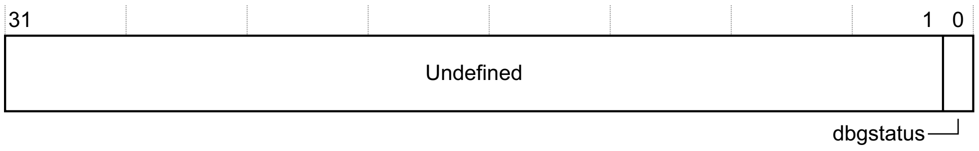


Figure 3-19 DBGSTATUS Register bit assignments

Table 3-25 lists the register bit assignments.

Table 3-25 DBGSTATUS Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined.
[0]	dbgstatus	The debug status encoding is: 0 = Idle 1 = Busy.

3.3.19 Debug Command Register

The DBGCMD Register controls the execution of debug commands in the DMAC as *Issuing instructions to the DMAC using an APB interface* on page 2-16 describes. Table 3-4 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-20 on page 3-38 shows the register bit assignments.

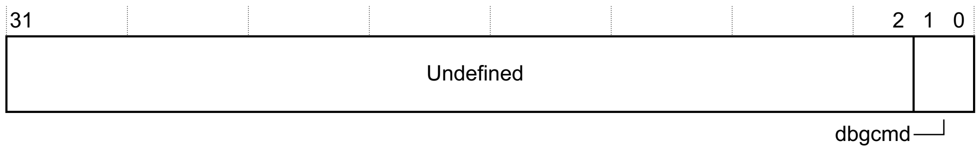


Figure 3-20 DBGCMD Register bit assignments

Table 3-26 lists the register bit assignments.

Table 3-26 DBGCMD Register bit assignments

Bits	Name	Function
[31:2]	-	Reserved. Write as zero.

[1:0]	dbgcmd	The debug encoding is as follows:
		b00 = execute the instruction that the DBGINST [1:0] Registers contain
		b01 = reserved
		b10 = reserved
		b11 = reserved.

3.3.20 Debug Instruction-0 Register

The DBGINST0 Register controls the debug instruction, channel, and thread information for the DMAC. See *Issuing instructions to the DMAC using an APB interface* on page 2-16 for more information. Table 3-4 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-21 shows the register bit assignments.

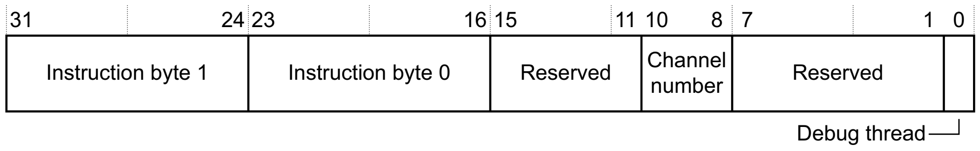


Figure 3-21 DBGINST0 Register bit assignments

Table 3-27 lists the register bit assignments.

Table 3-27 DBGINST0 Register bit assignments

Bits	Name	Function
[31:24]	Instruction byte 1	Instruction byte 1.
[23:16]	Instruction byte 0	Instruction byte 0.
[15:11]	-	Reserved. Write as zero.

[10:8]	Channel number	DMA channel number:
		b000 = DMA channel 0
		b001 = DMA channel 1
		b010 = DMA channel 2
		.
		.
		.
		b111 = DMA channel 7.

[7:1]	-	Reserved. Write as zero.
[0]	Debug thread	The debug thread encoding is as follows: 0 = DMA manager thread 1 = DMA channel. **Note** When set to 1, the Channel number field selects the DMA channel to debug.

3.3.21 Debug Instruction-1 Register

The DBGINST1 Register controls the upper bytes of the debug instruction for the DMAC. See *Issuing instructions to the DMAC using an APB interface* on page 2-16 for more information. Table 3-4 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-22 shows the register bit assignments.

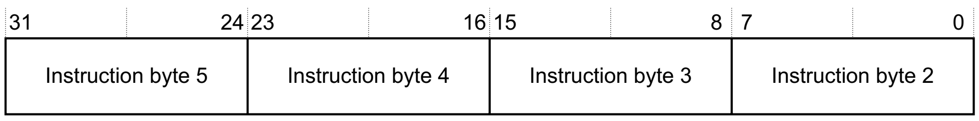


Figure 3-22 DBGINST1 Register bit assignments

Table 3-28 lists the register bit assignments.

Table 3-28 DBGINST1 Register bit assignments

Bits	Name	Function
[31:24]	Instruction byte 5	Instruction byte 5
[23:16]	Instruction byte 4	Instruction byte 4
[15:8]	Instruction byte 3	Instruction byte 4
[7:0]	Instruction byte 2	Instruction byte 2

3.3.22 Configuration Register 0

The CR0 Register provides the status of the tie-off control signals and information about the configuration of the DMAC:

- the number of DMA channels that it contains
- the number of peripheral request interfaces it provides
- the number of **irq** signals it provides.

Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-23 shows the register bit assignments.

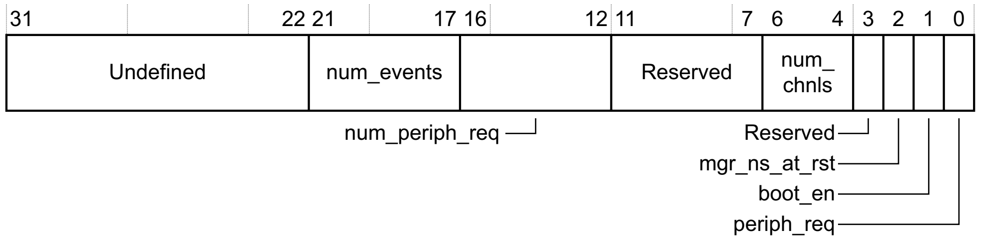


Figure 3-23 CR0 Register bit assignments

Table 3-29 lists the register bit assignments.

Table 3-29 CR0 Register bit assignments

Bits	Name	Function
[31:22]	-	Reserved, read undefined.

[21:17]	num_events	Number of interrupt outputs that the DMAC provides:
		b00000 = 1 interrupt output, **irq[0]**
		b00001 = 2 interrupt outputs, **irq[1:0]**
		b00010 = 3 interrupt outputs, **irq[2:0]**
		.
		.
		.
		b11111 = 32 interrupt outputs, **irq[31:0]**.

[16:12]	num_periph_req	Number of peripheral request interfaces that the DMAC provides: b00000 = 1 peripheral request interface b00001 = 2 peripheral request interfaces b00010 = 3 peripheral request interfaces . . . b11111 = 32 peripheral request interfaces. **Note** This field is only valid when the periph_req bit is set to 1.
[11:7]	-	Reserved, read undefined.

[6:4]	num_chnls	Number of DMA channels that the DMAC supports:
		b000 = 1 DMA channel
		b001 = 2 DMA channels
		b010 = 3 DMA channels
		.
		.
		.
		b111 = 8 DMA channels.

[3]	-	Reserved, read undefined.

Table 3-29 CR0 Register bit assignments (continued)

Bits	Name	Function
[2]	mgr_ns_at_rst	Indicates the status of the **boot_manager_ns** signal when the DMAC exited from reset: 0 = **boot_manager_ns** was LOW 1 = **boot_manager_ns** was HIGH.
[1]	boot_en	Indicates the status of the **boot_from_pc** signal when the DMAC exited from reset: 0 = **boot_from_pc** was LOW 1 = **boot_from_pc** was HIGH.
[0]	periph_req	Supports peripheral requests: 0 = the DMAC does not provide a peripheral request interface 1 = the DMAC provides the number of peripheral request interfaces that the num_periph_req field specifies.

3.3.23 Configuration Register 1

The CR1 Register provides information about the instruction cache configuration. Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-24 shows the register bit assignments.

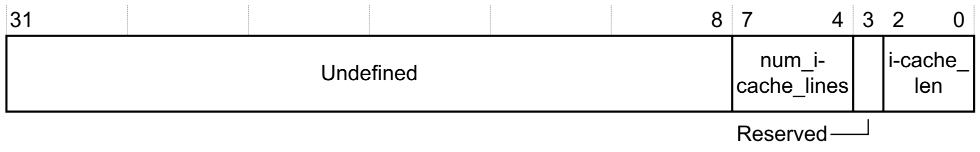


Figure 3-24 CR1 Registers bit assignments

Table 3-30 lists the register bit assignments.

Table 3-30 CR1 Registers bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined.

[7:4]	num_i-cache_lines	Number of i-cache lines:
		b0000 = 1 i-cache line
		b0001 = 2 i-cache lines
		b0010 = 3 i-cache lines
		.
		.
		.
		b1111 = 16 i-cache lines.

[3]	-	Reserved, read undefined.

[2:0]	i-cache_len	The length of an i-cache line:
		b000-b001 = reserved
		b010 = 4 bytes
		b011 = 8 bytes
		b100 = 16 bytes
		b101 = 32 bytes
		b110-b111 = reserved.

3.3.24 Configuration Register 2

The CR2 Register provides the value of the boot address that **boot_addr[31:0]** configures. Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-25 shows the register bit assignments.

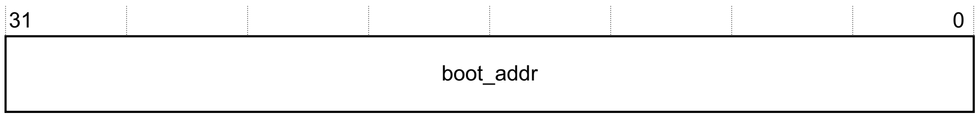


Figure 3-25 CR2 Registers bit assignments

Table 3-31 lists the register bit assignments.

Table 3-31 CR2 Register bit assignments

Bits	Name	Function
[31:0]	boot_addr	Provides the value of **boot_addr[31:0]** when the DMAC exited from reset.

3.3.25 Configuration Register 3

The CR3 Register provides the security state of the interrupt outputs that is initialized when the DMAC exits from reset. Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-26 shows the register bit assignments.

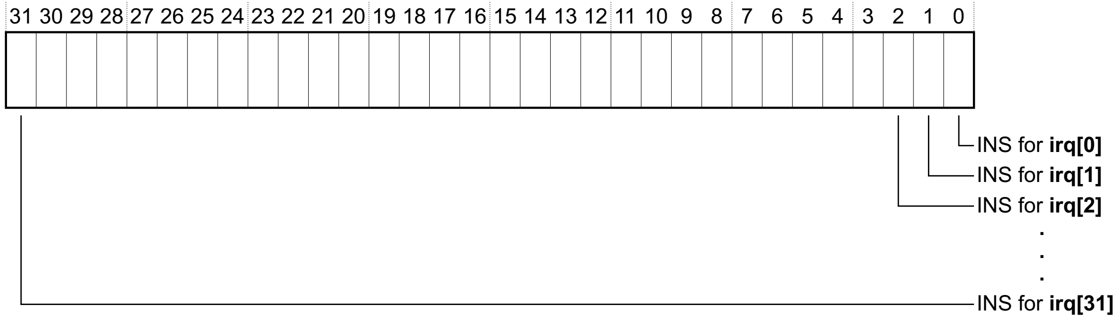


Figure 3-26 CR3 Registers bit assignments

Table 3-32 lists the register bit assignments.

Table 3-32 CR3 Register bit assignments

Bits	Name	Function
[31[a]:0]	INS	Provides the security state of the interrupt outputs: **Bit [*N*] = 0** Assigns **irq[N]** to the Secure state. **Bit [*N*] = 1** Assigns **irq[N]** to the Non-secure state. **Note** The **boot_irq_ns[x:0]** signals initialize the bits in this register, when the DMAC exits from reset. See Table A-12 on page A-13 for more information.

a. If you configure the DMAC to provide less than 32 **irq** outputs then the upper bits are undefined and read as zero.

3.3.26 Configuration Register 4

The CR4 Register provides the security state of the peripheral request interfaces that is initialized when the DMAC exits from reset. Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-27 shows the register bit assignments.

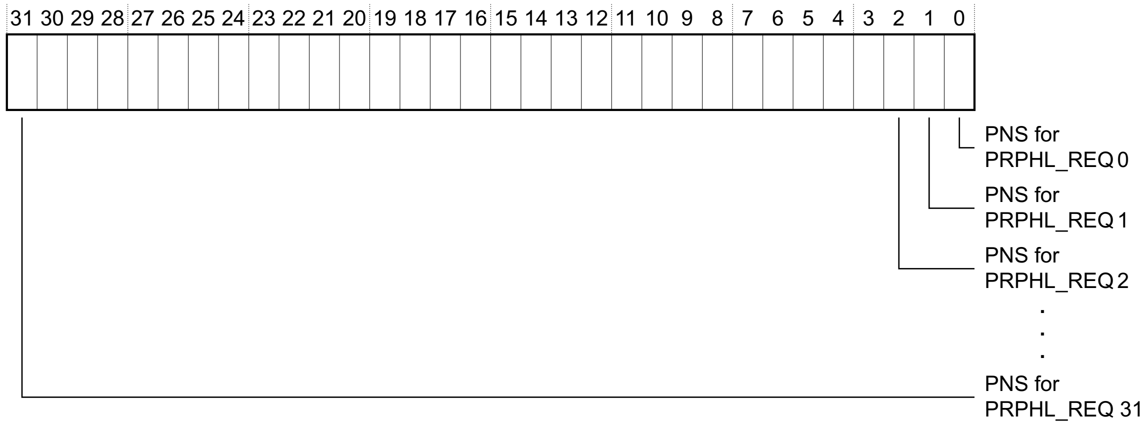


Figure 3-27 CR4 Registers bit assignments

Table 3-33 lists the register bit assignments.

Table 3-33 CR4 Register bit assignments

Bits	Name	Function
[31[a]:0]	PNS	Provides the security state of the peripheral request interfaces: **Bit [*N*] = 0** Assigns peripheral request interface N to the Secure state. **Bit [*N*] = 1** Assigns peripheral request interface N to the Non-secure state. **Note** The **boot_periph_ns** tie-off signals initialize the bits in this register, when the DMAC exits from reset. See Table A-12 on page A-13 for more information.

a. If you configure the DMAC to provide less than 32 peripheral request interfaces then the upper bits are undefined and read as zero.

3.3.27 Configuration Register Dn

The CRDn Register provides information about the configuration of the data buffer, data width, and write interleave capability of the DMAC. Table 3-5 on page 3-9 lists the address base offset, reset value, and access type for this register.

Figure 3-28 shows the register bit assignments.

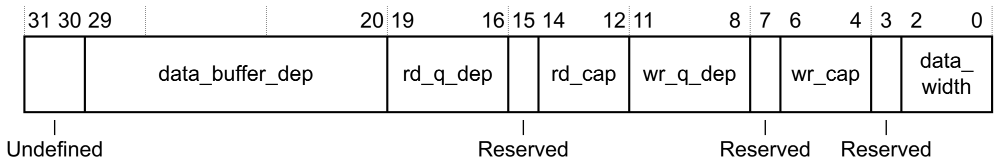


Figure 3-28 CRDn Register bit assignments

Table 3-34 lists the register bit assignments.

Table 3-34 CRDn Registers bit assignments

Bits	Name	Function
[31:30]	-	Reserved, read undefined.
[29:20]	data_buffer_dep	The number of lines that the data buffer contains: b000000000 = 1 line b000000001 = 2 lines . . . b111111111 = 1024 lines.
[19:16]	rd_q_dep	The depth of the read queue: b0000 = 1 line b0001 = 2 lines . . . b1111 = 16 lines.
[15]	-	Reserved, read undefined.

Table 3-34 CRDn Registers bit assignments (continued)

Bits	Name	Function
[14:12]	rd_cap	Read issuing capability that programs the number of outstanding read transactions: b000 = 1 b001 = 2 . . . b111 = 8.
[11:8]	wr_q_dep	The depth of the write queue: b0000 = 1 line b0001 = 2 lines . . . b1111 = 16 lines.
[7]	-	Reserved, read undefined.
[6:4]	wr_cap	Write issuing capability that programs the number of outstanding write transactions: b000 = 1 b001 = 2 . . . b111 = 8.
[3]	-	Reserved, read undefined.

[2:0]	data_width	The data bus width of the AXI interface:
		b000 = reserved
		b001 = reserved
		b010 = 32-bit
		b011 = 64-bit
		b100 = 128-bit
		b101-b111 = reserved.

3.3.28 Peripheral Identification Registers 0-3

The periph_id Registers are four 8-bit read-only registers, that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a single register that holds a 32-bit peripheral ID value. An external master reads them to discover the version of the DMAC. None of the registers 0-3 can be read when the DMAC is in reset.

Table 3-35 lists the register bit assignments.

Table 3-35 periph_id Register bit assignments

Bits	Name	Description
[31:25]	-	Reserved, read undefined.
[24]	integration_cfg	Configuration options are peripheral-specific. See *Peripheral Identification Register 3* on page 3-50.
[23:20]	-	The peripheral revision number is revision-dependent. See Table 3-38 on page 3-49.
[19:12]	designer	Designer's ID number. This is `0x41` for ARM.
[11:0]	part_number	Identifies the peripheral. This is `0x330` for the DMAC.

Figure 3-29 shows the correspondence between bits of the periph_id registers and the conceptual 32-bit Peripheral ID Register.

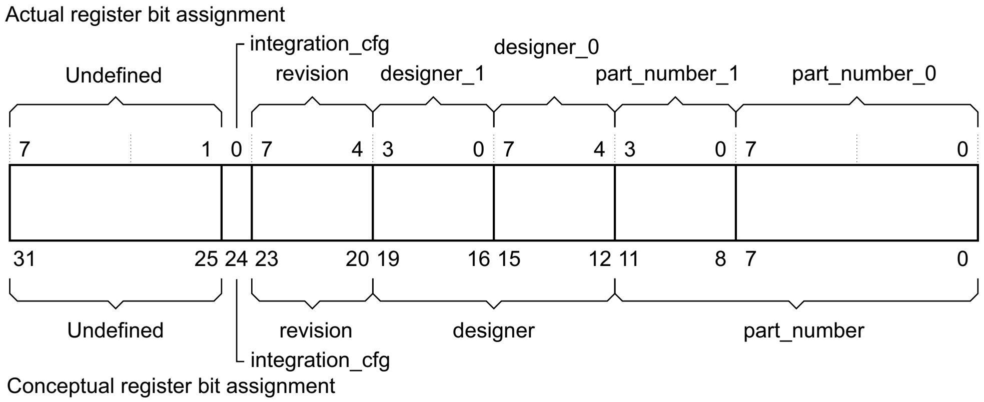


Figure 3-29 periph_id Register bit assignments

The following subsections describe the periph_id Registers:

- *Peripheral Identification Register 0* on page 3-49
- *Peripheral Identification Register 1* on page 3-49

- *Peripheral Identification Register 2*
- *Peripheral Identification Register 3* on page 3-50.

Peripheral Identification Register 0

The periph_id_0 Register is hard-coded and the fields in the register control the reset value. Table 3-36 lists the register bit assignments.

Table 3-36 periph_id_0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	part_number_0	These bits read back as `0x30`

Peripheral Identification Register 1

The periph_id_1 Register is hard-coded and the fields in the register control the reset value. Table 3-37 lists the register bit assignments.

Table 3-37 periph_id_1 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:4]	designer_0	These bits read back as `0x1`
[3:0]	part_number_1	These bits read back as `0x3`

Peripheral Identification Register 2

The periph_id_2 Register is hard-coded and the fields in the register control the reset value. Table 3-38 lists the register bit assignments.

Table 3-38 periph_id_2 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined.
[7:4]	revision	These bits read back as: • `0x0` for r0p0.
[3:0]	designer_1	These bits read back as `0x4`.

Peripheral Identification Register 3

The periph_id_3 Register is hard-coded and the fields in the register control the reset value. Table 3-39 lists the register bit assignments.

Table 3-39 periph_id_3 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined.
[7:1]	-	Reserved for future use, read undefined.
[0]	integration_cfg	Reads back as 0 to indicate that the DMAC does not contain any integration test logic.

3.3.29 PrimeCell Identification Registers 0-3

The pcell_id Registers are four eight-bit wide registers, that span address locations `0xFF0-0FFC`. The registers can conceptually be treated as a single register that holds a 32-bit PrimeCell ID value. You can use the register for automatic BIOS configuration. The pcell_id Register is set to `0xB105F00D`. You can access the register with one wait state. Table 3-40 lists the register bit assignments.

Table 3-40 pcell_id Register bit assignments

pcell_id_0-3 register				
Bits	**Reset value**	**Register**	**Bits**	**Description**
-	-	pcell_id_3	[31:8]	Read undefined
[31:24]	`0xB1`	pcell_id_3	[7:0]	These bits read back as `0xB1`
-	-	pcell_id_2	[31:8]	Read undefined
[23:16]	`0x05`	pcell_id_2	[7:0]	These bits read back as `0x05`
-	-	pcell_id_1	[31:8]	Read undefined
[15:8]	`0xF0`	pcell_id_1	[7:0]	These bits read back as `0xF0`
-	-	pcell_id_0	[31:8]	Read undefined
[7:0]	`0x0D`	pcell_id_0	[7:0]	These bits read back as `0x0D`

Figure 3-30 on page 3-51 shows the register bit assignments.

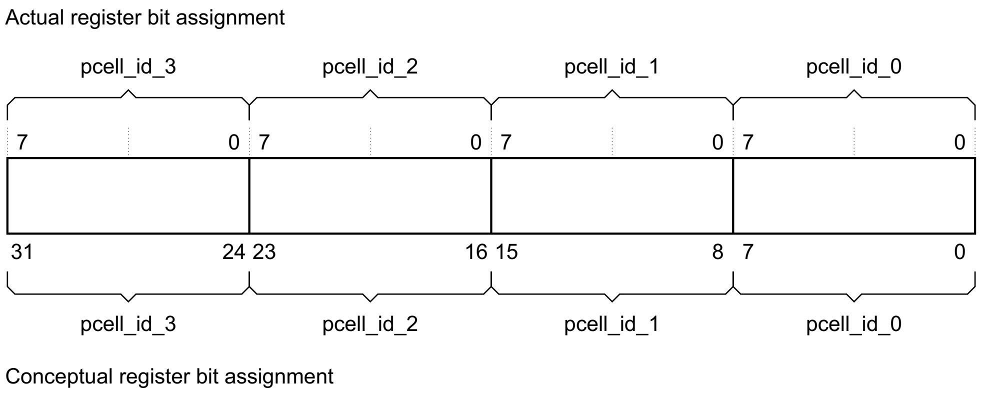


Figure 3-30 pcell_id Register bit assignments

The following subsections describe the pcell_id Registers:

- *PrimeCell Identification Register 0*
- *PrimeCell Identification Register 1* on page 3-52
- *PrimeCell Identification Register 2* on page 3-52
- *PrimeCell Identification Register 3* on page 3-52.

Note

You cannot read these registers when **aresetn** is active and the DMAC is in reset.

PrimeCell Identification Register 0

The pcell_id_0 Register is hard-coded and the fields in the register control the reset value. Table 3-41 lists the register bit assignments.

Table 3-41 pcell_id_0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	pcell_id_0	These bits read back as `0x0D`

PrimeCell Identification Register 1

The pcell_id_1 Register is hard-coded and the fields in the register control the reset value. Table 3-42 lists the register bit assignments.

Table 3-42 pcell_id_1 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	pcell_id_1	These bits read back as `0xF0`

PrimeCell Identification Register 2

The pcell_id_2 Register is hard-coded and the fields in the register control the reset value. Table 3-43 lists the register bit assignments.

Table 3-43 pcell_id_2 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	pcell_id_2	These bits read back as `0x5`

PrimeCell Identification Register 3

The pcell_id_3 Register is hard-coded and the fields in the register control the reset value. Table 3-44 lists the register bit assignments.

Table 3-44 pcell_id_3 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined
[7:0]	pcell_id_3	These bits read back as `0xB1`

Chapter 4
Instruction Set

This chapter describes the instruction set of the DMAC. It contains the following sections:

- *Instruction syntax conventions* on page 4-2
- *Instruction set summary* on page 4-3
- *Instructions* on page 4-5
- *Assembler directives* on page 4-25.

4.1 Instruction syntax conventions

The following conventions are used in assembler syntax prototype lines and their subfields:

`< >`	Any item bracketed by `<` and `>` is mandatory. A description of the item and of how it is encoded in the instruction is supplied by subsequent text.
`[ ]`	Any item bracketed by `[` and `]` is optional. A description of the item and of how its presence or absence is encoded in the instruction is supplied by subsequent text.
spaces	Single spaces are used for clarity, to separate items. When a space is obligatory in the assembler syntax, two or more consecutive spaces are used.

4.2 Instruction set summary

The DMAC instructions:

- use a `DMA` prefix, to provide a unique name-space
- have 8-bit opcodes that might use a variable data payload of 0, 8, 16, or 32-bits
- use suffixes that are consistent.

Table 4-1 lists a summary of the instruction syntax.

Table 4-1 Instruction syntax summary

Mnemonic	**Instruction**	**Thread usage:** • **M = DMA manager** • **C = DMA channel**		**Description**
`DMAADDH`	Add Halfword	-	C	See *DMAADDH* on page 4-5
`DMAEND`	End	M	C	See *DMAEND* on page 4-5
`DMAFLUSHP`	Flush and notify Peripheral	-	C	See *DMAFLUSHP* on page 4-6
`DMAGO`	Go	M	-	See *DMAGO* on page 4-6
`DMALD`	Load	-	C	See *DMALD[S|B]* on page 4-8
`DMALDP`	Load Peripheral	-	C	See *DMALDP<S|B>* on page 4-9
`DMALP`	Loop	-	C	See *DMALP* on page 4-10
`DMALPEND`	Loop End	-	C	See *DMALPEND[S|B]* on page 4-11
`DMALPFE`	Loop Forever	-	C	See *DMALPFE* on page 4-13
`DMAKILL`	Kill	M	C	See *DMAKILL* on page 4-14
`DMAMOV`	Move	-	C	See *DMAMOV* on page 4-15
`DMANOP`	No operation	M	C	See *DMANOP* on page 4-17
`DMARMB`	Read Memory Barrier	-	C	See *DMARMB* on page 4-17
`DMASEV`	Send Event	M	C	See *DMASEV* on page 4-18
`DMAST`	Store	-	C	See *DMAST[S|B]* on page 4-19
`DMASTP`	Store and notify Peripheral	-	C	See *DMASTP<S|B>* on page 4-20
`DMASTZ`	Store Zero	-	C	See *DMASTZ* on page 4-21

Table 4-1 Instruction syntax summary (continued)

Mnemonic	Instruction	Thread usage: • M = DMA manager	• C = DMA channel	Description
`DMAWFE`	Wait For Event	M	C	See *DMAWFE* on page 4-21
`DMAWFP`	Wait For Peripheral	-	C	See *DMAWFP<S|B|P>* on page 4-22
`DMAWMB`	Write Memory Barrier	-	C	See *DMAWMB* on page 4-23

4.3 Instructions

The following sections describe the instructions that a DMAC can execute.

4.3.1 `DMAADDH`

Add Halfword adds an immediate 16-bit value to the *Source Address Registers* on page 3-27 or *Destination Address Registers* on page 3-29, for the DMA channel thread. This enables the DMAC to support 2D DMA operations.

Figure 4-1 shows the instruction encoding.

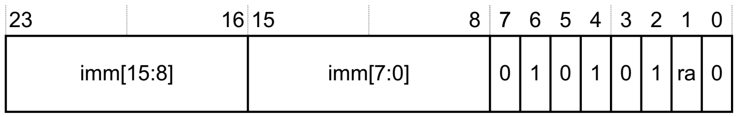


Figure 4-1 `DMAADDH` encoding

Assembler syntax

`DMAADDH <address_register>, <16-bit immediate>`

where:

`<address_register>`	Selects the address register to use. It must be either:	
	`SA`	*Source Address Registers* on page 3-27 and sets `ra` to 0.
	`DA`	*Destination Address Registers* on page 3-29 and sets `ra` to 1.
`<16-bit immediate>`	The immediate value to be added to the `<address_register>`.	

Operation

You can only use this instruction in a DMA channel thread.

4.3.2 `DMAEND`

End signals to the DMAC that the DMA sequence is complete. After all DMA transfers are complete for the DMA channel then the DMAC moves the channel to the Stopped state. When the DMAC receives `DMAEND` for a channel that is suspended, because of a previous abort, it moves the channel to the Stopped state. In all cases, it also flushes data from the MFIFO and invalidates all cache entries for the thread.

Figure 4-2 on page 4-6 shows the instruction encoding.

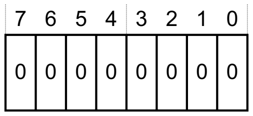


Figure 4-2 `DMAEND` **encoding**

Assembler syntax

```
DMAEND
```

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

4.3.3 `DMAFLUSHP`

Flush Peripheral clears the state in the DMAC that describes the contents of the peripheral and sends a message to the peripheral to resend its level status.

Figure 4-3 shows the instruction encoding.

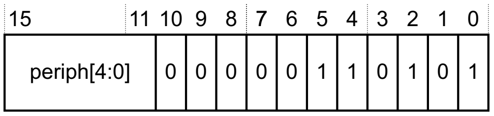


Figure 4-3 `DMAFLUSHP` **encoding**

Assembler syntax

```
DMAFLUSHP <peripheral>
```

where:

`<peripheral>` 5-bit immediate, value 0-31.

Operation

You can only use this instruction in a DMA channel thread.

4.3.4 `DMAGO`

When the DMA manager executes Go for a DMA channel that is in the Stopped state, it performs the following steps on the DMA channel:

- moves a 32-bit immediate into the program counter

- sets its security state
- updates it to the Executing state.

Note

If a DMA channel is not in the Stopped state when the DMA manager executes `DMAGO` then the DMA manager thread aborts.

Figure 4-4 shows the instruction encoding.

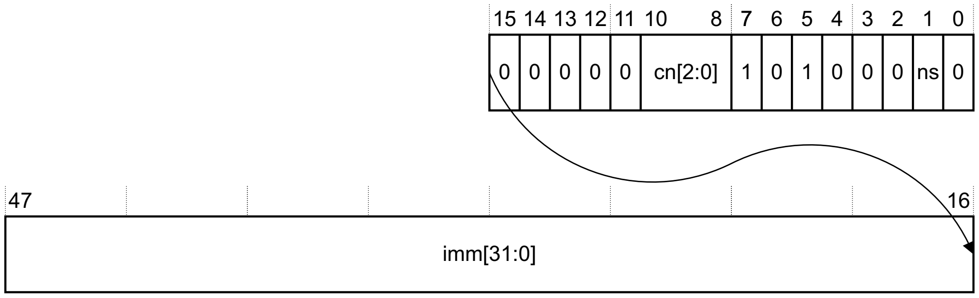


Figure 4-4 `DMAGO` encoding

Assembler syntax

```
DMAGO <channel_number>, <32-bit_immediate> [, ns]
```

where:

`<channel_number>` Selects a DMA channel. It must be one of:

`C0`	DMA channel 0.
`C1`	DMA channel 1.
`C2`	DMA channel 2.
`C3`	DMA channel 3.
`C4`	DMA channel 4.
`C5`	DMA channel 5.
`C6`	DMA channel 6.
`C7`	DMA channel 7.

Note

If you provide a channel number that is not available for your configuration of the DMAC then the DMA manager thread aborts.

`<32-bit_immediate>`	The immediate value that is written to the *Channel Program Counter Registers* on page 3-26, for the selected `<channel_number>`.
`[ns]`	• If ns is present, the DMA channel operates in the Non-secure state. • Otherwise the execution of the instruction depends on the security state of the DMA manager: **DMA manager is in the Secure state** DMA channel operates in the Secure state. **DMA manager is in the Non-secure state** DMAC aborts.

Operation

You can only use this instruction with the DMA manager thread.

4.3.5 DMALD[S|B]

Load instructs the DMAC to perform a DMA load, using AXI transactions that the *Source Address Registers* on page 3-27 and *Channel Control Registers* on page 3-30 specify. It places the read data into the MFIFO and tags it with the corresponding channel number. DMALD is an unconditional instruction but DMALDS and DMALDB are conditional on the setting of request_flag. If the src_inc bit in the *Channel Control Registers* on page 3-30 is set to incrementing then the DMAC updates the *Source Address Registers* on page 3-27 after it executes DMALD[S|B].

Note

The value of request_flag is set by the DMAC when it executes a DMAWFP instruction. See *DMAWFP<S|B|P>* on page 4-22.

Figure 4-5 shows the instruction encoding.

7	6	5	4	3	2	1	0
0	0	0	0	0	1	bs	x

Figure 4-5 DMALD[S|B] encoding

Assembler syntax

`DMALD[S|B]`

where:

`[S]` If S is present then the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**
The DMAC performs a DMA load.

`request_flag` = **Burst**
The DMAC performs a `DMANOP`.

`[B]` If B is present then the assembler sets `bs` to 1 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**
The DMAC performs a `DMANOP`.

`request_flag` = **Burst**
The DMAC performs a DMA load.

If you do not specify the `S` or `B` operand then the assembler sets `bs` to 0 and `x` to 0, and the DMAC always executes a DMA load.

Operation

You can only use this instruction in a DMA channel thread. If you specify the `S` or `B` operand then execution of the instruction is conditional on the setting of `request_flag` matching that of the instruction as *Assembler syntax* describes.

4.3.6 `DMALDP<S|B>`

Load and notify Peripheral instructs the DMAC to perform a DMA load, using AXI transactions that the *Source Address Registers* on page 3-27 and *Channel Control Registers* on page 3-30 specify. It places the read data into a FIFO that is tagged with the corresponding channel number and after it receives the last data item, it updates datype[1:0] to indicate to the peripheral that the data transfer is complete. If the src_inc bit in the *Channel Control Registers* on page 3-30 is set to incrementing then the DMAC updates the *Source Address Registers* on page 3-27 after it executes `DMALDP<S|B>`.

Figure 4-6 on page 4-10 shows the instruction encoding.

15 … 11	10	9	8	7	6	5	4	3	2	1	0
periph[4:0]	0	0	0	0	0	1	0	0	1	bs	1

Figure 4-6 `DMALDP<S|B>` **encoding**

Assembler syntax

```
DMALDP<S|B> <peripheral>
```

where:

`<S>` When S is present the assembler sets `bs` to 0. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**
The DMAC performs a load using a single DMA transfer.

`request_flag` = **Burst**
The DMAC performs a `DMANOP`.

`<B>` When B is present the assembler sets `bs` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**
The DMAC performs a `DMANOP`.

`request_flag` = **Burst**
The DMAC performs a load using a burst DMA transfer.

`<peripheral>` 5-bit immediate, value 0-31.

Operation

You can only use this instruction in a DMA channel thread. Execution of the instruction is conditional on the setting of `request_flag` matching that of the instruction as *Assembler syntax* describes.

4.3.7 DMALP

Loop instructs the DMAC to load an 8-bit value into the Loop Counter Register you specify. This instruction indicates the start of a section of instructions, and you set the end of the section using the `DMALPEND` instruction. See *DMALPEND[S|B]* on page 4-11. The DMAC repeats the set of instructions that you insert between `DMALP` and `DMALPEND` until the value in the Loop Counter Register reaches zero.

——— Note ———

The DMAC saves the value of the PC for the instruction that follows `DMALP`. After the DMAC executes `DMALPEND`, and the Loop Counter Register is not zero, this enables it to execute the first instruction in the loop.

Figure 4-7 shows the instruction encoding.

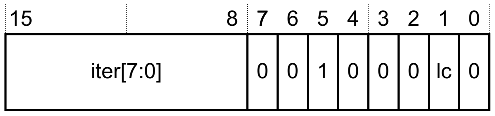


Figure 4-7 `DMALP` encoding

Assembler syntax

`DMALP <loop_iterations>`

where:

`<loop_iterations>`

An 8-bit value that specifies the number of loops to perform.

——— Note ———

The assembler determines the Loop Counter Register to use and either:

- Sets `lc` to 0. The DMAC writes the value, `loop_iterations` minus 1, to the *Loop Counter 0 Registers* on page 3-35.
- Sets `lc` to 1. The DMAC writes the value, `loop_iterations` minus 1, to the *Loop Counter 1 Registers* on page 3-36.

Operation

You can only use this instruction in a DMA channel thread.

4.3.8 `DMALPEND[S|B]`

Loop End indicates the last instruction in the program loop and instructs the DMAC to read the value of the Loop Counter Register. Depending on the value returned it either:

Value is zero DMAC executes a `DMANOP`.

Value is non-zero DMAC decrements the value in the Loop Counter Register and updates the thread PC to contain the address of the first instruction in the program loop, that is, the instruction that follows the `DMALP` or `DMALPFE`, that corresponds to the `DMALPEND`.

Figure 4-8 shows the instruction encoding.

15 8	7	6	5	4	3	2	1	0
backwards_jump[7:0]	0	0	1	nf	1	lc	bs	x

Figure 4-8 `DMALPEND[S|B]` **encoding**

Assembler syntax

`DMALPEND[S|B]`

where:

`[S]` If S is present then the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**

The DMAC continues executing the instructions in the loop.

`request_flag` = **Burst**

The DMAC performs a `DMANOP`.

`[B]` If B is present then the assembler sets `bs` to 1 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**

The DMAC performs a `DMANOP`.

`request_flag` = **Burst**

The DMAC continues executing the instructions in the loop.

Note

If you do not specify the `S` or `B` operand then the assembler sets `bs` to 0 and `x` to 0, and the DMAC always executes the instructions in the program loop.

Note

To correctly assign the additional bits in the `DMALPEND` instruction, that Figure 4-8 on page 4-12 shows, the assembler determines the values for:

`backwards_jump[7:0]`	Sets the relative location of the first instruction in the program loop.
`nf`	Sets it to: • 0 if `DMALPFE` started the program loop • 1 if `DMALP` started the program loop.
`lc`	Sets it to: • 0 if the *Loop Counter 0 Registers* on page 3-35 contains the loop counter value • 1 if the *Loop Counter 1 Registers* on page 3-36 contains the loop counter value.

Operation

You can only use this instruction in a DMA channel thread. If you specify the `S` or `B` operand then execution of the instruction is conditional on the setting of `request_flag` matching that of the instruction as *Assembler syntax* on page 4-12 describes.

4.3.9 DMALPFE

Loop Forever is used by the assembler to configure certain bits in `DMALPEND`. See *DMALPEND[S|B]* on page 4-11.

Note

When the assembler encounters `DMALPFE` it does not create an instruction for the DMAC but instead it modifies the behavior of `DMALPEND`. The insertion of `DMALPFE` in program code identifies the start of the loop.

Assembler syntax

```
DMALPFE
```

4.3.10 `DMAKILL`

Kill instructs the DMAC to immediately terminate execution of a thread. Depending on the thread type the DMAC performs the following steps:

DMA manager thread

1. Invalidates all cache entries for the DMA manager.
2. Moves the DMA manager to the Stopped state.

DMA channel thread

1. Moves the DMA channel to the Killing state.
2. Waits for AXI transactions, with an ID equal to the DMA channel number, to complete.
3. Invalidates all cache entries for the DMA channel.
4. Remove all entries in the MFIFO for the DMA channel.
5. Remove all entries in the read buffer queue and write buffer queue for the DMA channel.
6. Moves the DMA channel to the Stopped state.

Figure 4-9 shows the instruction encoding.

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

Figure 4-9 `DMAKILL` encoding

Assembler syntax

```
DMAKILL
```

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

Note

Programs for the DMAC must not use `DMAKILL`. Only use the *Debug Instruction-0 Register* on page 3-38 to issue this instruction.

4.3.11 DMAMOV

Move instructs the DMAC to move a 32-bit immediate into the following registers:

- *Source Address Registers* on page 3-27
- *Destination Address Registers* on page 3-29
- *Channel Control Registers* on page 3-30.

Figure 4-10 shows the instruction encoding.

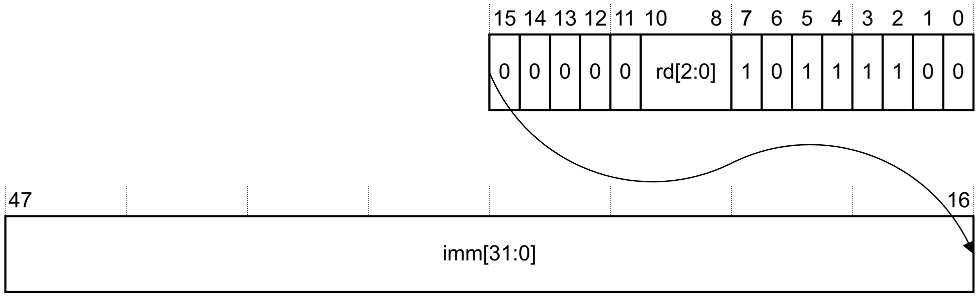


Figure 4-10 DMAMOV **encoding**

Assembler syntax

```
DMAMOV <destination_register>, <32-bit_immediate>
```

where:

`<destination_register>`

The valid registers are:

`SAR`	Selects the *Source Address Registers* on page 3-27 and sets `rd` to b000.
`CCR`	Selects the *Channel Control Registers* on page 3-30 and sets `rd` to b001.
`DAR`	Selects the *Destination Address Registers* on page 3-29 Register and sets `rd` to b010.

`<32-bit_immediate>`

A 32-bit value that is written to the specified destination register.

Note

See *DMAMOV CCR* on page 4-27 for information about using the assembler to program the various fields that the *Channel Control Registers* on page 3-30 contains.

Operation

You can only use this instruction in a DMA channel thread.

4.3.12 `DMANOP`

No Operation does nothing. You can use this instruction for code alignment purposes.

Figure 4-11 shows the instruction encoding.

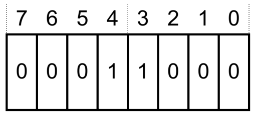


Figure 4-11 `DMANOP` encoding

Assembler syntax

`DMANOP`

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

4.3.13 `DMARMB`

Read Memory Barrier forces the DMA channel to wait until all active AXI read transactions associated with that channel are complete. This enables write-after-read sequences to the same address location with no hazards.

Figure 4-12 shows the instruction encoding.

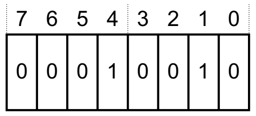


Figure 4-12 `DMARMB` encoding

Assembler syntax

`DMARMB`

Operation

You can only use this instruction in a DMA channel thread.

4.3.14 DMASEV

Send Event instructs the DMAC to signal an event. Depending on how you program the *Interrupt Enable Register* on page 3-13 this either:

- generates event `<event_num>`

  Note

  Typically, you use `DMAWFE` to stall a thread and then another thread executes `DMASEV`, using the appropriate event number, to unstall the waiting thread. See *Using an event to restart DMA channels* on page 2-23.

- signals an interrupt using **irq<event_num>**.

  Note

  The DMAC aborts the thread if you select an event_num that is not available for your configuration of the DMAC.

Figure 4-13 shows the instruction encoding.

15 11	10	9	8	7	6	5	4	3	2	1	0
event_num[4:0]	0	0	0	0	0	1	1	0	1	0	0

Figure 4-13 `DMASEV` encoding

Assembler syntax

```
DMASEV <event_num>
```

where:

`<event_num>`	5-bit immediate, value 0-31.

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

4.3.15 `DMAST[S|B]`

Store instructs the DMAC to transfer data from the FIFO to the location that the *Destination Address Registers* on page 3-29 specifies, using AXI transactions that the DA Register and *Channel Control Registers* on page 3-30 specify. If the dst_inc bit in the *Channel Control Registers* on page 3-30 is set to incrementing then the DMAC updates the *Destination Address Registers* on page 3-29 after it executes `DMAST[S|B]`.

Figure 4-14 shows the instruction encoding.

7	6	5	4	3	2	1	0
0	0	0	0	1	0	bs	x

Figure 4-14 `DMAST[S|B]` **encoding**

Assembler syntax

```
DMAST[S|B]
```

where:

`[S]` If S is present then the assembler sets `bs` to 0 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**

The DMAC performs a single DMA store.

Note

The DMAC ignores the setting of the dst_burst_len field in the *Channel Control Registers* on page 3-30 and always performs an AXI transfer with a burst length of one.

`request_flag` = **Burst**

The DMAC performs a `DMANOP`.

`[B]` If B is present then the assembler sets `bs` to 1 and `x` to 1. The instruction is conditional on the setting of `request_flag`:

`request_flag` = **Single**

The DMAC performs a `DMANOP`.

`request_flag` = **Burst**

The DMAC performs a DMA store.

If you do not specify the `S` or `B` operand then the assembler sets `bs` to 0 and `x` to 0, and the DMAC always executes a DMA store.

Operation

You can only use this instruction in a DMA channel thread. If you specify the `S` or `B` operand then execution of the instruction is conditional on the setting of `request_flag` matching that of the instruction as *Assembler syntax* on page 4-19 describes.

The DMAC only commences the burst when the MFIFO contains all of the data necessary to complete the burst transfer.

4.3.16 `DMASTP<S|B>`

Store and notify Peripheral instructs the DMAC to transfer data from the FIFO to the location that the *Destination Address Registers* on page 3-29 specifies, using AXI transactions that the DA Register and *Channel Control Registers* on page 3-30 specify. It uses the DMA channel number to access the appropriate location in the FIFO. After the DMA store is complete, and the DMAC has received a buffered write response, it updates datype[1:0], to notify the peripheral that the data transfer is complete. If the dst_inc bit in the *Channel Control Registers* on page 3-30 is set to incrementing then the DMAC updates the *Destination Address Registers* on page 3-29 after it executes `DMASTP<S|B>`.

Figure 4-15 shows the instruction encoding.

15 11	10	9	8	7	6	5	4	3	2	1	0
periph[4:0]	0	0	0	0	0	1	0	1	0	bs	1

Figure 4-15 `DMASTP<S|B>` **encoding**

Assembler syntax

```
DMASTP<S|B> <peripheral>
```

where:

`<S>` Sets `bs` to 0. This instructs the DMAC to perform:

- a single DMA store operation if `request_flag` is programmed to Single

  Note

  The DMAC ignores the setting of the dst_burst_len field in the *Channel Control Registers* on page 3-30 and always performs an AXI transfer with a burst length of one.

- a `DMANOP` if `request_flag` is programmed to Burst.

`<B>`	Sets `bs` to 1. This instructs the DMAC to perform: • the DMA store if `request_flag` is programmed to Burst • a `DMANOP` if `request_flag` is programmed to Single.
`<peripheral>`	5-bit immediate, value 0-31.

Operation

You can only use this instruction in a DMA channel thread.

The DMAC only commences the burst when the MFIFO contains all of the data necessary to complete the burst transfer.

4.3.17 `DMASTZ`

Store Zero instructs the DMAC to store zeros, using AXI transactions that the *Destination Address Registers* on page 3-29 and *Channel Control Registers* on page 3-30 specify. If the dst_inc bit in the *Channel Control Registers* on page 3-30 is set to incrementing then the DMAC updates the *Destination Address Registers* on page 3-29 after it executes `DMASTZ`.

Figure 4-16 shows the instruction encoding.

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

Figure 4-16 `DMASTZ` encoding

Assembler syntax

```
DMASTZ
```

Operation

You can only use this instruction in a DMA channel thread.

4.3.18 `DMAWFE`

Wait For Event instructs the DMAC to halt execution of the thread until the event, that `event_num` specifies, occurs. When the event occurs, the thread moves to the Executing state and the DMAC clears the event.

Figure 4-17 on page 4-22 shows the instruction encoding.

15 11	10	9	8	7	6	5	4	3	2	1	0
event_num[4:0]	0	i	0	0	0	1	1	0	1	1	0

Figure 4-17 `DMAWFE` **encoding**

Assembler syntax

```
DMAWFE <event_num>[, invalid]
```

where:

`<event_num>`	5-bit immediate, value 0-31.
`[invalid]`	If `invalid` is present the DMAC invalidates the instruction cache for the current DMA thread.

Note

- The DMAC aborts the thread if you select an event_num that is not available for your configuration of the DMAC.
- To ensure cache coherency, you must use `invalid` when a processor writes the instruction stream for a DMA channel.

Operation

You can use the instruction with the DMA manager thread and the DMA channel thread.

4.3.19 `DMAWFP<S|B|P>`

Wait For Peripheral instructs the DMAC to halt execution of the thread until the specified peripheral signals a DMA request for that DMA channel.

Figure 4-18 shows the instruction encoding.

15 11	10	9	8	7	6	5	4	3	2	1	0
peripheral[4:0]	0	0	0	0	0	1	1	0	0	bs	p

Figure 4-18 `DMAWFP<S|B|P>` **encoding**

Assembler syntax

```
DMAWFP<S|B|P> <peripheral>
```

where:

`<S>` Sets `bs` to 0 and `p` to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the `request_flag` to single, for that DMA channel.

`<B>` Sets `bs` to 1 and `p` to 0. This instructs the DMAC to continue executing the DMA channel thread after it receives a burst DMA request. The DMAC sets the `request_flag` to burst.

——— **Note** ———

The DMAC ignores single burst DMA requests, as Figure 2-10 on page 2-21shows.

———

`<P>` Sets `bs` to 0 and `p` to 1. This instructs the DMAC to continue executing the DMA channel thread after it receives a single or burst DMA request. The DMAC sets the `request_flag` to:

Single When it receives a single DMA request.

Burst When it receives a burst DMA request.

`<peripheral>` 5-bit immediate, value 0-31.

——— **Note** ———

The DMAC aborts the thread if you select a peripheral number that is not available for your configuration of the DMAC.

———

Operation

You can only use this instruction in a DMA channel thread.

4.3.20 `DMAWMB`

Write Memory Barrier forces the DMA channel to wait until all active AXI write transactions associated with that channel have completed. This enables read-after-write sequences to the same address location with no hazards.

Figure 4-19 shows the instruction encoding.

7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1

Figure 4-19 `DMAWMB` encoding

Assembler syntax

```
DMAWMB
```

Operation

You can only use this instruction in a DMA channel thread.

4.4 Assembler directives

The assembler provides the following additional commands:

- *DCD*
- *DCB*
- *DMALP*
- *DMALPFE* on page 4-26
- *DMALPEND* on page 4-26
- *DMAMOV CCR* on page 4-27.

4.4.1 `DCD`

Assembler directive to place a 32-bit immediate in the instruction stream.

Syntax

```
DCD imm32
```

4.4.2 `DCB`

Assembler directive to place an 8-bit immediate in the instruction stream.

Syntax

```
DCB imm8
```

4.4.3 `DMALP`

Assembler directive to insert an iterative loop.

Syntax

```
DMALP [<LC0>|<LC1>] <loop_iterations>
```

where:

`<loop_iterations>`
An 8-bit value that specifies the number of loops to perform.

Note

For clarity in writing assembler instructions, the 8-bit value is the actual number of iterations of the loop to be executed. The assembler decrements this by one to create the actual value (0-255) that the DMAC uses.

`[LC0]`	If LC0 is present then the DMAC stores `<loop_iterations>` in the *Loop Counter 0 Registers* on page 3-35.
`[LC1]`	If LC1 is present then the DMAC stores `<loop_iterations>` in the *Loop Counter 1 Registers* on page 3-36.

Note

If LC0 or LC1 is not present then the assembler determines the Loop Counter Register to use.

4.4.4 DMALPFE

Assembler directive to insert a repetitive loop.

Syntax

```
DMALPFE
```

Enables the assembler to clear the `nf` bit that is present in `DMALPEND`. See *DMALPEND*.

4.4.5 DMALPEND

Assembler directive that indicates the end of a loop.

Syntax

```
DMALPEND
```

To correctly assign the additional bits in the `DMALPEND` instruction, that Figure 4-8 on page 4-12 shows, the assembler determines the values for:

`backwards_jump[7:0]`	Sets the relative location of the first instruction in the program loop.
`nf`	Sets it to: • 0 if `DMALPFE` started the program loop • 1 if `DMALP` started the program loop.

`lc` Sets it to:

- 0 if the *Loop Counter 0 Registers* on page 3-35 contains the loop counter value
- 1 if the *Loop Counter 1 Registers* on page 3-36 contains the loop counter value.

4.4.6 DMAMOV CCR

Assembler directive that enables you to program the *Channel Control Registers* on page 3-30 using the format that *Syntax* shows.

Syntax

```
DMAMOV CCR, [SB<1-16>] [SS<1|2|4|8|16|32|64|128>] [SA<I|F>]
            [SP<imm3>] [SC<imm4>]
            [DB<1-16>] [DS<1|2|4|8|16|32|64|128>] [DA<I|F>]
            [DP<imm3>] [DC<imm4>]
            [ES<8|16|32|64|128>]
```

Table 4-2 lists the argument descriptions and the default values.

Table 4-2 `DMAMOV CCR` argument description and the default values

Syntax	Description	Options	Default

SA	Source address increment. Sets the value of **ARBURST[0]**.	I = Increment
		F = Fixed
			I

SS	Source burst size in bytes. Sets the value of **ARSIZE[2:0]**.	1, 2, 4, 8, 16, 32, 64, or 128	8
SB	Source burst length. Sets the value of **ARLEN[3:0]**.	1 to 16	1
SP	Source protection.	0 to 7[a]	0
SC	Source cache.	0 to 15[a][b]	0

DA	Destination address increment. Sets the value of **AWBURST[0]**.	I = Increment
		F = Fixed
			I

DS	Destination burst size in bytes. Sets the value of **AWSIZE[2:0]**.	1, 2, 4, 8, 16, 32, 64, or 128	8
DB	Destination burst length. Sets the value of **AWLEN[3:0]**.	1 to 16	1
DP	Destination protection.	0 to 7[a]	0
DC	Destination cache.	0 to 15[a][c]	0
ES	Endian swap size, in bits.	8, 16, 32, 64, or 128	8

a. You must use decimal values when programming this immediate value.
b. Because **ARCACHE[3]** is tied LOW by the DMAC then the assembler always sets bit 3 to 0 and uses bits [2:0] of your chosen value for SC. See *CC_n Register bit assignments* on page 3-31.
c. Because **AWCACHE[2]** is tied LOW by the DMAC then the assembler always sets bit 2 to 0 and uses bit [3] and bits [1:0] of your chosen value for DC. See *CC_n Register bit assignments* on page 3-31.

Appendix A
Signal Descriptions

This appendix lists and describes the DMAC signals. It contains the following sections:

- *Clocks and resets* on page A-2
- *AXI signals* on page A-3
- *APB signals* on page A-7
- *Peripheral request interface* on page A-9
- *Interrupt signals* on page A-11
- *Tie-off signals* on page A-12.

A.1 Clocks and resets

Table A-1 lists the clock and reset signals.

Table A-1 Clock and reset

Name	Type	Source/ destination	Description
aclk	Input	Clock source	AXI clock.
aresetn	Input	Reset source	DMAC active-LOW reset: 0 = apply DMAC reset 1 = do not apply DMAC reset.
pclken	Input	Clock generator	Clock enable signal that enables the APB interfaces to operate at either: • the **aclk** frequency • a divided integer multiple of **aclk** that is aligned to **aclk**. **Note** If **pclken** is not used then it must be tied HIGH. This results in the APB interfaces being clocked directly by **aclk**.

A.2 AXI signals

The following sections describe the AXI interface signals:

- *Write address (AXI-AW) channel signals*
- *Write data (AXI-W) channel signals* on page A-4
- *Write response (AXI-B) channel signals* on page A-4
- *Read address (AXI-AR) channel signals* on page A-5
- *Read data (AXI-R) channel signals* on page A-5.

A.2.1 Write address (AXI-AW) channel signals

Table A-2 lists the AXI write address signals.

Table A-2 AXI-AW signals

Signal	AMBA equivalent[a]
awaddr[31:0]	**AWADDR[31:0]**
awburst[1:0]	**AWBURST[1:0]**
awid[ID_MSB:0][b]	**AWID[ID_MSB:0]**
awlen[3:0]	**AWLEN[3:0]**
awprot[2:0]	**AWPROT[2:0]**
awready	**AWREADY**
awsize[2:0]	**AWSIZE[2:0]**
awvalid	**AWVALID**

a. See the *AMBA AXI Protocol Specification* for a description of these signals.
b. The value of ID_MSB is set during configuration of the DMAC.

The DMAC does not support locked or exclusive accesses and therefore **awlock[1:0]** is tied LOW.

A.2.2 Write data (AXI-W) channel signals

Table A-3 lists the AXI write data signals.

Table A-3 AXI-W signals

Signal	AMBA equivalent[a]
wdata[DATA_MSB:0][b]	**WDATA[DATA_MSB:0]**
wid[ID_MSB:0][b]	**WID[ID_MSB:0]**
wlast	**WLAST**
wready	**WREADY**
wstrb[STRB_MSB:0][b]	**WSTRB[STRB_MSB:0]**
wvalid	**WVALID**

a. See the *AMBA AXI Protocol Specification* for a description of these signals.
b. The value of DATA_MSB, ID_MSB, and STRB_MSB are set during configuration of the DMAC.

A.2.3 Write response (AXI-B) channel signals

Table A-4 lists the AXI write response signals.

Table A-4 AXI-B signals

Signal	AMBA equivalent[a]
bid[ID_MSB:0][b]	**BID[ID_MSB:0]**
bready	**BREADY**
bresp[1:0]	**BRESP[1:0]**
bvalid	**BVALID**

a. See the *AMBA AXI Protocol Specification* for a description of these signals.
b. The value of ID_MSB is set during configuration of the DMAC.

A.2.4 Read address (AXI-AR) channel signals

Table A-5 lists the AXI read address signals.

Table A-5 AXI-AR signals

Signal	AMBA equivalent[a]
araddr[31:0]	**ARADDR[31:0]**
arburst[1:0]	**ARBURST[1:0]**
arid[ID_MSB:0][b]	**ARID[ID_MSB:0]**
arlen[3:0]	**ARLEN[3:0]**
arprot[2:0]	**ARPROT[2:0]**
arready	**ARREADY**
arsize[2:0]	**ARSIZE[2:0]**
arvalid	**ARVALID**

a. See the *AMBA AXI Protocol Specification* for a description of these signals.
b. The value of ID_MSB is set during configuration of the DMAC.

The DMAC does not support locked or exclusive accesses and therefore **arlock[1:0]** is tied LOW.

A.2.5 Read data (AXI-R) channel signals

Table A-6 lists the AXI read data signals.

Table A-6 AXI-R signals

Signal	AMBA equivalent[a]
rdata[DATA_MSB:0][b]	**RDATA[DATA_MSB:0]**
rid[ID_MSB:0][b]	**RID[ID_MSB:0]**
rlast	**RLAST**
rready	**RREADY**
rresp[1:0]	**RRESP[1:0]**
rvalid	**RVALID**

a. See the *AMBA AXI Protocol Specification* for a description of these signals.
b. The value of DATA_MSB and ID_MSB are set during configuration of the DMAC.

A.3 APB signals

The DMAC provides the following APB interfaces:

- *Non-secure APB interface*
- *Secure APB interface*.

A.3.1 Non-secure APB interface

Table A-7 lists the signals that the non-secure APB interface provides.

Table A-7 Non-secure APB interface signals

Signal	AMBA equivalent[a]
paddr[31:0]	**PADDR**
penable	**PENABLE**
prdata[31:0]	**PRDATA**
pready	**PREADY**
psel	**PSELx**
pwdata[31:0]	**PWDATA**
pwrite	**PWRITE**

a. See the *AMBA 3 APB Protocol Specification* for a description of these signals.

A.3.2 Secure APB interface

Table A-8 lists the signals that the secure APB interface provides.

Table A-8 Secure APB interface signals

Signal	AMBA equivalent[a]
spaddr[31:0]	**PADDR**
spenable	**PENABLE**
sprdata[31:0]	**PRDATA**
spready	**PREADY**

Table A-8 Secure APB interface signals (continued)

Signal	AMBA equivalent[a]
spsel	**PSELx**
spwdata[31:0]	**PWDATA**
spwrite	**PWRITE**

a. See the *AMBA 3 APB Protocol Specification* for a description of these signals.

A.4 Peripheral request interface

Table A-9 lists the peripheral request interface signals that the DMAC provides, after you configure it to provide one or more peripheral request interfaces.

Note

You can configure the DMAC to contain no peripheral request interfaces. See the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* for more information.

Table A-9 Peripheral request interface

Name[a]	Type	Source/ destination	Description
daready_<x>	Input	Peripheral	Indicates if the peripheral can accept the information that the DMAC is providing on **datype_<x>[1:0]**: 0 = peripheral not ready 1 = peripheral ready.
datype_<x>[1:0]	Output	Peripheral	Indicates the type of acknowledgement, or request, that the DMAC is signaling: b00 = the DMAC has completed the single DMA transfer b01 = the DMAC has completed the burst DMA transfer b10 = DMAC requesting the peripheral to perform a flush request b11 = reserved.
davalid_<x>	Output	Peripheral	Indicates when the DMAC is providing valid control information: 0 = no control information is available 1 = **datype_<x>[1:0]** contains valid information for the peripheral.
drlast_<x>	Input	Peripheral	Indicates that the peripheral is sending the last data transfer for the current DMA transfer: 0 = last data request is not in progress 1 = last data request is in progress. **Note** The DMAC only uses this signal when **drtype_<x>[1:0]** is b00 or b01.

Table A-9 Peripheral request interface (continued)

Name[a]	**Type**	**Source/ destination**	**Description**
drready_<x>	Output	Peripheral	Indicates if the DMAC can accept the information that the peripheral is providing on **drtype_<x>[1:0]**: 0 = DMAC not ready. 1 = DMAC ready.

drtype_<x>[1:0]	Input	Peripheral	Indicates the type of acknowledgement, or request, that the peripheral is signaling:
			b00 = single level request
			b01 = burst level request
			b10 = acknowledging a flush request that the DMAC requested
			b11 = reserved.

drvalid_<x>	Input	Peripheral	Indicates when the peripheral is providing valid control information: 0 = no control information is available 1 = **drtype_<x>[1:0]** and **drlast_<x>** contain valid information for the DMAC.

a. Where <x> is the number for a peripheral request interface. The valid numbers for *x* depend on the configuration of the DMAC.

A.5 Interrupt signals

Table A-10 lists the interrupt signals.

Table A-10 Interrupt signals

Name	Type	Destination	Description
irq[x:0][a]	Output	Processor	Active HIGH interrupt output. The DMAC sets **irq<N>** HIGH when it executes a `DMASEV` instruction for event N, if the *Interrupt Enable Register* on page 3-13 is programmed to signal an interrupt for event N. Use the *Interrupt Clear Register* on page 3-17 to set **irq<N>** LOW.
irq_abort	Output	Processor	The DMAC sets this signal HIGH when an abort occurs and it remains HIGH if any thread is in the Faulting completing state or Faulting state. If all threads are in not in the Faulting completing state or Faulting state then the DMAC sets this signal LOW.

a. The valid numbers for *x* depend on the configuration of the DMAC.

A.6 Tie-off signals

Table A-11 lists the tie-off signals that all configurations of the DMAC contain.

Table A-11 DMAC tie-off signals

Name	Type	Source	Description
boot_addr[31:0]	Input	Tie-off	Configures the address location that contains the first instruction that the DMAC executes, when it exits from reset. **Note** The DMAC only uses this address when **boot_from_pc** is HIGH.
boot_from_pc	Input	Tie-off	Controls the location of where the DMAC executes its initial instruction, after it exits from reset: 0 = DMAC waits for an instruction from either APB interface 1 = DMA manager thread executes the instruction that is located at the address that **boot_addr[31:0]** provides.
boot_manager_ns	Input	Tie-off	When the DMAC exits from reset, this signal controls the security state of the DMA manager thread: 0 = assigns DMA manager to the Secure state 1 = assigns DMA manager to the Non-secure state.

Table A-12 lists the tie-off signals that control the security state of the interrupt outputs and peripheral request interfaces when the DMAC exits from reset.

Table A-12 Interrupt and peripheral tie-off signals

Name	Type	Source	Description
boot_irq_ns[x:0][a]	Input	Tie-off	Controls the security state of the interrupt outputs when the DMAC exits from reset: **boot_irq_ns[x] is LOW** The DMAC assigns **irq[x]** to the Secure state. **boot_irq_ns[x] is HIGH** The DMAC assigns **irq[x]** to the Non-secure state.
boot_periph_ns[x:0][a]	Input	Tie-off	Controls the security state of the peripheral request interface when the DMAC exits from reset: **boot_periph_ns[x] is LOW** The DMAC assigns peripheral request interface *x* to the Secure state. **boot_periph_ns[x] is HIGH** The DMAC assigns peripheral request interface *x* to the Non-secure state. **Note** Some configurations of the DMAC might not provide these signals because the DMAC does not contain a peripheral request interface. See *Peripheral request interface* on page A-9.

a. The width of this bus depends on the configuration of the DMAC. See the *AMBA Designer (FD001) PrimeCell DMA Controller (PL330) User Guide Supplement* for information about the bus widths that the DMAC permits.

Glossary

This glossary describes some of the terms used in technical documents from ARM.

Advanced eXtensible Interface (AXI)

A bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.

The AXI protocol also includes optional extensions to cover signaling for low-power operation.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced Microcontroller Bus Architecture (AMBA)

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that describes a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

Advanced Peripheral Bus (APB)

A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

Aligned

A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

AMBA

See Advanced Microcontroller Bus Architecture.

APB

See Advanced Peripheral Bus.

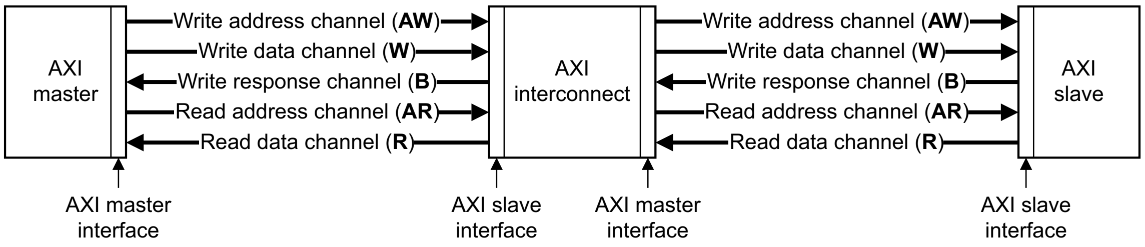
AXI

See Advanced eXtensible Interface.

AXI channel order and interfaces

The block diagram shows:

- the order in which AXI channel signals are described
- the master and slave interface conventions for AXI components.



AXI terminology

The following AXI terms are general. They apply to both masters and slaves:

Active read transaction

A transaction for which the read address has transferred, but the last read data has not yet transferred.

Active transfer

A transfer for which the **xVALID**[1] handshake has asserted, but for which **xREADY** has not yet asserted.

Active write transaction

A transaction for which the write address or leading write data has transferred, but the write response has not yet transferred.

Completed transfer	A transfer for which the **xVALID/xREADY** handshake is complete.
Payload	The non-handshake signals in a transfer.
Transaction	An entire burst of transfers, comprising an address, one or more data transfers and a response transfer (writes only).
Transmit	An initiator driving the payload and asserting the relevant **xVALID** signal.
Transfer	A single exchange of information. That is, with one **xVALID/xREADY** handshake.

The following AXI terms are master interface attributes. To obtain optimum performance, they must be specified for all components with an AXI master interface:

Combined issuing capability	The maximum number of active transactions that a master interface can generate. This is specified instead of write or read issuing capability for master interfaces that use a combined storage for active write and read transactions.
Read ID capability	The maximum number of different **ARID** values that a master interface can generate for all active read transactions at any one time.
Read ID width	The number of bits in the **ARID** bus.
Read issuing capability	The maximum number of active read transactions that a master interface can generate.
Write ID capability	The maximum number of different **AWID** values that a master interface can generate for all active write transactions at any one time.

1. The letter **x** in the signal name denotes an AXI channel as follows:

AW	Write address channel.
W	Write data channel.
B	Write response channel.
AR	Read address channel.
R	Read data channel.

Write ID width

The number of bits in the **AWID** and **WID** buses.

Write interleave capability

The number of active write transactions for which the master interface is capable of transmitting data. This is counted from the earliest transaction.

Write issuing capability

The maximum number of active write transactions that a master interface can generate.

The following AXI terms are slave interface attributes. To obtain optimum performance, they must be specified for all components with an AXI slave interface:

Combined acceptance capability

The maximum number of active transactions that a slave interface can accept. This is specified instead of write or read acceptance capability for slave interfaces that use a combined storage for active write and read transactions.

Read acceptance capability

The maximum number of active read transactions that a slave interface can accept.

Read data reordering depth

The number of active read transactions for which a slave interface can transmit data. This is counted from the earliest transaction.

Write acceptance capability

The maximum number of active write transactions that a slave interface can accept.

Write interleave depth

The number of active write transactions for which the slave interface can receive data. This is counted from the earliest transaction.

Beat Alternative word for an individual transfer within a burst. For example, an INCR4 burst comprises four beats.

See also Burst.

Big-endian Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Little-endian and Endianness.

Boundary scan chain
A boundary scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

Burst
A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AMBA are controlled using signals to indicate the length of the burst and how the addresses are incremented.

See also Beat.

Byte
An 8-bit data item.

Cache
A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

Cache hit
A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.

Cache line
The basic unit of storage in a cache. It is always a power of two words in size (usually four or eight words), and is required to be aligned to a suitable memory boundary.

Cache miss
A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.

Coherency
See Memory coherency.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

DMA
See Direct Memory Access.

Endianness
Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian

Halfword
A 16-bit data item.

Illegal instruction
An instruction that is architecturally Undefined.

Instruction cache	A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions. This is done to greatly increase the average speed of memory accesses and so improve processor performance.
Little-endian	Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory. *See also* Big-endian and Endianness.
Memory coherency	A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.
Microprocessor	*See* Processor.
Miss	*See* Cache miss.
Processor	A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.
Region	A partition of instruction or data memory space.
Reserved	A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.
Scan chain	A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.
Unaligned	A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.
Undefined	Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more information about ARM exceptions.
UNP	*See* Unpredictable.

Unpredictable	For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.
Word	A 32-bit data item.