AMBA® AXI and ACE

Protocol Specification

arm

ARM IHI 0022H (ID040120)

AMBA AXI and ACE
Protocol Specification

Release Information

The following changes have been made to this specification:

Change history

Date	Issue	Confidentiality	Change
16 June 2003	A	Non-Confidential	First release
19 March 2004	B	Non-Confidential	First release of AXI specification v1.0
03 March 2010	C	Non-Confidential	First release of AXI specification v2.0
03 June 2011	D-2c	Non-Confidential	Public beta draft of AMBA AXI and ACE Protocol Specification
28 October 2011	D	Non-Confidential	First release of AMBA AXI and ACE Protocol Specification
22 February 2013	E	Non-Confidential	Second release of AMBA AXI and ACE Protocol Specification
18 December 2017	F	Non-Confidential	EAC-0 release of version F. New interfaces defined for AMBA protocol: AXI5, AXI5-Lite, ACE5, ACE5-Lite, ACE5-LiteDVM, ACE5-LiteACP.
21 December 2017	F.b	Non-Confidential	EAC-1 release to address issues found with the EAC-0 release of release F. No change in content compared to the EAC-0 version.
30 July 2019	G	Non-Confidential	EAC-0 release of version G. New optional features defined for AMBA 5 interface variants.
31 March 2020	H	Non-Confidential	EAC-0 release of version H. New optional features defined for AMBA 5 interface variants.

Note that issue E.a, the first publication of issue E of this specification, was originally identified as issue E.

Issues B and C of this document included an AXI specification version, v1.0 and v2.0. These version numbers have been discontinued to remove confusion with the AXI versions AXI3 and AXI4.

ARM AMBA SPECIFICATION LICENCE

THIS END USER LICENCE AGREEMENT ("LICENCE") IS A LEGAL AGREEMENT BETWEEN YOU (EITHERASINGLE INDIVIDUAL, OR SINGLE LEGAL ENTITY) AND ARM LIMITED ("ARM") FOR THE USE OF THE RELEVANT AMBA SPECIFICATION ACCOMPANYING THIS LICENCE. ARM IS ONLY WILLING TO LICENSE THE RELEVANT AMBA SPECIFICATION TO YOU ON CONDITION THAT YOU ACCEPT ALL OF THE TERMS IN THIS LICENCE. BY CLICKING "I AGREE" OR OTHERWISE USING OR COPYING THE RELEVANT AMBA SPECIFICATION YOU INDICATE THAT YOU AGREE TO BE BOUND BY ALL THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENCE, ARM IS UNWILLING TO LICENSE THE RELEVANT AMBA SPECIFICATION TO YOU AND YOU MAY NOT USE OR COPY THE RELEVANT AMBA SPECIFICATION AND YOU SHOULD PROMPTLY RETURN THE RELEVANT AMBA SPECIFICATION TO ARM. "LICENSEE" means You and your Subsidiaries.

"Subsidiary" means, if You are a single entity, any company the majority of whose voting shares is now or hereafter owned or controlled, directly or indirectly, by You. A company shall be a Subsidiary only for the period during which such control exists.

1. Subject to the provisions of Clauses 2, 3 and 4, Arm hereby grants to LICENSEE a perpetual, non-exclusive, non-transferable, royalty free, worldwide licence to:
(i) use and copy the relevant AMBA Specification for the purpose of developing and having developed products that comply with the relevant AMBA Specification;
(ii) manufacture and have manufactured products which either: (a) have been created by or for LICENSEE under the licence granted in Clause 1(i); or (b) incorporate a product(s) which has been created by a third party(s) under a licence granted by Arm in Clause 1(i) of such third party's Arm AMBA Specification Licence; and
(iii) offer to sell, sell, supply or otherwise distribute products which have either been (a) created by or for LICENSEE under the licence granted in Clause 1(i); or (b) manufactured by or for LICENSEE under the licence granted in Clause 1(ii).

2. LICENSEE hereby agrees that the licence granted in Clause 1 is subject to the following restrictions:
(i) where a product created under Clause 1(i) is an integrated circuit which includes a CPU then either: (a) such CPU shall only be manufactured under licence from Arm; or (b) such CPU is neither substantially compliant with nor marketed as being compliant with the Arm instruction sets licensed by Arm from time to time;
(ii) the licences granted in Clause 1(iii) shall not extend to any portion or function of a product that is not itself compliant with part of the relevant AMBA Specification; and
(iii) no right is granted to LICENSEE to sublicense the rights granted to LICENSEE under this Agreement.

3. Except as specifically licensed in accordance with Clause 1, LICENSEE acquires no right, title or interest in any Arm technology or any intellectual property embodied therein. In no event shall the licences granted in accordance with Clause 1 be construed as granting LICENSEE, expressly or by implication, estoppel or otherwise, a licence to use any Arm technology except the relevant AMBA Specification.

4. THE RELEVANT AMBA SPECIFICATION IS PROVIDED "AS IS" WITH NO REPRESENTATION OR WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT ANY USE OR IMPLEMENTATION OF SUCH ARM TECHNOLOGY WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER INTELLECTUAL PROPERTY RIGHTS.

5. NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS AGREEMENT, TO THE FULLEST EXTENT PETMITTED BY LAW, THE MAXIMUM LIABILITY OF ARM IN AGGREGATE FOR ALL CLAIMS MADE AGAINST ARM, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS AGREEMENT (INCLUDING WITHOUT LIMITATION (I) LICENSEE'S USE OF THE ARM TECHNOLOGY; AND (II) THE IMPLEMENTATION OF THE ARM TECHNOLOGY IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS AGREEMENT) SHALL NOT EXCEED THE FEES PAID (IF ANY) BY LICENSEE TO ARM UNDER THIS AGREEMENT. THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

6. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the Arm tradename, or AMBA trademark in connection with the relevant AMBA Specification or any products based thereon. Nothing in Clause 1 shall be construed as authority for LICENSEE to make any representations on behalf of Arm in respect of the relevant AMBA Specification.

7. This Licence shall remain in force until terminated by you or by Arm. Without prejudice to any of its other rights if LICENSEE is in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to You. You may terminate this Licence at any time. Upon expiry or termination of this Licence by You or by Arm LICENSEE shall stop using the relevant AMBA Specification and destroy all copies of the relevant AMBA Specification in your possession together with all documentation and related materials. Upon expiry or termination of this Licence, the provisions of clauses 6 and 7 shall survive.

8. The validity, construction and performance of this Agreement shall be governed by English Law.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

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

Web Address

`http://www.arm.com`

Contents
AMBA AXI and ACE Protocol Specification

Part G Appendices

Preface

This preface introduces the *AMBA AXI and ACE Protocol Specification*. It contains the following sections:

About this specification

This specification describes the AMBA protocols for AXI and ACE. Several release levels and variants are described:

- The AMBA 3 AXI protocol release is referred to as AXI3.
- The AMBA 4 AXI protocol releases are referred to as AXI4 and AXI4-Lite.
- The AMBA 5 AXI protocol releases are referred to as AXI5 and AXI5-Lite.
- The AMBA 4 ACE protocol releases are referred to as ACE and ACE-Lite.
- The AMBA 5 ACE protocol releases are referred to as:
  - ACE5
  - ACE5-Lite
  - ACE5-LiteDVM
  - ACE5-LiteACP

Specifications for AXI3 apply to all subsequent versions, with some exceptions. AXI4 extends AXI3 and has some protocol changes. AXI4 protocol differences are marked appropriately.

AMBA 5 introduces new features that are generally optional additions to the previous AMBA 4 specifications. Those working with new designs are encouraged to use the AMBA 5 family of interfaces.

Early issues of this document describe earlier versions of the AMBA AXI Protocol Specification. In particular, Issue B of the document describes the version that is now called AXI3.

Issue C adds the definition of an extended version of the protocol that is called AXI4 and a new interface, AXI4-Lite.

Issue D integrates the definitions of AXI3 and AXI4 that were presented separately in Issue C.

Issue E adds clarifications, recommendations, and specifies new capabilities. To maintain compatibility, a property is used to declare a new capability.

Issue F defines new AMBA 5 interfaces: ACE5, ACE5-Lite, ACE5-LiteDVM, ACE5-LiteACP, AXI5, AXI5-Lite.

Issue G and Issue H add new optional features that are defined for AMBA 5 interface variants.

Note

Some previous issues of this document included a version number in the title. That version number does not refer to the version of the AXI protocol.

Intended audience

This specification is written for hardware and software engineers who want to become familiar with AMBA and design systems and modules that are compatible with the AXI protocol.

Using this specification

The information in this specification is organized into parts, as described in this section.

If using an AXI3 or AXI4 interface, the following sections should be read:

- Part A *AMBA AXI Protocol Specification*

If using an AXI4-Lite interface, the following sections should be read:

- Part A *AMBA AXI Protocol Specification*
- Part B *AMBA AXI4-Lite Interface Specification*

If using either AXI5 or AXI5-Lite interfaces, the following sections should be read:

- Part A *AMBA AXI Protocol Specification*
- Part C *AMBA AXI5 and AXI5-Lite Interface Specification*
- Part E *AMBA 5 Protocol Features*

ACE and ACE-Lite interfaces are an extension of AXI interfaces. If using ACE or ACE-Lite types of interface, the following sections should be read:

- Part A *AMBA AXI Protocol Specification*
- Part D *AMBA ACE and ACE-Lite Protocol Specification*
- Part E *AMBA 5 Protocol Features*
- Part F *AMBA ACE5, ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACPInterface Specification*

Those already familiar with this specification need only read the new and changed sections of this specification. Refer to Appendix G7 *Revisions*.

Part A, AMBA AXI Protocol Specification

Part A describes the AXI architecture, protocol and signaling. This is common to all interfaces described in this specification. AXI3 and AXI4 interfaces are fully described in this part. It contains the following chapters:

Chapter A1 *Introduction*

An introduction to the AXI architecture and terminology that is used in this specification.

Chapter A2 *Signal Descriptions*

A description of the signals that are used by the AXI3 and AXI4 protocols.

Chapter A3 *Single Interface Requirements*

A description of the basic AXI protocol transaction requirements between a master and slave.

Chapter A4 *Transaction Attributes*

A description of the AXI protocol and signaling that supports system topology and system level caches.

Chapter A5 *Transaction Identifiers*

A description of the AXI protocol and signaling that supports out-of-order transaction completion and the issuing of multiple outstanding addresses.

Chapter A6 *AXI Ordering Model*

A description of the AXI ordering model.

Chapter A7 *Atomic Accesses*

A description of the mechanisms that support atomic accesses.

Chapter A8 *AMBA 4 Additional Signaling*

A description of the additional signaling introduced in AXI4 to extend the application of the AXI interface.

Chapter A9 *Default Signaling and Interoperability*

A description of the interoperability of interfaces that use reduced AXI signal sets.

Part B, AMBA AXI4-Lite Interface Specification

Part B describes AMBA AXI4-Lite. It contains the following chapter:

Chapter B1 *AMBA AXI4-Lite*

A description of AXI4-Lite that provides a simpler control register-style interface for systems that do not require the full functionality of AXI4.

Part C, AMBA AXI5 and AXI5-Lite Interface Specification

Part C describes the signals and features for AXI5 and AXI5-Lite interfaces. It contains the following chapters:

Chapter C1 *AMBA AXI5*

An overview of the new capabilities, the set of properties that specify the supported behavior, and the AXI5 interface signaling requirements.

Chapter C2 *AMBA AXI5-Lite*

An overview of the new capabilities, the set of properties that specify the supported behavior, and the AXI5-Lite interface signaling requirements.

Part D, AMBA ACE and ACE-Lite Protocol Specification

Part D describes the ACE protocol. It contains the following chapters:

Chapter D1 *About ACE*

An overview of system level coherency and the architecture of the *AXI Coherency Extensions* (ACE) protocol.

Chapter D2 *Signal Descriptions*

A description of the additional ACE interface signals.

Chapter D3 *Channel Signaling*

A description of the basic channel signaling requirements on an ACE interface.

Chapter D4 *Coherency Transactions on the Read Address and Write Address Channels*

A description of the transactions issued on the read address and write address channels.

Chapter D5 *Snoop Transactions*

A description of the snoop transactions seen on the snoop address channel.

Chapter D6 *Interconnect Requirements*

A description of the ACE interconnect requirements.

Chapter D7 *Cache Maintenance*

A description of the ACE cache maintenance operations.

Chapter D8 *Barrier Transactions*

A description of the ACE memory and synchronization barrier transactions.

Chapter D9 *Exclusive Accesses from ACE Masters*

A description of the ACE Exclusive Accesses to Shareable memory.

Chapter D10 *Optional External Snoop Filtering*

A description of using an external snoop filter in an ACE system.

Chapter D11 *AMBA ACE-Lite*

A description of the ACE-Lite interface.

Chapter D12 *Interface Control*

A description of the optional signals that can be used to configure the ACE interface.

Chapter D13 *Distributed Virtual Memory Transactions*

A description of *Distributed Virtual Memory* (DVM) transactions.

Chapter D14 *Master Design Recommendations*

A set of recommendations for the design of master components that improve the ability to bridge the master to different protocol interfaces.

Part E AMBA 5 Protocol Features

Part E describes changes to the AMBA family of interfaces in version 5. It contains the following chapters:

Chapter E1 *Additional Features in AMBA 5*

A description of the new features in AMBA 5.

Chapter E2 *Interface and data protection*

Specification of schemes for the protection of data and interfaces with the addition of poison and parity signaling.

Part F AMBA ACE5, ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACP Interface Specification

Part F describes the ACE5 and ACE5-Lite family of interfaces. It contains the following chapters:

Chapter F1 *AMBA ACE5*

An overview of the new capabilities, the set of properties that specify the supported behavior, and the ACE5 interface signaling requirements.

Chapter F2 *AMBA ACE5-Lite*

A description of the new capabilities in the ACE5-Lite protocol specification.

Chapter F3 *AMBA ACE5-LiteDVM*

A description of the new ACE5-LiteDVM protocol specification introduced in AMBA 5.

Chapter F4 *ACE5-LiteACP*

A description of the ACE5-LiteACP protocol specification introduced in AMBA 5.

Chapter F5 *Changes in ACE5 and ACE5-Lite*

A description of the changes in AMBA 5 to the ACE and ACE-Lite channel signaling requirements.

Part G Appendices

This specification contains the following appendices:

Appendix G1 *Transaction Naming*

This appendix defines the naming scheme for full cache line and partial cache line write transactions.

Appendix G2 *Signal Lists*

This appendix defines the required and optional signals for each of the AMBA 5 interfaces.

Appendix G3 *Interface Property Summary*

This appendix defines properties of the AMBA 5 interfaces and when each interface property was introduced into this specification.

Appendix G4 *Summary of AxSNOOP encodings*

This appendix shows all possible **AxSNOOP** encodings and the property that is used to determine if a particular value is supported for a given interface.

Appendix G5 *Summary of ID constraints*

This appendix lists the restrictions on ID usage that are specified in this document.

Appendix G6 *Summary of response codes*

This appendix contains a summary of read and write response codes.

Appendix G7 *Revisions*

This appendix describes the technical changes between released issues of this specification.

Conventions

The following sections describe conventions that this specification can use:

- *Typographic conventions*
- *Timing diagrams*
- *Signals* on page xx
- *Numbers* on page xx

Typographic conventions

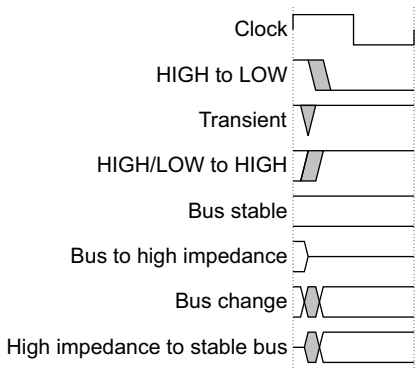
The typographical conventions are:

italic	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
bold	Denotes signal names, and is used for terms in descriptive lists, where appropriate.
`monospace`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
SMALL CAPITALS	Used for a few terms that have specific technical meanings.

Timing diagrams

The figure named *Key to timing diagram conventions* 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

Timing diagrams sometimes show single-bit signals as HIGH and LOW at the same time and they look similar to the bus change shown in *Key to timing diagram conventions*. If a timing diagram shows a single-bit signal in this way then its value does not affect the accompanying description.

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.

Lower-case x At the second letter of a signal name denotes a collective term for both Read and Write. For example, **AxCACHE** refers to both the **ARCACHE** and **AWCACHE** signals.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. Both are written in a `monospace` font.

Additional reading

This section lists relevant publications from Arm.

See Arm Developer https://developer.arm.com/docs, for access to Arm documentation.

Arm publications

- *AMBA APB Protocol Specification* (ARM IHI 0024)
- *AMBA 4 AXI4-Stream Protocol Specification* (ARM IHI 0051)
- *AMBA 5 CHI Architecture Specification* (ARM IHI 0050)
- *AMBA Low Power Interface Specification* (ARM IHI 0068)
- *Arm® Architecture Reference Manual Armv8*, for Armv8-A architecture profile (ARM DDI 0487)
- *Arm®Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A* (ARM DDI 0598)
- *Arm System Memory Management Unit Architecture Specification* (ARM IHI 0070)

Feedback

Arm welcomes feedback on its documentation.

Feedback on this specification

If you have comments on the content of this specification, send e-mail to `errata@arm.com`. Give:

- The title, AMBA AXI and ACE Protocol Specification
- The number, ARM IHI 0022H
- The page number(s) that your comments apply
- A concise explanation of your comments

Arm also welcomes general suggestions for additions and improvements.

Part A

AMBA AXI Protocol Specification

Chapter A1
Introduction

This chapter introduces the architecture of the AXI protocol and the terminology that is used in this specification:

A1.1 About the AXI protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between master and slave components.

The AXI protocol features are:

- It is suitable for high-bandwidth and low-latency designs.
- High-frequency operation is provided, without using complex bridges.
- The protocol meets the interface requirements of a wide range of components.
- It is suitable for memory controllers with high initial access latency.
- Flexibility in the implementation of interconnect architectures is provided.
- It is backward-compatible with AHB and APB interfaces.

The key features of the AXI protocol are:

- Separate address/control and data phases.
- Support for unaligned data transfers, using byte strobes.
- Uses burst-based transactions with only the start address issued.
- Separate read and write data channels, that can provide low-cost *Direct Memory Access* (DMA).
- Support for issuing multiple outstanding addresses.
- Support for out-of-order transaction completion.
- Permits easy addition of register stages to provide timing closure.

The AXI protocol includes:

- AXI4-Lite, a subset of AXI4 for communication with simpler control register style interfaces within components. See Chapter B1 *AMBA AXI4-Lite*.
- AXI5-Lite, a subset of AXI5 for using AXI5 features with simpler control register style interfaces within components. See Chapter C2 *AMBA AXI5-Lite*.

A1.2 AXI Architecture

The AXI protocol is burst-based and defines five independent transaction channels:

- Read address, which has signal names beginning with **AR**.
- Read data, which has signal names beginning with **R**.
- Write address, which has signal names beginning with **AW**.
- Write data, which has signal names beginning with **W**.
- Write response, which has signal names beginning with **B**.

An address channel carries control information that describes the nature of the data to be transferred. The data is transferred between master and slave using either:

- A write data channel to transfer data from the master to the slave. In a write transaction, the slave uses the write response channel to signal the completion of the transfer to the master.
- A read data channel to transfer data from the slave to the master.

The AXI protocol:

- Permits address information to be issued ahead of the actual data transfer.
- Supports multiple outstanding transactions.
- Supports out-of-order completion of transactions.

Figure A1-1 shows how a write transaction uses the write address, write data, and write response channels.

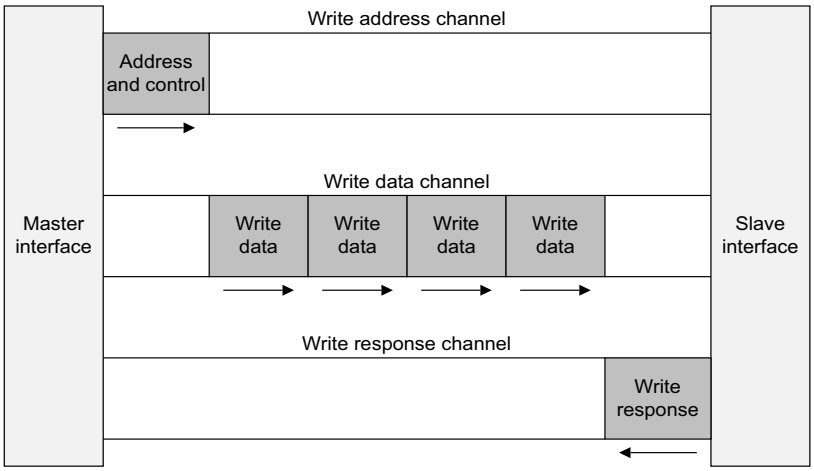


Figure A1-1 Channel architecture of writes

Figure A1-2 shows how a read transaction uses the read address and read data channels.

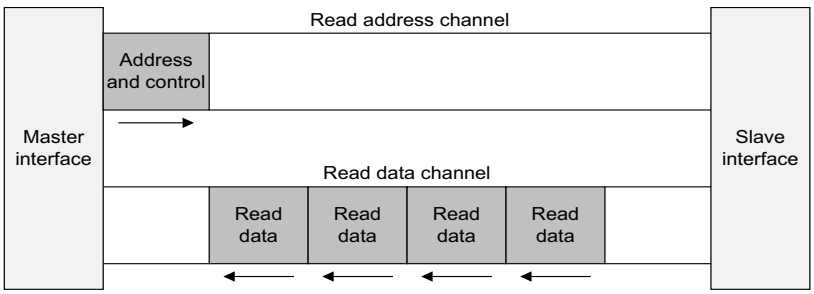


Figure A1-2 Channel architecture of reads

A1.2.1 Channel definition

Each of the five independent channels consists of a set of information signals and **VALID** and **READY** signals that provide a two-way handshake mechanism. See *Basic read and write transactions* on page A3-41.

The information source uses the **VALID** signal to show when valid address, data, or control information is available on the channel. The destination uses the **READY** signal to show when it can accept the information. Both the read data channel and the write data channel also include a **LAST** signal to indicate the transfer of the final data item in a transaction.

Read and write address channels

Read and write transactions each have their own address channel. The appropriate address channel carries all the required address and control information for a transaction.

Read data channel

The read data channel carries both the read data and the read response information from the slave to the master, and includes:

- The data bus, which can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
- A read response signal indicating the completion status of the read transaction.

Write data channel

The write data channel carries the write data from the master to the slave and includes:

- The data bus, which can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
- A byte lane strobe signal for every eight data bits, indicating the bytes of the data that are valid.

Write data channel information is always treated as buffered, so that the master can perform write transactions without slave acknowledgement of previous write transactions.

Write response channel

A slave uses the write response channel to respond to write transactions. All write transactions require completion signaling on the write response channel.

As Figure A1-1 on page A1-27 shows, completion is signaled only for a complete transaction, not for each data transfer in a transaction.

A1.2.2 Interface and interconnect

A typical system consists of several master and slave devices that are connected together through some form of interconnect, as Figure A1-3 shows.

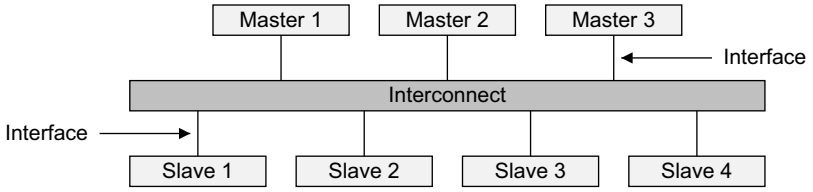


Figure A1-3 Interface and interconnect

The AXI protocol provides a single interface definition, for the interfaces between:

- A master and the interconnect
- A slave and the interconnect
- A master and a slave

This interface definition supports many different interconnect implementations.

Note

An interconnect between devices is equivalent to another device with symmetrical master and slave ports that the real master and slave devices can be connected.

Typical system topologies

Most systems use one of three interconnect topologies:

- Shared address and data buses
- Shared address buses and multiple data buses
- Multilayer, with multiple address and data buses

In most systems, the address channel bandwidth requirement is significantly less than the data channel bandwidth requirement. Such systems can achieve a good balance between system performance and interconnect complexity by using a shared address bus with multiple data buses to enable parallel data transfers.

A1.2.3 Register slices

Each AXI channel transfers information in only one direction, and the architecture does not require any fixed relationship between the channels. These qualities mean that a register slice can be inserted at almost any point in any channel, at the cost of an additional cycle of latency.

Note

These qualities make the following possible:

- Trade-off between cycles of latency and maximum frequency of operation.
- Direct, fast connection between a processor and high-performance memory, but to use simple register slices to isolate a longer path to less performance critical peripherals.

A1.3 Terminology

This section summarizes terms that are used in this specification, and are defined in the *Glossary*, or elsewhere. Where appropriate, terms that are listed in this section link to the corresponding glossary definition.

A1.3.1 AXI components and topology

The following terms describe AXI components:

- *Component*
- *Master component*
- *Slave component*, which includes *Memory slave component*s and *Peripheral slave component*s
- *Interconnect component*

For a particular AXI transaction, *Upstream* and *Downstream* refer to the relative positions of AXI components within the AXI topology.

A1.3.2 AXI transactions, and memory types

When an AXI master initiates an AXI operation, targeting an AXI slave:

- The complete set of required operations on the AXI bus form the AXI *Transaction*
- Any required payload data is transferred as an AXI *Burst*
- A burst can comprise multiple data transfers, or AXI *Beat*s

A1.3.3 Caches and cache operation

This specification does not define standard cache terminology, that is defined in any reference work on caching. However, the glossary entries for *Cache* and *Cache line* clarify how these terms are used in this document.

A1.3.4 Temporal description

The AXI specification uses the term *in a timely manner*.

Chapter A2
Signal Descriptions

This chapter introduces the AXI interface signals. Most of the signals are required for AXI3 and AXI4 implementations of the protocol, and the tables summarizing the signals identify the exceptions. This chapter contains the following sections:

Later chapters define the signal parameters and usage.

A2.1 Global signals

Table A2-1 shows the global AXI signals. These signals are used by the AXI3 and AXI4 protocols.

Table A2-1 Global signals

Signal	Source	Description
ACLK	Clock source	Global clock signal. Synchronous signals are sampled on the rising edge of the global clock. See *Clock* on page A3-40.
ARESETn	Reset source	Global reset signal. This signal is active-LOW. See *Reset* on page A3-40.

All signals are sampled on the rising edge of the global clock.

A2.2 Write address channel signals

Table A2-2 shows the AXI write address channel signals. Unless the description indicates otherwise, a signal is used by AXI3 and AXI4.

Table A2-2 Write address channel signals

Signal	Source	Description
AWID	Master	Identification tag for a write transaction. See *ID signals* on page A5-81.
AWADDR	Master	The address of the first transfer in a write transaction. See *Address structure* on page A3-48.
AWLEN	Master	Length, the exact number of data transfers in a write transaction. This information determines the number of data transfers associated with the address. This changes between AXI3 and AXI4. See *Burst length* on page A3-48.
AWSIZE	Master	Size, the number of bytes in each data transfer in a write transaction. See *Burst size* on page A3-49.
AWBURST	Master	Burst type, indicates how address changes between each transfer in a write transaction. See *Burst type* on page A3-49.
AWLOCK	Master	Provides information about the atomic characteristics of a write transaction. This changes between AXI3 and AXI4. See *Locked accesses* on page A7-99.
AWCACHE	Master	Indicates how a write transaction is required to progress through a system. See *Memory types* on page A4-69.
AWPROT	Master	Protection attributes of a write transaction: privilege, security level, and access type. See *Access permissions* on page A4-75.
AWQOS	Master	Quality of Service identifier for a write transaction. Not implemented in AXI3. See *QoS signaling* on page A8-102.
AWREGION	Master	Region indicator for a write transaction. Not implemented in AXI3. See *Multiple region signaling* on page A8-103.
AWUSER	Master	User-defined extension for the write address channel. Not implemented in AXI3. See *User-defined signaling* on page A8-104.
AWVALID	Master	Indicates that the write address channel signals are valid. See *Channel handshake signals* on page A3-42.
AWREADY	Slave	Indicates that a transfer on the write address channel can be accepted. See *Channel handshake signals* on page A3-42.

A2.3 Write data channel signals

Table A2-3 shows the AXI write data channel signals. Unless the description indicates otherwise, a signal is used by AXI3 and AXI4.

Table A2-3 Write data channel signals

Signal	Source	Description
WID	Master	The ID tag of the write data transfer. Implemented in AXI3 only. See *ID signals* on page A5-81.
WDATA	Master	Write data. See *Write data channel* on page A3-43.
WSTRB	Master	Write strobes, indicate which byte lanes hold valid data. See *Write strobes* on page A3-54.
WLAST	Master	Indicates whether this is the last data transfer in a write transaction. See *Write data channel* on page A3-43.
WUSER	Master	User-defined extension for the write data channel. Not implemented in AXI3. See *User-defined signaling* on page A8-104.
WVALID	Master	Indicates that the write data channel signals are valid. See *Channel handshake signals* on page A3-42.
WREADY	Slave	Indicates that a transfer on the write data channel can be accepted. See *Channel handshake signals* on page A3-42.

A2.4 Write response channel signals

Table A2-4 shows the AXI write response channel signals. Unless the description indicates otherwise, a signal is used by AXI3 and AXI4.

Table A2-4 Write response channel signals

Signal	Source	Description
BID	Slave	Identification tag for a write response. See *ID signals* on page A5-81.
BRESP	Slave	Write response, indicates the status of a write transaction. See *Read and write response structure* on page A3-59.
BUSER	Slave	User-defined extension for the write response channel. Not implemented in AXI3. See *User-defined signaling* on page A8-104.
BVALID	Slave	Indicates that the write response channel signals are valid. See *Channel handshake signals* on page A3-42.
BREADY	Master	Indicates that a transfer on the write response channel can be accepted. See *Channel handshake signals* on page A3-42.

A2.5 Read address channel signals

Table A2-5 shows the AXI read address channel signals. Unless the description indicates otherwise, a signal is used by AXI3 and AXI4.

Table A2-5 Read address channel signals

Signal	Source	Description
ARID	Master	Identification tag for a read transaction. See *ID signals* on page A5-81.
ARADDR	Master	The address of the first transfer in a read transaction. See *Address structure* on page A3-48.
ARLEN	Master	Length, the exact number of data transfers in a read transaction. This changes between AXI3 and AXI4. See *Burst length* on page A3-48.
ARSIZE	Master	Size, the number of bytes in each data transfer in a read transaction. See *Burst size* on page A3-49.
ARBURST	Master	Burst type, indicates how address changes between each transfer in a read transaction. See *Burst type* on page A3-49.
ARLOCK	Master	Provides information about the atomic characteristics of a read transaction. This changes between AXI3 and AXI4. See *Locked accesses* on page A7-99.
ARCACHE	Master	Indicates how a read transaction is required to progress through a system. See *Memory types* on page A4-69.
ARPROT	Master	Protection attributes of a read transaction: privilege, security level, and access type. See *Access permissions* on page A4-75.
ARQOS	Master	Quality of Service identifier for a read transaction. Not implemented in AXI3. See *QoS signaling* on page A8-102.
ARREGION	Master	Region indicator for a read transaction. Not implemented in AXI3. See *Multiple region signaling* on page A8-103.
ARUSER	Master	User-defined extension for the read address channel. Not implemented in AXI3. See *User-defined signaling* on page A8-104.
ARVALID	Master	Indicates that the read address channel signals are valid. See *Channel handshake signals* on page A3-42.
ARREADY	Slave	Indicates that a transfer on the read address channel can be accepted. See *Channel handshake signals* on page A3-42.

A2.6 Read data channel signals

Table A2-6 shows the AXI read data channel signals. Unless the description indicates otherwise, a signal is used by AXI3 and AXI4.

Table A2-6 Read data channel signals

Signal	Source	Description
RID	Slave	Identification tag for read data and response. See *ID signals* on page A5-81.
RDATA	Slave	Read data. See *Read data channel* on page A3-43.
RRESP	Slave	Read response, indicates the status of a read transfer. See *Read and write response structure* on page A3-59.
RLAST	Slave	Indicates whether this is the last data transfer in a read transaction. See *Read data channel* on page A3-43.
RUSER	Slave	User-defined extension for the read data channel. Not implemented in AXI3. See *User-defined signaling* on page A8-104.
RVALID	Slave	Indicates that the read data channel signals are valid. See *Channel handshake signals* on page A3-42.
RREADY	Master	Indicates that a transfer on the read data channel can be accepted. See *Channel handshake signals* on page A3-42.

Chapter A3
Single Interface Requirements

This chapter describes the basic AXI protocol transaction requirements between a single master and slave. It contains the following sections:

A3.1 Clock and reset

This section describes the requirements for implementing the AXI global clock and reset signals **ACLK** and **ARESETn**.

A3.1.1 Clock

Each AXI interface has a single clock signal, **ACLK**. All input signals are sampled on the rising edge of **ACLK**. All output signal changes can only occur after the rising edge of **ACLK**.

On master and slave interfaces, there must be no combinatorial paths between input and output signals.

A3.1.2 Reset

The AXI protocol uses a single active-LOW reset signal, **ARESETn**. The reset signal can be asserted asynchronously, but deassertion can only be synchronous with a rising edge of **ACLK**.

During reset the following interface requirements apply:

- A master interface must drive **ARVALID**, **AWVALID**, and **WVALID** LOW.
- A slave interface must drive **RVALID** and **BVALID** LOW.
- All other signals can be driven to any value.

The earliest point after reset that a master is permitted to begin driving **ARVALID**, **AWVALID**, or **WVALID** HIGH is at a rising **ACLK** edge after **ARESETn** is HIGH. Figure A3-1 shows the earliest point after reset that **ARVALID**, **AWVALID**, or **WVALID**, can be driven HIGH.

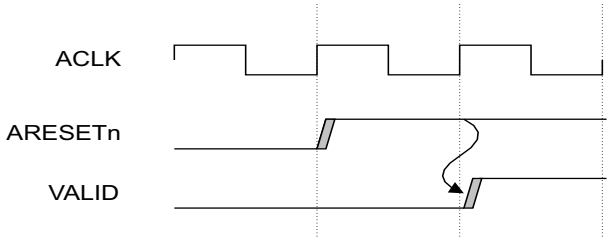


Figure A3-1 Exit from reset

A3.2 Basic read and write transactions

This section defines the basic mechanisms for AXI protocol transactions. The basic mechanisms are:

- The *Handshake process*
- The *Channel signaling requirements* on page A3-42

A3.2.1 Handshake process

All five transaction channels use the same **VALID**/**READY** handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate that the information moves between master and slave. The *source* generates the **VALID** signal to indicate when the address, data, or control information is available. The *destination* generates the **READY** signal to indicate that it can accept the information. Transfer occurs only when *both* the **VALID** and **READY** signals are HIGH.

On master and slave interfaces, there must be no combinatorial paths between input and output signals.

Figure A3-2 to Figure A3-4 on page A3-42 show examples of the handshake process.

The source presents information after T1 and asserts the **VALID** signal as shown in Figure A3-2. The destination asserts the **READY** signal after T2. The source must keep its information stable until the transfer occurs at T3, when this assertion is recognized.

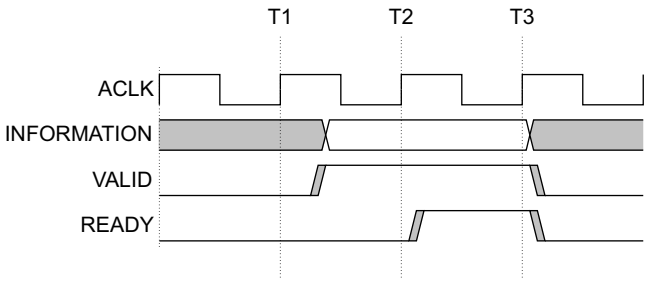


Figure A3-2 VALID before READY handshake

A source is not permitted to wait until **READY** is asserted before asserting **VALID**.

When **VALID** is asserted, it must remain asserted until the handshake occurs, at a rising clock edge when **VALID** and **READY** are both asserted.

In Figure A3-3 the destination asserts **READY** after T1, before the address, data, or control information is valid. This assertion indicates that it can accept the information. The source presents the information and asserts **VALID** after T2, then the transfer occurs at T3, when this assertion is recognized. In this case, transfer occurs in a single cycle.

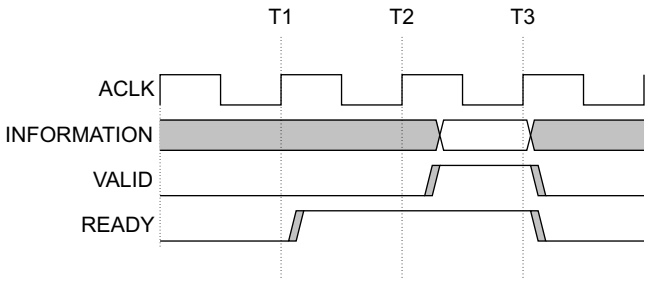


Figure A3-3 READY before VALID handshake

A destination is permitted to wait for **VALID** to be asserted before asserting the corresponding **READY**.

If **READY** is asserted, it is permitted to deassert **READY** before **VALID** is asserted.

In Figure A3-4, both the source and destination happen to indicate that they can transfer the address, data, or control information after T1. In this case, the transfer occurs at the rising clock edge when the assertion of both **VALID** and **READY** can be recognized. These assertions means that the transfer occurs at T2.

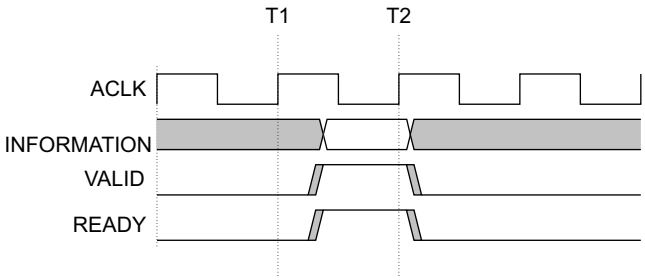


Figure A3-4 VALID with READY handshake

The individual AXI protocol channel handshake mechanisms are described in *Channel signaling requirements*.

A3.2.2 Channel signaling requirements

The following sections define the handshake signals and the handshake rules for each channel:

- *Channel handshake signals*
- *Write address channel*
- *Write data channel* on page A3-43
- *Write response channel* on page A3-43
- *Read address channel* on page A3-43
- *Read data channel* on page A3-43

Channel handshake signals

Each channel has its own **VALID**/**READY** handshake signal pair. Table A3-1 shows the signals for each channel.

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	**AWVALID**, **AWREADY**
Write data channel	**WVALID**, **WREADY**
Write response channel	**BVALID**, **BREADY**
Read address channel	**ARVALID**, **ARREADY**
Read data channel	**RVALID**, **RREADY**

Write address channel

The master can assert the **AWVALID** signal only when it drives valid address and control information. When asserted, **AWVALID** must remain asserted until the rising clock edge after the slave asserts **AWREADY**.

The default state of **AWREADY** can be either HIGH or LOW. This specification recommends a default state of HIGH. When **AWREADY** is HIGH, the slave must be able to accept any valid address that is presented to it.

Note

This specification does not recommend a default **AWREADY** state of LOW, because it forces the transfer to take at least two cycles, one to assert **AWVALID** and another to assert **AWREADY**.

Write data channel

During a write burst, the master can assert the **WVALID** signal only when it drives valid write data. When asserted, **WVALID** must remain asserted until the rising clock edge after the slave asserts **WREADY**.

The default state of **WREADY** can be HIGH, but only if the slave can always accept write data in a single cycle.

The master must assert the **WLAST** signal while it is driving the final write transfer in the burst.

This specification recommends that **WDATA** is driven to zero for inactive byte lanes.

Write response channel

The slave can assert the **BVALID** signal only when it drives a valid write response. When asserted, **BVALID** must remain asserted until the rising clock edge after the master asserts **BREADY**.

The default state of **BREADY** can be HIGH, but only if the master can always accept a write response in a single cycle.

Read address channel

The master can assert the **ARVALID** signal only when it drives valid address and control information. When asserted, **ARVALID** must remain asserted until the rising clock edge after the slave asserts the **ARREADY** signal.

The default state of **ARREADY** can be either HIGH or LOW. This specification recommends a default state of HIGH. If **ARREADY** is HIGH, then the slave must be able to accept any valid address that is presented to it.

Note

This specification does not recommend a default **ARREADY** value of LOW, because it forces the transfer to take at least two cycles, one to assert **ARVALID** and another to assert **ARREADY**.

Read data channel

The slave can assert the **RVALID** signal only when it drives valid read data. When asserted, **RVALID** must remain asserted until the rising clock edge after the master asserts **RREADY**. Even if a slave has only one source of read data, it must assert the **RVALID** signal only in response to a request for data.

The master interface uses the **RREADY** signal to indicate that it accepts the data. The default state of **RREADY** can be HIGH, but only if the master is able to accept read data immediately when it starts a read transaction.

The slave must assert the **RLAST** signal when it is driving the final read transfer in the burst.

This specification recommends that **RDATA** is driven to zero for inactive byte lanes.

A3.3 Relationships between the channels

The AXI protocol requires the following relationships to be maintained:

- A write response must always follow the last write transfer in a write transaction.
- Read data must always follow the read address of the data.
- Channel handshakes must conform to the dependencies defined in *Dependencies between channel handshake signals*.

The protocol does not define any other relationship between the channels.

The lack of relationship means, for example, that the write data can appear at an interface before the write address for the transaction. This can occur if the write address channel contains more register stages than the write data channel. Similarly, the write data might appear in the same cycle as the address.

Note

When the interconnect is required to determine the destination address space or slave space, it must realign the address and write data. This realignment is required to assure that the write data is signaled as being valid only to the slave that it is destined for.

When a master issues a write request, it must be able to provide all write data for that transaction, without dependency on other transactions from that master.

When a master issues a read request, it must be able to accept all read data for that transaction, without dependency on other transactions from that master.

Note that a master can rely on read data returning in order from transactions that use the same ID, so the master only needs enough storage for read data from transactions with different IDs.

A3.3.1 Dependencies between channel handshake signals

To prevent a deadlock situation, the dependency rules that exist between the handshake signals must be observed.

As summarized in *Channel signaling requirements* on page A3-42, in any transaction:

- The **VALID** signal of the AXI interface sending information must not be dependent on the **READY** signal of the AXI interface receiving that information.
- An AXI interface that is receiving information can wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.

Note

It is acceptable to wait for **VALID** to be asserted before asserting **READY.** It is also acceptable to assert **READY** before detecting the corresponding **VALID**. This can result in a more efficient design.

In addition, there are dependencies between the handshake signals on different channels, and AXI4 defines an additional write response dependency. The following subsections define these dependencies:

- *Read transaction dependencies* on page A3-45
- *AXI3 write transaction dependencies* on page A3-45
- *AXI4 and AXI5 write transaction dependencies* on page A3-46

In the dependency diagrams:

- Single-headed arrows point to signals that can be asserted before or after the signal at the start of the arrow.
- Double-headed arrows point to signals that must be asserted only after assertion of the signal at the start of the arrow.

Read transaction dependencies

Figure A3-5 shows the read transaction handshake signal dependencies, and shows that, in a read transaction:

- The master must not wait for the slave to assert **ARREADY** before asserting **ARVALID**.
- The slave can wait for **ARVALID** to be asserted before it asserts **ARREADY**.
- The slave can assert **ARREADY** before **ARVALID** is asserted.
- The slave must wait for both **ARVALID** and **ARREADY** to be asserted before it asserts **RVALID** to indicate that valid data is available.
- The slave must not wait for the master to assert **RREADY** before asserting **RVALID**.
- The master can wait for **RVALID** to be asserted before it asserts **RREADY**.
- The master can assert **RREADY** before **RVALID** is asserted.

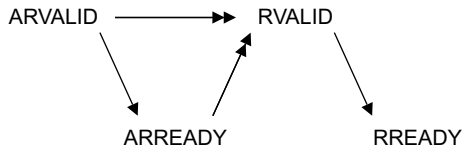


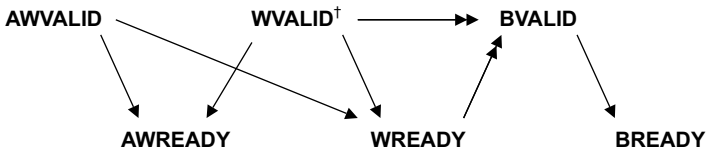
Figure A3-5 Read transaction handshake dependencies

AXI3 write transaction dependencies

Figure A3-6 shows the write transaction handshake signal dependencies, and shows that in a write transaction:

- The master must not wait for the slave to assert **AWREADY** or **WREADY** before asserting **AWVALID** or **WVALID**.
- The slave can wait for **AWVALID** or **WVALID**, or both before asserting **AWREADY**.
- The slave can assert **AWREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave can wait for **AWVALID** or **WVALID**, or both, before asserting **WREADY**.
- The slave can assert **WREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave must wait for both **WVALID** and **WREADY** to be asserted before asserting **BVALID**.

  The slave must also wait for **WLAST** to be asserted before asserting **BVALID**. Waiting is required because the write response, **BRESP**, must be signaled only after the last data transfer of a write transaction.
- The slave must not wait for the master to assert **BREADY** before asserting **BVALID**.
- The master can wait for **BVALID** before asserting **BREADY**.
- The master can assert **BREADY** before **BVALID** is asserted.



† Dependencies on the assertion of **WVALID** also require the assertion of **WLAST**

Figure A3-6 AXI3 write transaction handshake dependencies

Caution

The dependency rules must be observed to prevent a deadlock condition. For example, a master must not wait for **AWREADY** to be asserted before driving **WVALID**. A deadlock condition can occur if the slave is waiting for **WVALID** before asserting **AWREADY**.

AXI4 and AXI5 write transaction dependencies

AXI4 and AXI5 define an additional slave write response dependency. The slave must wait for **AWVALID**, **AWREADY**, **WVALID**, and **WREADY** to be asserted before asserting **BVALID**. By issuing a write response, the slave takes responsibility for hazard checking the write transaction against all subsequent transactions.

Note

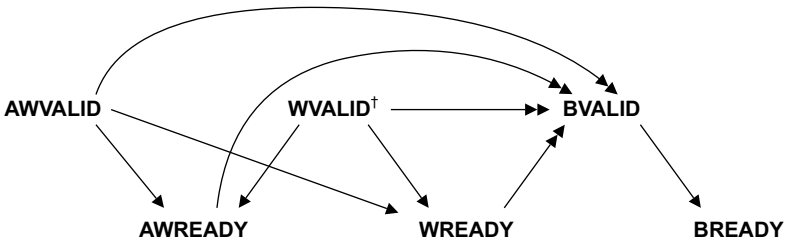
This additional dependency reflects the expected use in AXI3, because it is not expected that any components would accept all write data and provide a write response before the address is accepted.

Figure A3-7 shows all the AXI4 and AXI5 required slave write response handshake dependencies. The single-headed arrows point to signals that can be asserted before or after the previous signal is asserted. Double-headed arrows point to signals that must be asserted only after assertion of the previous signal.

These dependencies are:

- The master must not wait for the slave to assert **AWREADY** or **WREADY** before asserting **AWVALID** or **WVALID**.
- The slave can wait for **AWVALID** or **WVALID**, or both, before asserting **AWREADY**.
- The slave can assert **AWREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave can wait for **AWVALID** or **WVALID**, or both, before asserting **WREADY**.
- The slave can assert **WREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave must wait for **AWVALID**, **AWREADY**, **WVALID**, and **WREADY** to be asserted before asserting **BVALID**.

  The slave must also wait for **WLAST** to be asserted before asserting **BVALID**. This wait is because the write response, **BRESP**, must be signaled only after the last data transfer of a write transaction.
- The slave must not wait for the master to assert **BREADY** before asserting **BVALID.**
- The master can wait for **BVALID** before asserting **BREADY.**
- The master can assert **BREADY** before **BVALID** is asserted.



† Dependencies on the assertion of **WVALID** also require the assertion of **WLAST**

Figure A3-7 AXI4 and AXI5 write transaction handshake dependencies

A3.3.2 Legacy considerations

The additional dependency that is described in *AXI4 and AXI5 write transaction dependencies* on page A3-46 means that an AXI3 slave that accepts all write data and provides a write response before accepting the address is not compliant with AXI4 or AXI5. Converting an AXI3 legacy slave to AXI4 or AXI5 requires the addition of a wrapper. That wrapper ensures a returning write response is not provided until the appropriate address has been accepted by the slave.

Note

This specification strongly recommends that any new AXI3 slave implementation includes this additional dependency.

Any AXI3 master complies with the AXI4 and AXI5 write response requirements.

A3.4 Transaction structure

This section describes the structure of transactions. The following sections define the address, data, and response structures:

- *Address structure*
- *Pseudocode description of the transfers* on page A3-52
- *Data read and write structure* on page A3-54
- *Read and write response structure* on page A3-59

For the definitions of terms that are used in this section, see *Glossary* on page Glossary-493.

A3.4.1 Address structure

The AXI protocol is burst-based. The master begins each burst by driving control information and the address of the first byte in the transaction to the slave. As the burst progresses, the slave must calculate the addresses of subsequent transfers in the burst.

A burst must not cross a 4KB address boundary.

Note

This prohibition prevents a burst from crossing a boundary between two slaves. It also limits the number of address increments that a slave must support.

Burst length

The burst length is specified by:

- **ARLEN[7:0]**, for read transfers
- **AWLEN[7:0]**, for write transfers

In this specification, **AxLEN** indicates **ARLEN** or **AWLEN**.

AXI3 supports burst lengths of 1-16 transfers, for all burst types.

AXI4 extends burst length support for the INCR burst type to 1-256 transfers. Support for all other burst types in AXI4 remains at 1-16 transfers.

The burst length for AXI3 is defined as:

```
Burst_Length = AxLEN[3:0] + 1
```

To accommodate the extended burst length of the INCR burst type in AXI4, the burst length for AXI4 is defined as:

```
Burst_Length = AxLEN[7:0] + 1
```

AXI has the following rules governing the use of bursts:

- For wrapping bursts, the burst length must be 2, 4, 8, or 16.
- A burst must not cross a 4KB address boundary.
- Early termination of bursts is not supported.

No component can terminate a burst early. However, to reduce the number of data transfers in a write burst, the master can disable further writing by deasserting all the write strobes. In this case, the master must complete the remaining transfers in the burst. In a read burst, the master can discard read data, but it must complete all transfers in the burst.

Note

Discarding read data that is not required can result in lost data when accessing a read-sensitive device such as a FIFO. When accessing such a device, a master must use a burst length that exactly matches the size of the required data transfer.

Exclusive access restrictions on page A7-97 defines additional rules affecting bursts during an exclusive access.

In AXI4, transactions with INCR burst type and length greater than 16 can be converted to multiple smaller bursts, even if the transaction attributes indicate that the transaction is *Non-modifiable*. See *AXI4 changes to memory attribute signaling* on page A4-64. In this case, the generated bursts must retain the same transaction characteristics as the original transaction, the only exception is that:

- The burst length is reduced.
- The address of the generated bursts is adapted appropriately.

Note

The ability to break longer bursts into multiple shorter bursts is required for AXI3 compatibility. This ability might also be needed to reduce the impact of longer bursts on the QoS guarantees.

Burst size

The maximum number of bytes to transfer in each data transfer, or beat, in a burst, is specified by:

- **ARSIZE[2:0]**, for read transfers
- **AWSIZE[2:0]**, for write transfers

In this specification, **AxSIZE** indicates **ARSIZE** or **AWSIZE**.

Table A3-2 shows the **AxSIZE** encoding.

Table A3-2 Burst size encoding

AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

If the AXI bus is wider than the burst size, the AXI interface must determine from the transfer address which byte lanes of the data bus to use for each transfer. See *Data read and write structure* on page A3-54.

The size of any transfer must not exceed the data bus width of either agent in the transaction.

Burst type

The AXI protocol defines three burst types:

FIXED In a fixed burst:

- The address is the same for every transfer in the burst.
- The byte lanes that are valid are constant for all beats in the burst. However, within those byte lanes, the actual bytes that have **WSTRB** asserted can differ for each beat in the burst.

This burst type is used for repeated accesses to the same location such as when loading or emptying a FIFO.

INCR Incrementing. In an incrementing burst, the address for each transfer in the burst is an increment of the address for the previous transfer. The increment value depends on the size of the transfer. For example, for an aligned start address, the address for each transfer in a burst with a size of 4 bytes is the previous address plus four.

This burst type is used for accesses to normal sequential memory.

WRAP A wrapping burst is similar to an incrementing burst, except that the address wraps around to a lower address if an upper address limit is reached.

The following restrictions apply to wrapping bursts:

- The start address must be aligned to the size of each transfer.
- The length of the burst must be 2, 4, 8, or 16 transfers.

The behavior of a wrapping burst is:

- The lowest address that is used by the burst is aligned to the total size of the data to be transferred, that is, to ((size of each transfer in the burst) × (number of transfers in the burst)). This address is defined as the *wrap boundary*.
- After each transfer, the address increments in the same way as for an INCR burst. However, if this incremented address is ((wrap boundary) + (total size of data to be transferred)), then the address wraps round to the wrap boundary.
- The first transfer in the burst can use an address that is higher than the wrap boundary, subject to the restrictions that apply to wrapping bursts. The address wraps for any WRAP burst when the first address is higher than the wrap boundary.

This burst type is used for cache line accesses.

The burst type is specified by:

- **ARBURST[1:0]**, for read transfers
- **AWBURST[1:0]**, for write transfers

In this specification, **AxBURST** indicates **ARBURST** or **AWBURST**.

Table A3-3 shows the **AxBURST** signal encoding.

Table A3-3 Burst type encoding

AxBURST[1:0]	Burst type
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reserved

Burst address

This section provides methods for determining the address and byte lanes of transfers within a burst. The equations use the following variables:

Start_Address The start address that is issued by the master.

Number_Bytes The maximum number of bytes in each data transfer.

Data_Bus_Bytes The number of byte lanes in the data bus.

Aligned_Address The aligned version of the start address.

Burst_Length The total number of data transfers within a burst.

Address_N The address of transfer N in a burst. N is 1 for the first transfer in a burst.

Wrap_Boundary The lowest address within a wrapping burst.

Lower_Byte_Lane The byte lane of the lowest addressed byte of a transfer.

Upper_Byte_Lane The byte lane of the highest addressed byte of a transfer.

INT(x) The rounded-down integer value of x.

These equations determine addresses of transfers within a burst:

```
Start_Address = AxADDR
Number_Bytes = 2 ^ AxSIZE
Burst_Length = AxLEN + 1
Aligned_Address = (INT(Start_Address / Number_Bytes)) × Number_Bytes
```

This equation determines the address of the first transfer in a burst:

```
Address_1 = Start_Address
```

For an INCR burst, and for a WRAP burst for which the address has not wrapped, this equation determines the address of any transfer after the first transfer in a burst:

```
Address_N = Aligned_Address + (N – 1) × Number_Bytes
```

For a WRAP burst, the Wrap_Boundary variable defines the wrapping boundary:

```
Wrap_Boundary = (INT(Start_Address / (Number_Bytes × Burst_Length)))× (Number_Bytes × Burst_Length)
```

For a WRAP burst, if `Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)`, then:

- Use this equation for the current transfer:

  ```
  Address_N = Wrap_Boundary
  ```

- Use this equation for any subsequent transfers:

  ```
  Address_N = Start_Address + ((N – 1) × Number_Bytes) – (Number_Bytes × Burst_Length)
  ```

These equations determine the byte lanes to use for the first transfer in a burst:

```
Lower_Byte_Lane = Start_Address – (INT(Start_Address / Data_Bus_Bytes)) × Data_Bus_Bytes

Upper_Byte_Lane = Aligned_Address + (Number_Bytes – 1) –
                  (INT(Start_Address / Data_Bus_Bytes)) × Data_Bus_Bytes
```

These equations determine the byte lanes to use for all transfers after the first transfer in a burst:

```
Lower_Byte_Lane = Address_N – (INT(Address_N / Data_Bus_Bytes)) × Data_Bus_Bytes
Upper_Byte_Lane = Lower_Byte_Lane + Number_Bytes – 1
```

Data is transferred on:

```
DATA((8 × Upper_Byte_Lane) + 7: (8 × Lower_Byte_Lane))
```

The transaction container describes all the bytes that could be accessed in that transaction, if the address is aligned and strobes are asserted:

```
Container_Size = Number_Bytes x Burst_Length
```

For INCR bursts:

```
Container_Lower = Aligned_Address
Container_Upper = Aligned_Address + Container_Size
```

For WRAP bursts:

```
Container_Lower = Wrap_Boundary
Container_Upper = Wrap_Boundary + Container_Size
```

A3.4.2 Pseudocode description of the transfers

```
// DataTransfer()
// ==============

DataTransfer(Start_Address, Number_Bytes, Burst_Length, Data_Bus_Bytes, Mode, IsWrite)
// Data_Bus_Bytes is the number of 8-bit byte lanes in the bus
// Mode is the AXI transfer mode
// IsWrite is TRUE for a write, and FALSE for a read

    assert Mode IN {FIXED, WRAP, INCR};
    addr = Start_Address;                          // Variable for current address
    Aligned_Address = (INT(addr/Number_Bytes) * Number_Bytes);
    aligned = (Aligned_Address == addr);           // Check whether addr is aligned to nbytes
    dtsize = Number_Bytes * Burst_Length;          // Maximum total data transaction size

    if mode == WRAP then
        Lower_Wrap_Boundary = (INT(addr/dtsize) * dtsize);
        // addr must be aligned for a wrapping burst
        Upper_Wrap_Boundary = Lower_Wrap_Boundary + dtsize;

    for n = 1 to Burst_Length
        Lower_Byte_Lane = addr - (INT(addr/Data_Bus_Bytes)) * Data_Bus_Bytes;
        if aligned then
            Upper_Byte_Lane = Lower_Byte_Lane + Number_Bytes - 1
        else
            Upper_Byte_Lane = Aligned_Address + Number_Bytes - 1
                              - (INT(addr/Data_Bus_Bytes)) * Data_Bus_Bytes;

        // Peform data transfer
        if IsWrite then
            dwrite(addr, low_byte, high_byte)
        else
            dread(addr, low_byte, high_byte);

        // Increment address if necessary
        if mode != FIXED then
            if aligned then
```

```
            addr = addr + Number_Bytes;
            if mode == WRAP then
                // WRAP mode is always aligned
                if addr >= Upper_Wrap_Boundary then addr = Lower_Wrap_Boundary;
        else
            addr = Aligned_Address + Number_Bytes;
            aligned = TRUE;            // All transfers after the first are aligned
return;
```

A3.4.3 Regular transactions

There are many options of burst, size, and length for a transaction. However, some interfaces and transaction types might only use a subset of these options. If a slave component is attached to a master which uses only a subset of transaction options, it can be designed with simplified decode logic.

The Regular attribute is defined, to identify transactions which meet the following criteria:

- **AxLEN** is 1, 2, 4, 8, or 16.
- **AxSIZE** is the same as the data bus width, if **AxLEN** is greater than 1.
- **AxBURST** is INCR or WRAP, not FIXED.
- **AxADDR** is aligned to the transaction container for INCR transactions.
- **AxADDR** is aligned to **AxSIZE** for WRAP transactions.

Regular transactions property

The Regular_Transactions_Only property is used to define whether a master issues only Regular type transactions and if a slave only supports Regular transactions:

TRUE Only Regular transactions are supported.

FALSE All legal combinations of **AxBURST**, **AxSIZE**, and **AxLEN** are supported.

If Regular_Transactions_Only is not declared, it is considered to be False.

The Regular_Transactions_Only property can be True for the following interfaces:

- AXI5
- ACE5
- ACE5-Lite
- ACE5-LiteDVM

Interoperability

Table A3-4 gives guidance applies for connecting master and slave components with different property values:

Table A3-4 Regular_Transactions_Only Interoperability

	Slave: False	**Slave: True**
Master: False	Compatible.	Not compatible. If the master issues a transaction that is not Regular, then data corruption or deadlock might occur.
Master: True	Compatible.	Compatible.

A3.4.4 Data read and write structure

This section describes the transfers of varying sizes on the AXI read and write data buses and how the interface performs mixed-endian and unaligned transfers. It contains the following sections:

- *Write strobes*
- *Narrow transfers*
- *Byte invariance* on page A3-55
- *Unaligned transfers* on page A3-56

Write strobes

The **WSTRB[n:0]** signals when HIGH, specify the byte lanes of the data bus that contain valid information. There is one write strobe for each 8 bits of the write data bus, therefore **WSTRB[n]** corresponds to **WDATA[(8n)+7: (8n)]**.

A master must ensure that the write strobes are HIGH only for byte lanes that contain valid data.

When **WVALID** is LOW, the write strobes can take any value, although this specification recommends that they are either driven LOW or held at their previous value.

Narrow transfers

When a master generates a transfer that is narrower than its data bus, the address and control information determine the byte lanes that the transfer uses:

- In incrementing or wrapping bursts, different byte lanes are used on each beat of the burst.
- In a fixed burst, the same byte lanes are used on each beat.

Figure A3-8 and Figure A3-9 on page A3-55 give two examples of byte lanes use. The shaded cells indicate bytes that are not transferred.

In Figure A3-8:

- The burst has five transfers.
- The starting address is 0.
- Each transfer is 8 bits.
- The transfers are on a 32-bit bus.
- The burst type is INCR.

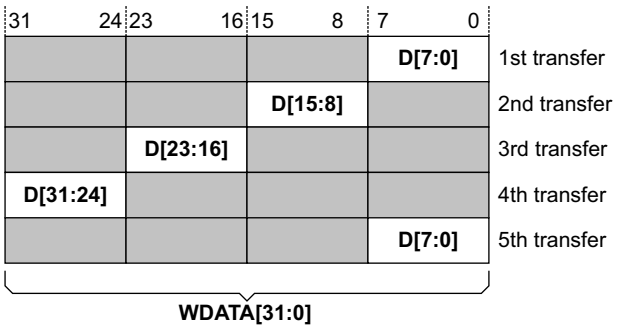


Figure A3-8 Narrow transfer example with 8-bit transfers

In Figure A3-9:

- The burst has three transfers.
- The starting address is 4.
- Each transfer is 32 bits.
- The transfers are on a 64-bit bus.

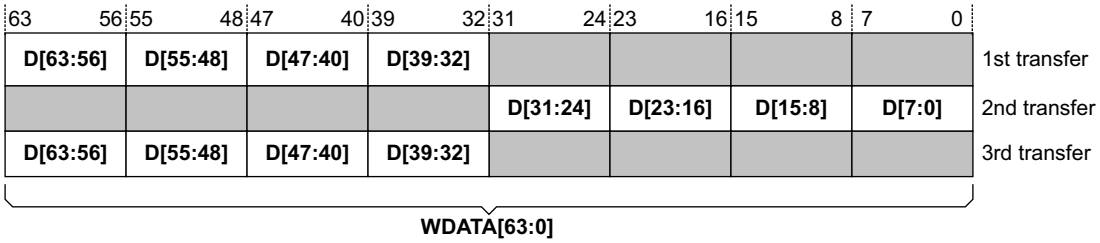


Figure A3-9 Narrow transfer example with 32-bit transfers

Byte invariance

To access mixed-endian data structures in a single memory space, the AXI protocol uses a *byte-invariant endianness* scheme.

Byte-invariant endianness means that, for any multi-byte element in a data structure:

- The element uses the same continuous bytes of memory, regardless of the endianness of the data.
- The endianness determines the order of those bytes in memory, meaning it determines whether the first byte in memory is the *most significant byte* (MSB) or the *least significant byte* (LSB) of the element.
- Any byte transfer to an address passes the 8 bits of data on the same data bus wires, to the same address location, regardless of the endianness of any larger data element that it is a constituent of.

Components that have only one transfer width must have their byte lanes that are connected to the appropriate byte lanes of the data bus. Components that support multiple transfer widths might require a more complex interface to convert an interface that is not naturally byte-invariant.

Most little-endian components can connect directly to a byte-invariant interface. Components that support only big-endian transfers require a conversion function for byte-invariant operation.

The examples in Figure A3-10 and on page A3-56 show a 32-bit number `0x0A0B0C0D`, stored in a register and in a memory.

Figure A3-10 shows an example of the big-endian, byte-invariant, data structure. In this structure:

- The *most significant byte* (MSB) of the data, which is `0x0A`, is stored in the MSB position in the register.
- The MSB of the data is stored in the memory location with the lowest address.
- The other data bytes are positioned in decreasing order of significance.

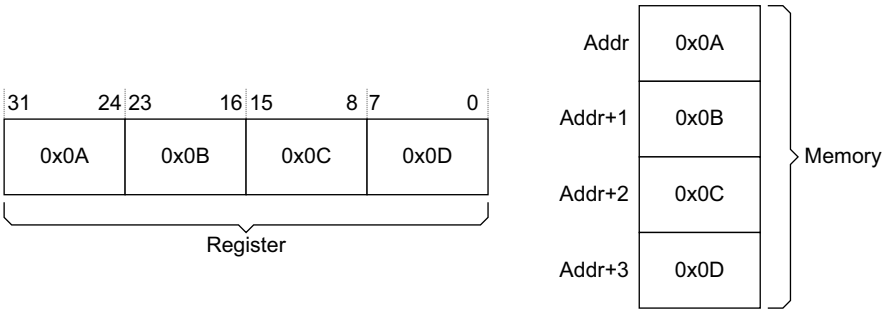


Figure A3-10 Example big-endian byte-invariant data structure

Figure A3-11 shows an example of the little-endian, byte-invariant, data structure. In this structure:

- The *least significant byte* (LSB) of the data, which is `0x0D`, is stored in the LSB position in the register.
- The LSB of the data is stored in the memory location with the lowest address.
- The other data bytes are positioned in increasing order of significance.

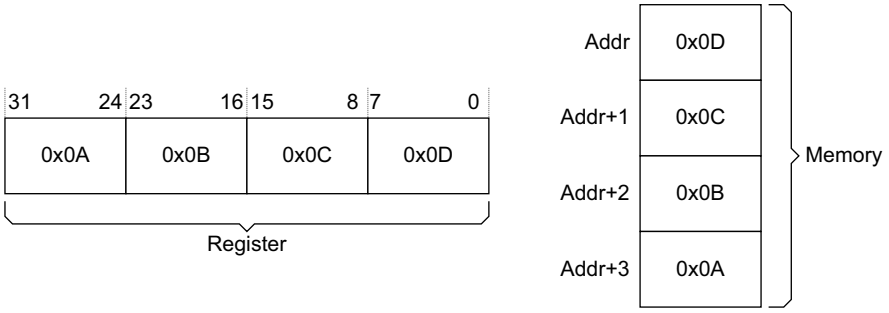


Figure A3-11 Example little-endian byte-invariant data structure

The examples in Figure A3-10 on page A3-55 and Figure A3-11 show that byte invariance ensures that big-endian and little-endian structures can coexist in a single memory space without corruption. Figure A3-12 shows an example of a data structure that requires byte-invariant access. In this example, the header fields use little-endian ordering, and the payload uses big-endian ordering.

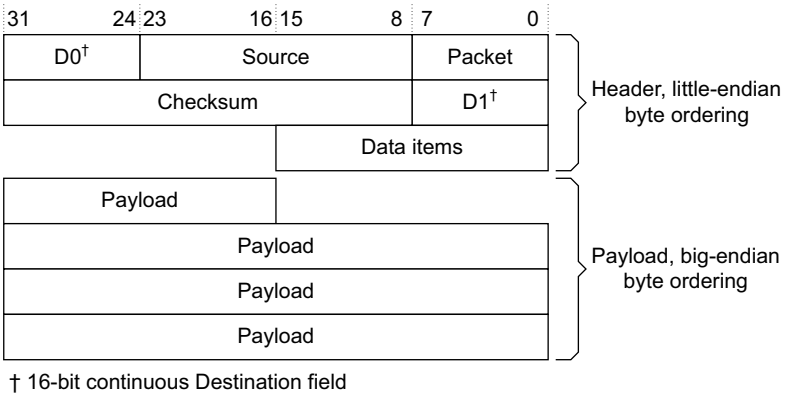


Figure A3-12 Example mixed-endian data structure

In this structure, for example, Data items is a two-byte little-endian element, meaning its lowest address is its LSB. The use of byte invariance ensures that a big-endian access to the payload does not corrupt the little-endian element.

Unaligned transfers

AXI supports unaligned transfers. For any burst that is made up of data transfers wider than 1 byte, the first bytes accessed might be unaligned with the natural address boundary. For example, a 32-bit data packet that starts at a byte address of `0x1002` is not aligned to the natural 32-bit address boundary.

A master can:

- Use the low-order address lines to signal an unaligned start address.
- Provide an aligned address and use the byte lane strobes to signal the unaligned start address.

Note

The information on the low-order address lines must be consistent with the information on the byte lane strobes.

The slave is not required to take special action based on any alignment information from the master.

Figure A3-13 shows examples of incrementing bursts, with aligned and unaligned 32-bit transfers, on a 32-bit bus. Each row in the figure represents a transfer and the shaded cells indicate bytes that are not transferred.

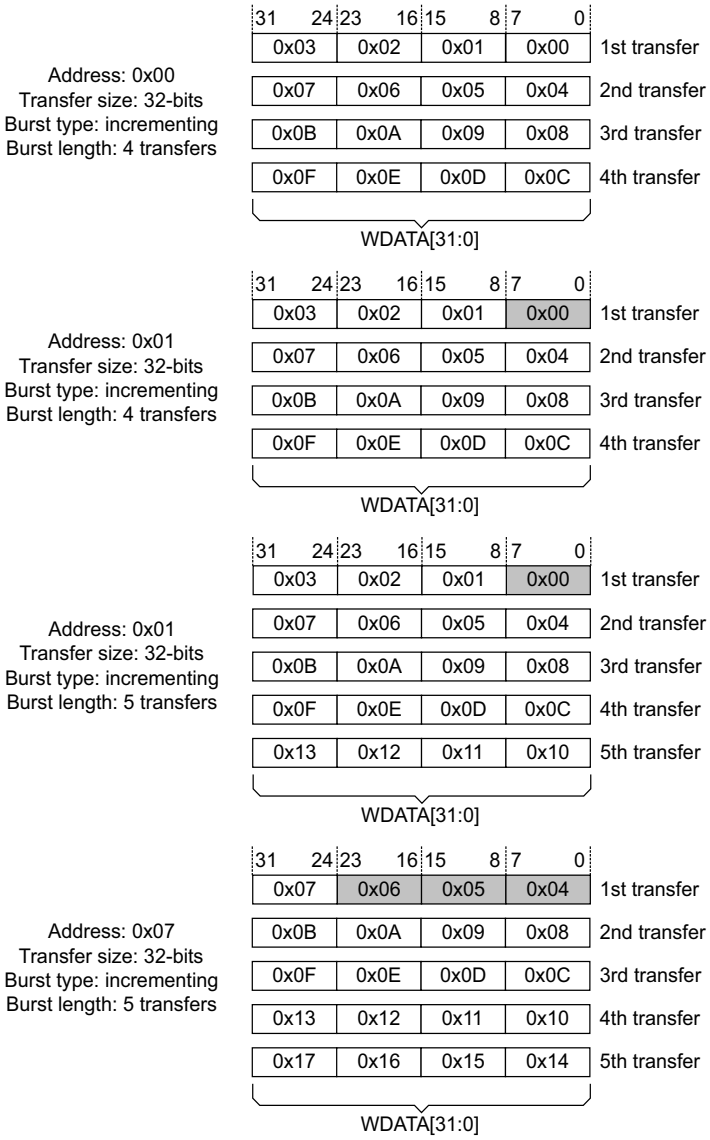


Figure A3-13 Aligned and unaligned transfers on a 32-bit bus

Figure A3-14 shows examples of incrementing bursts, with aligned and unaligned 32-bit transfers, on a 64-bit bus. Each row in the figure represents a transfer and the shaded cells indicate bytes that are not transferred.

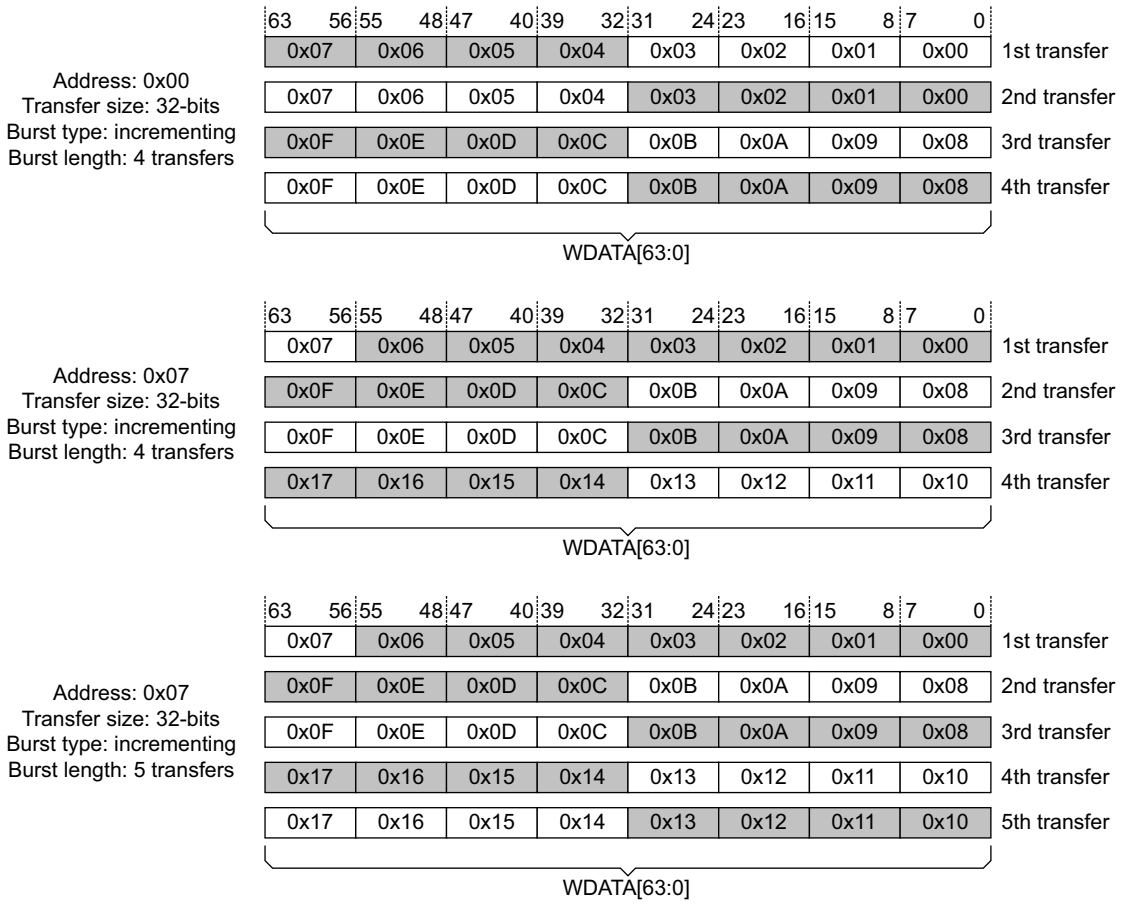


Figure A3-14 Aligned and unaligned transfers on a 64-bit bus

Figure A3-15 shows an example of a wrapping burst, with aligned 32-bit transfers, on a 64-bit bus. Each row in the figure represents a transfer and the shaded cells indicate bytes that are not transferred.

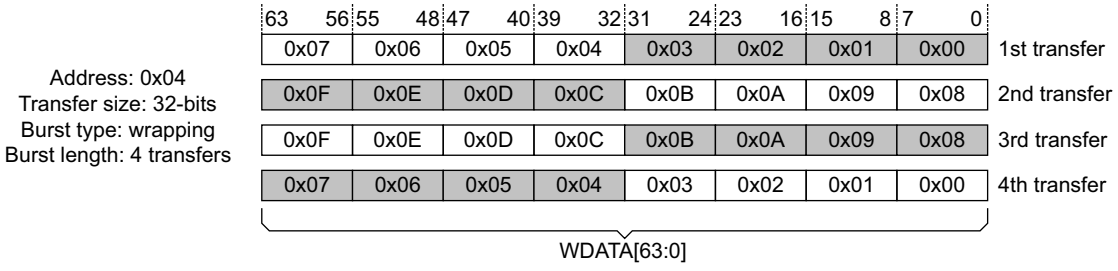


Figure A3-15 Aligned wrapping transfers on a 64-bit bus

A3.4.5 Read and write response structure

The AXI protocol provides response signaling for both read and write transactions:

- For read transactions, the response information from the slave is signaled on the read data channel.
- For write transactions the response information is signaled on the write response channel.

The responses are signaled by:

- **RRESP[1:0]**, for read transfers.
- **BRESP[1:0]**, for write transfers.

The responses are:

OKAY Normal access success. Indicates that a normal access has been successful. Can also indicate that an exclusive access has failed. See *OKAY, normal access success*.

EXOKAY Exclusive access okay. Indicates that either the read or write portion of an exclusive access has been successful. See *EXOKAY, exclusive access success* on page A3-60.

SLVERR Slave error. Used when the access has reached the slave successfully, but the slave wishes to return an error condition to the originating master. See *SLVERR, slave error* on page A3-60.

DECERR Decode error. Generated, typically by an interconnect component, to indicate that there is no slave at the transaction address. See *DECERR, decode error* on page A3-60.

Table A3-5 shows the encoding of the **RRESP** and **BRESP** signals.

Table A3-5 RRESP and BRESP encoding

RRESP[1:0] BRESP[1:0]	Response
0b00	OKAY
0b01	EXOKAY
0b10	SLVERR
0b11	DECERR

For a write transaction, a single response is signaled for the entire burst, and not for each data transfer within the burst.

In a read transaction, the slave can signal different responses for different transfers in a burst. For example, in a burst of 16 read transfers the slave might return an OKAY response for 15 of the transfers and a SLVERR response for one of the transfers.

The protocol specifies that the required number of data transfers must be performed, even if an error is reported. For example, if a read of eight transfers is requested from a slave but the slave has an error condition, the slave must perform eight data transfers, each with an error response. The remainder of the burst is not canceled if the slave gives a single error response.

OKAY, normal access success

An OKAY response indicates any one of the following:

- The success of a normal access.
- The failure of an exclusive access.
- An exclusive access to a slave that does not support exclusive access.

OKAY is the response for most transactions.

EXOKAY, exclusive access success

An EXOKAY response indicates the success of an exclusive access. This response can only be given as the response to an exclusive read or write. See *Exclusive accesses* on page A7-96.

SLVERR, slave error

The SLVERR response indicates an unsuccessful transaction.

To simplify system monitoring and debugging, this specification recommends that error responses are used only for error conditions and not for signaling normal, expected events. Examples of slave error conditions are:

- FIFO or buffer overrun or underrun condition
- Unsupported transfer size attempted.
- Write access attempted to read-only location
- Timeout condition in the slave
- Access attempted to a disabled or powered-down function

DECERR, decode error

The DECERR response indicates that the interconnect cannot successfully decode a slave access.

If the interconnect cannot successfully decode a slave access, it must return the DECERR response. This specification recommends that the interconnect routes the access to a default slave, and the default slave returns the DECERR response.

The AXI protocol requires that all data transfers for a transaction are completed, even if an error condition occurs. Any component giving a DECERR response must meet this requirement.

Chapter A4
Transaction Attributes

This chapter describes the attributes that determine how a transaction should be treated by system components such as caches, buffers, and memory controllers. It contains the following sections:

A4.1 Transaction types and attributes

Slaves are classified as either:

Memory Slave

A memory slave is required to handle all transaction types correctly.

Peripheral Slave

A peripheral slave has an IMPLEMENTATION DEFINED method of access. Typically, the method of access is defined in the component data sheet, that describes the transaction types that the slave handles correctly.

Any access to the peripheral slave that is not part of the IMPLEMENTATION DEFINED method of access must complete, in compliance with the protocol. However, when such an access has been made, there is no requirement that the peripheral slave continues to operate correctly. It is only required to continue to complete further transactions in a protocol-compliant manner.

Note

- Compliant completion of all transaction types is required to prevent system deadlock, however, continued correct operation of the peripheral slave is not required.
- Because a peripheral slave is required to work correctly only for a defined method of access, it can have a reduced set of interface signals.

The AXI protocol defines a set of transaction attributes that support memory and peripheral slaves. The **ARCACHE** and **AWCACHE** signals specify the transaction attributes. They control:

- How a transaction progresses through the system.
- How any system-level caches handle the transaction.

In this specification, the term **AxCACHE** refers collectively to the **ARCACHE** and **AWCACHE** signals.

The following sections describe the transaction attributes:

- *AXI3 memory attribute signaling* on page A4-63
- *AXI4 changes to memory attribute signaling* on page A4-64

A4.2 AXI3 memory attribute signaling

In AXI3, the **AxCACHE[3:0]** signals specify the *Bufferable*, *Cacheable*, and *Allocate* attributes of the transaction.

Table A4-1 shows the **AxCACHE** encoding.

Table A4-1 Transaction attribute encoding

AxCACHE	Value	Transaction attribute
[0]	0	Non-bufferable
	1	Bufferable
[1]	0	Non-cacheable
	1	Cacheable
[2]	0	No Read-Allocate
	1	Read-Allocate
[3]	0	No Write-Allocate
	1	Write-Allocate

AxCACHE[0], Bufferable (B) bit

When this bit is asserted, the interconnect, or any component, can delay the transaction reaching its final destination for any number of cycles.

Note

Normally, the Bufferable attribute is only relevant to writes.

AxCACHE[1], Cacheable (C) bit

When this bit is deasserted, allocation of the transaction is forbidden.

When this bit is asserted:

- Allocation of the transaction is permitted. RA and WA give additional hint information.
- The characteristics of a transaction at the final destination does not have to match the characteristics of the original transaction.

  For writes this means that several different writes can be merged together.

  For reads this means that the contents of a location can be prefetched, or the values from a single fetch can be used for multiple read transactions.

AxCACHE[2], Read-Allocate (RA) bit

When this bit is asserted, read allocation of the transaction is recommended but is not mandatory.

The RA bit must not be asserted if the C bit is deasserted.

AxCACHE[3], Write-Allocate (WA) bit

When this bit is asserted, write allocation of the transaction is recommended but is not mandatory.

The WA bit must not be asserted if the C bit is deasserted.

A4.3 AXI4 changes to memory attribute signaling

AXI4 makes the following changes to the AXI3 memory attribute signaling:

- The **AxCACHE[1]** bits are renamed as the *Modifiable* bits.
- Ordering requirements are defined for *Non-modifiable* transactions.
- The meanings of *Read-Allocate* and *Write-Allocate* are updated.

A4.3.1 AxCACHE[1], Modifiable

In AXI4, the **AxCACHE[1]** bit is the *Modifiable* bit. When HIGH, Modifiable indicates that the characteristics of the transaction can be modified. When Modifiable is LOW, the transaction is *Non-modifiable*.

Note

The **AxCACHE[1]** bit is renamed from the *Cacheable* bit to the *Modifiable* bit to better describe the required functionality. The actual functionality is unchanged.

The following sections describe the properties of Non-modifiable and Modifiable transactions.

Non-modifiable transactions

A Non-modifiable transaction is indicated by setting **AxCACHE[1]** LOW.

A Non-modifiable transaction must not be split into multiple transactions or merged with other transactions.

In a Non-modifiable transaction, the parameters that are shown in Table A4-2 must not be changed.

Table A4-2 Parameters fixed as Non-modifiable

Parameter	Signals
Transfer address	**AxADDR**, and therefore **AxREGION**
Burst size	**AxSIZE**
Burst length	**AxLEN**
Burst type	**AxBURST**
Lock type	**AxLOCK**
Protection type	**AxPROT**

The **AxCACHE** attribute can only be modified to convert a transaction from being Bufferable to Non-bufferable. No other change to **AxCACHE** is permitted.

The transaction ID and the QoS values can be modified.

A Non-modifiable transaction with burst length greater than 16 can be split into multiple transactions. Each resulting transaction must meet the requirements that are given in this subsection, except that:

- The burst length is reduced.
- The address of the generated bursts is adapted appropriately.

A Non-modifiable transaction that is an Exclusive access, as indicated by **AxLOCK** asserted, is permitted to have the transaction size, **AxSIZE**, and the transaction length, **AxLEN**, modified if the total number of bytes accessed remains the same.

Note

There are circumstances where it is not possible to meet the requirements of Non-modifiable transactions. For example, when downsizing to a bus width narrower than that required by the transaction size, **AxSIZE**, the transaction must be modified.

A component that performs such an operation can optionally include an IMPLEMENTATION DEFINED mechanism to indicate that a modification has occurred. This mechanism can assist with software debug.

Modifiable transactions

A Modifiable transaction is indicated by asserting **AxCACHE[1]**.

A Modifiable transaction can be modified in the following ways:

- A transaction can be broken into multiple transactions.
- Multiple transactions can be merged into a single transaction.
- A read transaction can fetch more data than required.
- A write transaction can access a larger address range than required, using the **WSTRB** signals to ensure that only the appropriate locations are updated.
- In each generated transaction, the following signals can be modified:
  - The transfer address, **AxADDR**
  - The burst size, **AxSIZE**
  - The burst length, **AxLEN**
  - The burst type, **AxBURST**

The following must not be changed:

- The lock type, **AxLOCK**
- The protection type, **AxPROT**

The memory attribute, **AxCACHE**, can be modified, but any modification must ensure that the visibility of transactions by other components is not reduced, either by preventing propagation of transactions to the required point, or by changing the need to look up a transaction in a cache. Any modification to the memory attributes must be consistent for all transactions to the same address range.

The transaction ID and QoS values can be modified.

No transaction modification is permitted that:

- Causes accesses to a different 4KByte address space than that of the original transaction.
- Causes a single access to a single-copy atomicity sized region to be performed as multiple accesses. See *Single-copy atomicity size* on page A7-94.

A4.3.2 Updated meaning of Read-Allocate and Write-Allocate

In AXI4, the meaning of the *Read-Allocate* and *Write-Allocate* bits is updated so that one bit indicates that an allocation that occurred for the transaction and the other bit indicates that an allocation could have been made due to another transaction.

For read transactions, the Write-Allocate bit is redefined to indicate that:

- The location could have been previously allocated in the cache because of a write transaction (as the AXI3 definition).
- The location could have been previously allocated in the cache because of the actions of another master (additional AXI4 definition).

For write transactions, the Read-Allocate bit is redefined to indicate that:

- The location could have been previously allocated in the cache because of a read transaction (as the AXI3 definition).
- The location could have been previously allocated in the cache because of the actions of another master (additional AXI4 definition).

These changes mean:

- A transaction must be looked up in a cache if the value of **AxCACHE[3:2]** is not `0b00`.
- A transaction does not need to be looked up in a cache if the value of **AxCACHE[3:2]** is `0b00`.

Note

The change to the definition of **AxCACHE** means that these signals can differ for a read and write transaction to the same location.

Table A4-3 shows the AXI4 bit allocations for the **AWCACHE** signals.

Table A4-3 AWCACHE bit allocations

Signal	AXI4 definition	Description
AWCACHE[3]	Allocate	When asserted, the transaction must be looked up in a cache because it could have been previously allocated. The transaction must also be looked up in a cache if **AWCACHE[2]** is asserted. When deasserted, if **AWCACHE[2]** is also deasserted, then the transaction does not need to be looked up in a cache and the transaction must propagate to the final destination. When asserted, this specification recommends that this transaction is allocated in the cache for performance reasons.
AWCACHE[2]	Other Allocate	When asserted, the transaction must be looked up in a cache because it could have been previously allocated in the cache by another transaction, either a read transaction or a transaction from another master. The transaction must also be looked up in a cache if **AWCACHE[3]** is asserted. When deasserted, if **AWCACHE[3]** is also deasserted, then the transaction does not need to be looked up in a cache and the transaction must propagate to the final destination.
AWCACHE[1]	Modifiable	When asserted, the characteristics of the transaction can be modified and writes can be merged. When deasserted, the characteristics of the transaction must not be modified.
AWCACHE[0]	Bufferable	When deasserted, if both of **AWCACHE[3:2]** are deasserted, the write response must be given from the final destination. When asserted, if both of **AWCACHE[3:2]** are deasserted, the write response can be given from an intermediate point, but the write transaction is required to be made visible at the final destination *in a timely manner*. When deasserted, if either of **AWCACHE[3:2]** is asserted, the write response can be given from an intermediate point, but the write transaction is required to be made visible at the final destination *in a timely manner*. When asserted, if either of **AWCACHE[3:2]** is asserted, the write response can be given from an intermediate point. The write transaction is not required to be made visible at the final destination.

Table A4-4 shows the AXI4 bit allocations for the **ARCACHE** signals.

Table A4-4 ARCACHE bit allocations

Signal	AXI4 definition	Description
ARCACHE[3]	Other Allocate	When asserted, the transaction must be looked up in a cache because it could have been allocated in the cache by another transaction, either a write transaction or a transaction from another master. The transaction must also be looked up in a cache if **ARCACHE[2]** is asserted. When deasserted, if **ARCACHE[2]** is also deasserted, then the transaction does not need to be looked up in a cache.
ARCACHE[2]	Allocate	When asserted, the transaction must be looked up in a cache because it could have been allocated. The transaction must also be looked up in a cache if **ARCACHE[3]** is asserted. When deasserted, if **ARCACHE[3]** is also deasserted, then the transaction does not need to be looked up in a cache. When asserted, this specifacation recommends that this transaction is allocated in the cache for performance reasons.
ARCACHE[1]	Modifiable	When asserted, the characteristics of the transaction can be modified and a larger quantity of read data can be fetched than is required. When deasserted the characteristics of the transaction must not be modified.
ARCACHE[0]	Bufferable	This bit has no effect when **ARCACHE[3:1]** are deasserted. When **ARCACHE[3:2]** are deasserted and **ARCACHE[1]** is asserted: • If this bit is deasserted, the read data must be obtained from the final destination. • If this bit is asserted, the read data can be obtained from the final destination or from a write that is progressing to the final destination. When either **ARCACHE[3]** is asserted, or **ARCACHE[2]** is asserted, this bit can be used to distinguish between Write-Through and Write-Back memory types.

A4.4 Memory types

The AXI4 protocol introduces new names for the *memory types* that are identified by the **AxCACHE** encoding. Table A4-5 shows the AXI4 **AxCACHE** encoding and associated memory types. Some memory types have different encodings in AXI3 and these encodings are shown in brackets.

Note

The same memory type can have different encodings on the read channel and write channel. These encodings provide backwards compatibility with AXI3 **AxCACHE** definitions.

In AXI4, it is legal to use more than one **AxCACHE** value for a particular memory type. Table A4-5 shows the preferred AXI4 value with the legal AXI3 value in brackets.

Table A4-5 Memory type encoding

ARCACHE[3:0]	AWCACHE[3:0]	Memory type
0b0000	0b0000	Device Non-bufferable
0b0001	0b0001	Device Bufferable
0b0010	0b0010	Normal Non-cacheable Non-bufferable
0b0011	0b0011	Normal Non-cacheable Bufferable
0b1010	0b0110	Write-Through No-Allocate
0b1110 (0b0110)	0b0110	Write-Through Read-Allocate
0b1010	0b1110 (0b1010)	Write-Through Write-Allocate
0b1110	0b1110	Write-Through Read and Write-Allocate
0b1011	0b0111	Write-Back No-Allocate
0b1111 (0b0111)	0b0111	Write-Back Read-Allocate
0b1011	0b1111 (0b1011)	Write-Back Write-Allocate
0b1111	0b1111	Write-Back Read and Write-Allocate

All values that are not shown in Table A4-5 are reserved.

A4.4.1 Memory type requirements

This section specifies the required behavior for each of the memory types.

Device Non-bufferable

The required behavior for Device Non-bufferable memory is:

- The write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- Transactions are Non-modifiable, see *Non-modifiable transactions* on page A4-64.
- Reads must not be prefetched. Writes must not be merged.

Device Bufferable

The required behavior for the Device Bufferable memory type is:

- The write response can be obtained from an intermediate point.
- Write transactions must be made visible at the final destination *in a timely manner*, as defined in *Transaction buffering* on page A4-74.
- Read data must be obtained from the final destination.
- Transactions are Non-modifiable, see *Non-modifiable transactions* on page A4-64.
- Reads must not be prefetched. Writes must not be merged.

Note

Both Device memory types are Non-modifiable. In this protocol specification, the terms Device memory and Non-modifiable memory are interchangeable.

For read transactions, there is no difference in the required behavior for Device Non-bufferable and Device Bufferable memory types.

Normal Non-cacheable Non-bufferable

The required behavior for the Normal Non-cacheable Non-bufferable memory type is:

- The write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- Transactions are Modifiable, see *Modifiable transactions* on page A4-65
- Writes can be merged.

Normal Non-cacheable Bufferable

The required behavior for the Normal Non-cacheable Bufferable memory type is:

- The write response can be obtained from an intermediate point.
- Write transactions must be made visible at the final destination *in a timely manner*, as defined in *Transaction buffering* on page A4-74. There is no mechanism to determine when a write transaction is visible at its final destination.
- Read data must be obtained from either:
  - The final destination
  - A write transaction that is progressing to its final destination

  If read data is obtained from a write transaction:
  - It must be obtained from the most recent version of the write.
  - The data must not be cached to service a later read.
- Transactions are Modifiable, see *Modifiable transactions* on page A4-65.
- Writes can be merged.

Note

For a Normal Non-cacheable Bufferable read, data can be obtained from a write transaction that is still progressing to its final destination. This data is indistinguishable from the read and write transactions propagating to arrive at the final destination at the same time. Read data that is returned in this manner does not indicate that the write transaction is visible at the final destination.

Write-Through No-Allocate

The required behavior for the Write-Through No-Allocate memory type is:

- The write response can be obtained from an intermediate point.
- Write transactions must be made visible at the final destination *in a timely manner*, as defined in *Transaction buffering* on page A4-74. There is no mechanism to determine when a write transaction is visible at the final destination.
- Read data can be obtained from an intermediate cached copy.
- Transactions are Modifiable, see *Modifiable transactions* on page A4-65.
- Reads can be prefetched.
- Writes can be merged.
- A cache lookup is required for read and write transactions.
- The No-Allocate attribute is an allocation hint, that is, it is a recommendation to the memory system that, for performance reasons, these transactions are not allocated. However, the allocation of read and write transactions is not prohibited.

Write-Through Read-Allocate

The required behavior for the Write-Through Read-Allocate memory type is the same as for Write-Through No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is recommended.
- Allocation of write transactions is not recommended.

Write-Through Write-Allocate

The required behavior for the Write-Through Write-Allocate memory type is the same as for Write-Through No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is not recommended.
- Allocation of write transactions is recommended.

Write-Through Read and Write-Allocate

The required behavior for the Write-Through Read and Write-Allocate memory type is the same as for Write-Through No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is recommended.
- Allocation of write transactions is recommended.

Write-Back No-Allocate

The required behavior for the Write-Back No-Allocate memory type is:

- The write response can be obtained from an intermediate point.
- Write transactions are not required to be made visible at the final destination.
- Read data can be obtained from an intermediate cached copy.
- Transactions are Modifiable, see *Modifiable transactions* on page A4-65.
- Reads can be prefetched.
- Writes can be merged.
- A cache lookup is required for read and write transactions.
- The No-Allocate attribute is an allocation hint, that is, it is a recommendation to the memory system that, for performance reasons, these transactions are not allocated. However, the allocation of read and write transactions is not prohibited.

Write-Back Read-Allocate

The required behavior for the Write-Back Read-Allocate memory type is the same as for Write-Back No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is recommended.
- Allocation of write transactions is not recommended.

Write-Back Write-Allocate

The required behavior for the Write-Back Write-Allocate memory type is the same as for Write-Back No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is not recommended.
- Allocation of write transactions is recommended.

Write-Back Read and Write-Allocate

The required behavior for the Write-Back Read and Write-Allocate memory type is the same as for Write-Back No-Allocate memory. But in this case the allocation hint is that, for performance reasons:

- Allocation of read transactions is recommended.
- Allocation of write transactions is recommended.

A4.5 Mismatched memory attributes

Multiple agents that are accessing the same area of memory, can use mismatched memory attributes. However, for functional correctness, the following rules must be obeyed:

- All masters accessing the same area of memory must have a consistent view of the cacheability of that area of memory at any level of hierarchy. The rules to be applied are:

  Address region not Cacheable

  All masters must use transactions with both **AxCACHE[3:2]** deasserted.

  Address region Cacheable

  All masters must use transactions with either of **AxCACHE[3:2]** asserted.

- Different masters can use different allocation hints.
- If an addressed region is Normal Non-cacheable, any master can access it using a Device memory transaction.
- If an addressed region has the Bufferable attribute, any master can access it using transactions that do not permit bufferable behavior.

  Note

  For example, a transaction that requires the response from the final destination does not permit bufferable behavior.

A4.5.1 Changing memory attributes

The attributes for a particular memory region can be changed from one type to another incompatible type. For example, the attribute can be changed from Write-Through Cacheable to Normal Non-cacheable. This change requires a suitable process to perform the change. Typically, the following process is performed:

1. All masters stop accessing the region.
2. A single master performs any required cache maintenance operations.
3. All masters restart accessing the memory region, using the new attributes.

A4.6 Transaction buffering

Write access to the following memory types do not require a transaction response from the final destination, but do require that write transactions are made visible at the final destination *in a timely manner*:

- Device Bufferable
- Normal Non-cacheable Bufferable
- Write-Through

For write transactions, all three memory types require the same behavior. For read transactions, the required behavior is as follows:

- For Device Bufferable memory, read data must be obtained from the final destination.
- For Normal Non-cacheable Bufferable memory, read data must be obtained either from the final destination or from a write transaction that is progressing to its final destination.
- For Write-Through memory, read data can be obtained from an intermediate cached copy.

In addition to ensuring that write transactions progress towards their final destination *in a timely manner*, intermediate buffers must behave as follows:

- An intermediate buffer that can respond to a transaction must ensure that, over time, any read transaction to Normal Non-cacheable Bufferable propagates towards its destination. This propagation means that, when forwarding a read transaction, the attempted forwarding must not continue indefinitely, and any data that is used for forwarding must not persist indefinitely. The protocol does not define any mechanism for determining how long data that is used for forwarding a read transaction can persist. However, in such a mechanism, the act of reading the data must not reset the data timeout period.

  Note

  Without this requirement, continued polling of the same location can prevent the timeout of a read that is held in the buffer, preventing the read progressing towards its destination.

- An intermediate buffer that can hold and merge write transactions must ensure that transactions do not remain in its buffer indefinitely. For example, merging write transactions must not reset the mechanism that determines when a write is drained towards its final destination.

  Note

  Without this requirement, continued writes to the same location can prevent the timeout of a write held in the buffer, preventing the write progressing towards its destination.

For information about the required behavior of read accesses to these memory types, see:

- *Device Bufferable* on page A4-70
- *Normal Non-cacheable Bufferable* on page A4-70
- *Write-Through No-Allocate* on page A4-71

A4.7 Access permissions

AXI provides access permissions signals that can be used to protect against illegal transactions:

- **ARPROT[2:0]** defines the access permissions for read accesses.
- **AWPROT[2:0]** defines the access permissions for write accesses.

The term **AxPROT** refers collectively to the **ARPROT** and **AWPROT** signals.

Table A4-6 shows the **AxPROT[2:0]** encoding.

Table A4-6 Protection encoding

AxPROT	Value	Function
[0]	0	Unprivileged access
	1	Privileged access
[1]	0	Secure access
	1	Non-secure access
[2]	0	Data access
	1	Instruction access

The protection attributes are:

Unprivileged or privileged

An AXI master might support more than one level of operating privilege, and extend this concept of privilege to memory access. **AxPROT[0]** identifies an access as unprivileged or privileged.

Note

Some processors support multiple levels of privilege, see the documentation for the selected processor to determine the mapping to AXI privilege levels. The only distinction AXI can provide is between privileged and unprivileged access.

Secure or Non-secure

An AXI master might support Secure and Non-secure operating states, and extend this concept of security to memory access. **AxPROT[1]** identifies an access as Secure or Non-secure. **AxPROT**[1] can be considered as defining two address spaces, a Secure address space and a Non-secure address space. This signal can be treated as an additional address bit. Any aliasing between the Secure and Non-secure address spaces must be handled correctly.

Note

This bit is defined so that when it is asserted the transaction is identified as Non-secure. This definition is consistent with other signaling in implementations of the Arm Security Extensions.

Instruction or data

This bit indicates that the transaction is an instruction access or a data access.

The AXI protocol defines this indication as a hint. It is not accurate in all cases, for example, where a transaction contains a mix of instruction and data items. This specification recommends that a master sets **AxPROT[2]** LOW, to indicate a data access unless the access is known to be an instruction access.

A4.8 Legacy considerations

AXI4 introduces additional requirements for the handling of some of the **AxCACHE** memory attributes.

In AXI4, all Device transactions using the same ID to the same slave must be ordered with respect to each other.

Note

- This ordering is not an explicit requirement of AXI3. Any AXI4 component that relies on this behavior cannot be connected to an AXI3 interconnect that does not exhibit this behavior.
- Arm believes that most implemented AXI3 interconnects support the required AXI4 behavior.

This specification strongly recommends that any new AXI3 design implements the AXI4 requirement.

For **AxCACHE** bits names and memory type names, it is required that AXI4 uses the new terms. AXI3 components can use either the AXI3 or AXI4 names.

A4.9 Usage examples

This section gives examples of memory type usage.

A4.9.1 Use of Device memory types

The specification supports the combined use of Device Non-bufferable and Device Bufferable memory types to force write transactions to reach their final destination and ensure that the issuing master knows when the transaction is visible to all other masters.

A write transaction that is marked as Device Bufferable is required to reach its final destination *in a timely manner*. However, the write response for the transaction can be signaled by an intermediate buffer. Therefore, the issuing master cannot know when the write is visible to all other masters.

If a master issues a Device Bufferable write transaction, or stream of write transactions, followed by a Device Non-bufferable write transaction, and all transactions use the same AXI ID, the AXI ordering requirements force all of the Device Bufferable write transactions to reach the final destination before a response is given to the Device Non-bufferable transaction. Therefore, the response to the Device Non-bufferable transaction indicates that all the transactions are visible to all masters.

Note

A Device Non-bufferable transaction can only guarantee the completion of Device Bufferable transactions that are issued with the same ID, and are to the same slave device.

Chapter A5
Transaction Identifiers

This chapter describes the mechanism that enables out-of-order transaction completion and the issuing of multiple outstanding addresses. It contains the following sections:

A5.1 AXI transaction identifiers

The AXI protocol includes AXI ID transaction identifiers. A master can use these to identify separate transactions that must be returned in order.

All transactions with a given AXI ID value must remain ordered, but there is no restriction on the ordering of transactions with different ID values. A single physical port can support out-of-order transactions by acting as a number of logical ports, each handling its transactions in order.

By using AXI IDs, a master can issue transactions without waiting for earlier transactions to complete. This can improve system performance, because it enables parallel processing of transactions.

Note

There is no requirement for slaves or masters to use AXI transaction IDs. Masters and slaves can process one transaction at a time. Transactions are processed in the order they are issued.

Slaves are required to reflect on the appropriate **BID** or **RID** response an AXI ID received from a master.

A5.2 ID signals

Each transaction channel has its own transaction ID. Table A5-1 shows these designated signals.

Table A5-1 Channel transaction ID

Transaction channel	Transaction ID
Write address channel	**AWID**
Write data channel, AXI3 only	**WID**[a]
Write response channel	**BID**
Read address channel	**ARID**
Read data channel	**RID**

a. The **WID** signal is implemented only in AXI3.

Note

The AXI4 protocol supports an extended ordering model based on the use of the AXI ID transaction identifier. See Chapter A6 *AXI Ordering Model*.

A5.2.1 Read data ordering

The slave must ensure that the **RID** value of any returned data matches the **ARID** value of the address that it is responding to.

The interconnect must ensure that the read data from a sequence of transactions with the same **ARID** value targeting different slaves is received by the master in the order that it issued the addresses.

The read data reordering depth is the number of addresses pending in the slave that can be reordered. A slave that processes all transactions in order has a read data reordering depth of one. The read data reordering depth is a static value that must be specified by the designer of the slave.

There is no mechanism that a master can use to determine the read data reordering depth of a slave.

A5.2.2 Write data ordering

A master must issue write data in the same order that it issues the transaction addresses.

An interconnect that combines write transactions from different masters must ensure that it forwards the write data in address order.

The interleaving of write data with different IDs was permitted in AXI3, but is deprecated in AXI4 and later. See the AMBA AXI and ACE Protocol Specification issue F specification for more details on write data interleaving.

A5.2.3 Interconnect use of transaction identifiers

When a master is connected to an interconnect, the interconnect appends additional bits to the **ARID**, **AWID** and **WID** identifiers that are unique to that master port. This has two effects:

- Masters do not have to know what ID values are used by other masters because the interconnect makes the ID values used by each master unique by appending the master number to the original identifier.
- The ID identifier at a slave interface is wider than the ID identifier at a master interface.

For read data, the interconnect uses the additional bits of the **RID** identifier to determine which master port the read data is destined for. The interconnect removes these bits of the **RID** identifier before passing the **RID** value to the correct master port.

For write response, the interconnect uses the additional bits of the **BID** identifier to determine which master port the write response is destined for. The interconnect removes these bits of the **BID** identifier before passing the **BID** value to the correct master port.

Chapter A6
AXI Ordering Model

This chapter specifies:

A6.1 AXI ordering model overview

The AXI ordering model is based on the use of the transaction identifier, which is signaled on **ARID** or **AWID**.

Transaction requests on the same channel, with the same ID and destination are guaranteed to remain in order.

Transaction responses with the same ID are returned in the same order as the requests were issued.

The ordering model does not give any ordering guarantees between:

- Transactions from different masters
- Read and write transactions
- Transactions with different IDs
- Transactions to different Peripheral regions
- Transactions to different Memory locations

If a master requires ordering between transactions that have no ordering guarantee, the master must wait to receive a response to the first transaction before issuing the second transaction.

A6.2 Memory locations and Peripheral regions

The address map in AMBA is made up of Memory locations and Peripheral regions.

A Memory location has all of the following properties:

- A read of a byte from a Memory location returns the last value that was written to that byte location.
- A write to a byte of a Memory location updates the value at that location to a new value that is obtained by a subsequent read of that location.
- Reading or writing to a Memory location has no side-effects on any other Memory location.
- Observation guarantees for Memory are given for each location.
- The size of a Memory location is equal to the single-copy atomicity size for that component.

A Peripheral region has all of the following properties:

- A read from an address in a Peripheral region does not necessarily return the last value that is written to that address.
- A write to a byte address in a Peripheral region does not necessarily update the value at that address to a new value that is obtained by subsequent reads.
- Accessing an address within a Peripheral region might have side-effects on other addresses within that region.
- Observation guarantees for Peripherals are given per region.
- The size of a Peripheral region is IMPLEMENTATION DEFINED, but it must be contained within a single slave component.

A6.3 Transactions and ordering

A transaction is a read or a write to one or more address locations. The locations are determined by **AxADDR** and any relevant qualifiers such as the Non-secure bit in **AxPROT**.

- Ordering guarantees are given only between accesses to the same Memory location or Peripheral region.
- A transaction to a Peripheral region must be entirely contained within that region.
- A transaction that spans multiple Memory locations has multiple ordering guarantees.

Transactions can be either of type Device or Normal:

Device A read or write where the request has **AxCACHE[1]** deasserted.

Device transactions can be used to access Peripheral regions or Memory locations.

Normal A read or write where the request has **AxCACHE[1]** asserted.

Normal transactions are used to access Memory locations and are not expected to be used to access Peripheral regions.

A Normal access to a Peripheral region must complete in a protocol-compliant manner, but the result is IMPLEMENTATION DEFINED.

A write transaction can be either Non-bufferable or Bufferable. It is possible to send an early response to Bufferable writes.

- A Non-bufferable write has **AWCACHE[0]** deasserted.
- A Bufferable write has **AWCACHE[0]** asserted.

A6.4 Observation and completion definitions

For accesses to Peripheral regions, a Device read or write access DRW1 is observed by a Device read or write access DRW2, when DRW1 arrives at the slave component before DRW2.

For accesses to Memory locations, all of the following apply:

- A write W1 is observed by a write W2, if W2 takes effect after W1.
- A read R1 is observed by a write W2, if R1 returns data from a write W3, when W2 is after W3.
- A write W1 is observed by a read R2, if R2 returns data from either W1 or from write W3, when W3 is after W1.

Read R1 or write W1 can be of type Device or Normal.

The definitions of write and read completions are:

Write completion response

The cycle when the associated **BRESP** handshake is given, when **BVALID** and **BREADY** are asserted.

Read completion response

The cycle when the last associated **RDATA** handshake is given, when **RVALID**, **RLAST** and **RREADY** are asserted.

A6.5 Master ordering guarantees

There are three types of ordering model guarantees:

- Observability guarantees before a completion response is received.
- Observability guarantees from a completion response.
- Response ordering guarantees.

A6.5.1 Guarantees before a completion response is received

All of the following guarantees apply to transactions from the same master, using the same ID:

- A Device write DW1 is guaranteed to arrive at the destination before Device write DW2, where DW2 is issued after DW1 and to the same Peripheral region.
- A Device read DR1 is guaranteed to arrive at the destination before Device read DR2, where DR2 is issued after DR1 and to the same Peripheral region.
- A write W1 is guaranteed to be observed by a write W2, where W2 is issued after W1 and to the same Memory location.
- A write W1 that has been observed by a read R2 is guaranteed to be observed by a read R3, where R3 is issued after R2 and to the same Memory location.

The guarantees imply that there are ordering guarantees between Device and Normal accesses to the same Memory location.

A6.5.2 Guarantees from a completion response

A completion response guarantees all of the following:

- A completion response to a read request guarantees that it is observable to a subsequent read or write request from any master.
- A completion response to a write request guarantees that it is observable to a subsequent read or write request from any master. This observability is a requirement of a system that is multi-copy atomic.

Systems that contain Arm Architecture-compliant processors must be multi-copy atomic. That is, the Multi_Copy_Atomicity property must be True.

The response to a Bufferable write request can be sent from an intermediate point. It does not guarantee that the write has completed at the endpoint, but it is observable to future transactions.

A6.5.3 Response ordering guarantees

Transaction responses have all the following ordering guarantees:

- A read R1 is guaranteed to receive a response before the response to a read R2, where R2 is issued from the same master after R1 with the same ID.
- A write W1 is guaranteed to receive a response before the response to a write W2, where W2 is issued from the same master after W1 with the same ID.

A6.6 Ordering requirements

To meet the master ordering guarantees, there are certain requirements on slave and interconnect components.

A6.6.1 Slave ordering requirements

For Peripheral locations, the execution order of transactions to Peripheral locations is IMPLEMENTATION DEFINED. This execution order is typically expected to match the arrival order, but that is not a requirement.

For Memory locations:

- A write W1 must be ordered before a write W2 with the same ID, to the same Memory location, where W2 is received after W1 is received.
- A write W1 must be ordered before a write W2 to the same Memory location, where W2 is received after the completion response for W1 is given.
- A write W1 must be ordered before a read R2 to the same Memory location, where R2 is received after the completion response for W1 is given.
- A read R1 must be ordered before a write W2 to the same Memory location, where W2 is received after the completion response for R1 is given.

Response ordering requirements:

- The response to read R1 must be returned before the response to a read R2, where R2 is received after R1 with the same ID.
- The response to write W1 must be returned before the response to a write W2, where W2 is received after W1 with the same ID.

A6.6.2 Interconnect ordering requirements

An interconnect component has the following attributes:

- A request is received on one port and is either issued on a different port or responded to.
- A response is received on one port and is either issued on a different port or consumed.

When the interconnect issues requests or responses, it must adhere to the following requirements:

- A read R1 request must be issued before a read R2 request, where R2 is received after R1, with the same ID and to the same or overlapping locations.
- A write W1 request must be issued before a write W2 request, where W2 is received after W1, with the same ID, to the same or overlapping locations.
- A Device read DR1 request must be issued before a Device read DR2 request, where DR2 is received after DR1, with the same ID and to the same Peripheral region.
- A Device write DW1 request must be issued before a Device write DW2 request, where DW2 is received after DW1, with the same ID and to the same Peripheral region.
- A read R1 response must be issued before a read R2 response, where R2 is received after R1, with the same ID.
- A write W1 response must be issued before a write W2 response, where W2 is received after W1, with the same ID.

When the interconnect is acting as a slave component, it must also adhere to the slave requirements.

Any manipulation of the AXI ID values that are associated with a transaction must ensure that the ordering requirements of the original ID values are maintained.

A6.7 Response before the endpoint

To improve system performance, it is possible for an intermediate component to issue a response to some transactions. This action is known as an early response. The intermediate component issuing an early response must ensure that visibility and ordering guarantees are met.

A6.7.1 Early read response

For Normal read transactions, an intermediate component can respond with read data from a local memory if it is up-to-date with respect to all earlier writes to the same or overlapping address. In this case, the request is not required to propagate beyond the intermediate component.

An intermediate component must observe ID ordering rules, which means a read response can only be sent if all earlier reads with the same ID have already had a response.

A6.7.2 Early write response

For Bufferable write transactions, an intermediate component can send an early write response for transactions that have no downstream observers. If the intermediate component sends an early write response, the intermediate component can store a local copy of the data, but must propagate the transaction downstream, before discarding that data.

An intermediate component must observe ID ordering rules, that means a write response can only be sent if all earlier writes with the same ID have already had a response.

After sending an early write response, the component must be responsible for ordering and observability of that transaction until the write has been propagated downstream and a write response is received. During the period between sending the early write response and receiving a response from downstream, the component must ensure that:

- If an early write response was given for a Normal transaction, all subsequent transactions to the same or overlapping Memory locations are ordered after the write that has had an early response.
- If an early write response was given for a Device transaction, then all subsequent transactions to the same Peripheral region are ordered after the write that has had an early response.

When giving an early write response for a Device Bufferable transaction, the intermediate component is expected to propagate the write transaction without dependency on other transactions. The intermediate component cannot wait for another read or write to arrive before propagating a previous Device write.

A6.8 Ordered write observation

To improve compatibility with interface protocols that support a different ordering model, a slave interface can give stronger ordering guarantees for write transactions. A stronger ordering guarantee is known as Ordered Write Observation.

The Ordered_Write_Observation property is used to define whether an interface exhibits Ordered Write Observation, it can be True or False for a single interface.

True An interface is defined as having the Ordered Write Observation property.

False An interface that does not have the Ordered Write Observation property.

If Ordered_Write_Observation is not declared, it is considered False.

An interface that exhibits Ordered Write Observation gives guarantees for write transactions that are not dependent on the destination or address:

- A write W1 is guaranteed to be observed by a write W2, where W2 is issued after W1, from the same master, with the same ID.

A master using the Producer-Consumer ordering model that is connected to a slave interface that exhibits Ordered Write Observation is not required to wait for the completion response from earlier writes before issuing dependent writes.

Chapter A7
Atomic Accesses

This chapter describes the AXI4 concept of single-copy atomicity size and how the AXI protocol implements exclusive access and locked access mechanisms. It contains the following sections:

A7.1 Single-copy atomicity size

The AXI4 protocol introduces the concept of single-copy atomicity size. This term defines the minimum number of bytes that a transaction updates atomically. The AXI4 protocol requires a transaction that is larger than the single-copy atomicity size must update memory in blocks of at least the single-copy atomicity size.

Note

Atomicity does not define the exact instant when the data is updated. What must be ensured is that no master can ever observe a partially updated form of the atomic data. For example, in many systems data structures such as linked lists are made up of 32-bit atomic elements. An atomic update of one of these elements requires that the entire 32-bit value is updated at the same time. It is not acceptable for any master to observe an update of only 16-bits at one time, and then the update of the other 16-bits at a later time.

More complex systems require support for larger atomic elements, in particular 64-bit atomic elements, so that masters can communicate using data structures that are based on these larger atomic elements.

The single-copy atomicity sizes that are supported in a system are important because all of the components involved in a given communication must support the required size of atomic element. If two masters are communicating through an interconnect and a single slave, then all of the components involved must ensure that transactions of the required size are treated atomically.

The AXI4 protocol does not require a specific single-copy atomicity size and systems can be designed to support different single-copy atomicity sizes.

Different groups of components can have different single-copy atomicity sizes for communication within the groups. In AXI4 the term single-copy atomic group describes a group of components that can communicate at a particular atomicity. For example, Figure A7-1 shows a system in which:

- The processor, *Digital Signal Processor* (DSP), DRAM controller, DMA controller, peripherals, SRAM memory and associated interconnect, are in a 32-bit single-copy atomic group.
- The processor, DSP, DRAM controller, and associated interconnect, are also in a 64-bit single-copy atomic group.

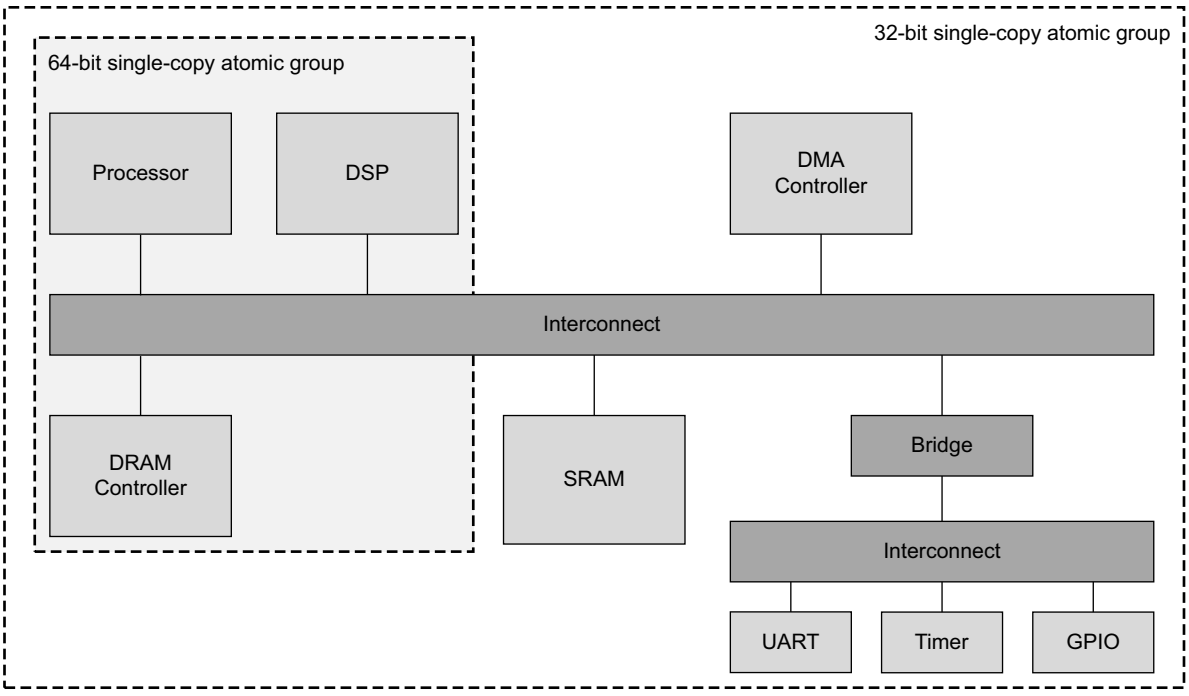


Figure A7-1 Example system with different single-copy atomic groups

A transaction never has an atomicity guarantee greater than the alignment of its start address. For example, a burst in a 64-bit single-copy atomic group that is not aligned to an 8-byte boundary does not have any 64-bit single-copy atomic guarantee.

Byte strobes associated with a transaction do not affect the single-copy atomicity size.

A7.1.1 Multi-copy write atomicity

To specify that a system provides multi-copy atomicity, a Multi_Copy_Atomicity property is defined.

True Multi_Copy_Atomicity is supported.

False Multi_Copy_Atomicity is not supported. If Multi_Copy_Atomicity is not declared, it is considered False.

A system is defined as being multi-copy atomic if:

- Writes to the same location are observed in the same order by all agents.
- A write to a location that is observable by an agent, is observable by all agents.

Multi-copy atomicity can be ensured by:

- Using a single *Point of Serialization*, for a given address, so that all accesses to the same location are ordered. This must ensure that all coherent cached copies of a location are invalidated before the new value of the location is made visible to any agents.
- Avoiding the use of forwarding buffers that are upstream of any agents. This prevents a buffered write of a location becoming visible to some agents before it is visible to all agents.
- It is required that the Multi_Copy_Atomicity property is True for Issue G and later of this specification.

Note

A system must have the Multi_Copy_Atomicity property if Arm v8 Architecture processors are used. This is required to support the Load with Acquire and Store with Release instructions. The Store with Release instruction requires that the store is multi-copy atomic.

A7.2 Exclusive accesses

The exclusive access mechanism can provide semaphore-type operations without requiring the bus to remain dedicated to a particular master for the duration of the operation. This means the semaphore-type operations do not impact either the bus access latency or the maximum achievable bandwidth.

The **AxLOCK** signals select exclusive access, and the **RRESP** and **BRESP** signals indicate the success or failure of the exclusive access read or write respectively.

The slave requires additional logic to support exclusive access. The AXI protocol provides a mechanism to indicate when a master attempts an exclusive access to a slave that does not support it. The remainder of this section describes the AXI Exclusive access mechanism.

A7.2.1 Exclusive access process

The basic mechanism of an exclusive access is:

1. A master performs an exclusive read from an address.
2. At some later time, the master attempts to complete the exclusive operation by performing an exclusive write to the same address, and with an **AWID** that matches the **ARID** used for the exclusive read.
3. This exclusive write access is signaled as either:
   - Successful, if no other master has written to that location since the exclusive read access. In this case the exclusive write updates memory.
   - Failed, if another master has written to that location since the exclusive read access. In this case the memory location is not updated.

   A master might not complete the write portion of an exclusive operation. The exclusive access monitoring hardware monitors only one address for each transaction ID. If a master does not complete the write portion of an exclusive operation, a subsequent exclusive read by that master using the same transaction ID changes the address that is being monitored for exclusive accesses.

A7.2.2 Exclusive access from the perspective of the master

A master starts an exclusive operation by performing an exclusive read. If the transaction is successful, the slave returns the EXOKAY response, indicating that the slave recorded the address to be monitored for exclusive accesses.

If the master attempts an exclusive read from a slave that does not support exclusive accesses, the slave returns the OKAY response instead of the EXOKAY response.

Note

The master can treat the OKAY response as an error condition indicating that the exclusive access is not supported. This specification recommends that the master does not perform the write portion of this exclusive operation.

At some time after the exclusive read, the master tries an exclusive write to the same location. If the contents of the addressed location have not been updated since the exclusive read, the exclusive write operation succeeds. The slave returns the EXOKAY response, and updates the memory location.

If the contents of the addressed location have been updated since the exclusive read, the exclusive write attempt fails, and the slave returns the OKAY response instead of the EXOKAY response. The exclusive write attempt does not update the memory location.

A master might not complete the write portion of an exclusive operation. If this happens, the slave continues to monitor the address for exclusive accesses until another exclusive read starts a new exclusive access sequence.

A master must not start the write part of an exclusive access sequence until the read part is complete.

A7.2.3 Exclusive access from the perspective of the slave

A slave that does not support exclusive accesses can ignore the **AxLOCK** signals. It must provide an OKAY response for both normal and exclusive accesses.

A slave that supports exclusive access must have monitor hardware. This specification recommends that such a slave has a monitor unit for each exclusive-capable master ID that can access it. The *Arm Architecture Reference Manual, Armv7-A and Armv7-R edition* defines an exclusive access monitor, and a single-ported slave can have such an exclusive access monitor external to the slave. A multiported slave might require internal monitoring.

The exclusive access monitor records the address and **ARID** value of any exclusive read operation. Then it monitors that location until either a write occurs to that location or until another exclusive read with the same **ARID** value resets the monitor to a different address.

When the slave receives an exclusive write with a given **AWID** value, the monitor checks to see if that address is being monitored for exclusive access with that **AWID**. If it is, then this indicates that no write has occurred to that location since the exclusive read access, and the exclusive write proceeds, completing the exclusive access. The slave returns the EXOKAY response to the master, and updates the addressed memory location.

If the address is not being monitored with the same **AWID** value at the time of an exclusive write, this indicates one of the following:

- The location has been updated since the exclusive read access.
- The monitor has been reset to another location.
- The master did not issue an exclusive read with the same attributes as the exclusive write.

In both cases the exclusive write must not update the addressed location, and the slave must return the OKAY response instead of the EXOKAY response.

A7.2.4 Exclusive access restrictions

The following restrictions apply to exclusive accesses:

- The address of an exclusive access must be aligned to the total number of bytes in the transaction, that is, the product of the burst size and burst length.
- The number of bytes to be transferred in an exclusive access burst must be a power of 2, that is, 1, 2, 4, 8, 16, 32, 64, or 128 bytes.
- The maximum number of bytes that can be transferred in an exclusive burst is 128.
- The burst length for an exclusive access must not exceed 16 transfers.
- The value of the AxCACHE signals must guarantee that the transaction reaches the slave that is monitoring exclusive accesses. If there is a buffer or cache which might respond to an exclusive access before it reaches the monitor, then the exclusive access must be Non-bufferable or Non-cacheable.
- The domain must be Non-shareable or System-shareable.
- The transaction type must be ReadNoSnoop or WriteNoSnoop.

Failure to observe these restrictions causes UNPREDICTABLE behavior.

For an exclusive read and an exclusive write to be considered part of the same exclusive access sequence, the following signals must be the same for both transfers:

- **AxID**
- **AxADDR**
- **AxREGION**
- **AxLEN**
- **AxSIZE**
- **AxBURST**
- **AxLOCK**
- **AxCACHE**

- **AxPROT**
- **AxDOMAIN**
- **AxSNOOP**
- **AxMMUSECSID**
- **AxMMUSID**
- **AxMMUSSIDV**
- **AxMMUSSID**
- **AxMMUATST**

The minimum number of bytes to be monitored during an exclusive operation is defined by the burst length and burst size of the transaction. The slave can monitor a larger number of bytes, up to 128, which is the maximum size of an exclusive access. However, this can result in a successful exclusive access being indicated as failing because a neighboring byte was updated.

A7.2.5 Responses to exclusive access

The response signals, **RRESP** and **BRESP**, include an OKAY response for successful normal accesses and an EXOKAY response for successful exclusive accesses. This means that a slave that does not support exclusive accesses can provide an OKAY response to indicate the failure of an exclusive access.

Note

- An exclusive write to a slave that does not support exclusive access always updates the memory location.
- An exclusive write to a slave that supports exclusive access updates the memory location only if the exclusive write is successful.

An exclusive write has a single response using **BRESP**, which can be OKAY, EXOKAY, SLVERR or DECERR.

An exclusive read has one or more response beats. These can be a mixture of EXOKAY, SLVERR and DECERR or a mixture of OKAY, SLVERR and DECERR. Mixing EXOKAY and OKAY responses in the same transaction is not permitted.

A7.3 Locked accesses

AXI4 does not support locked transactions. However, an AXI3 implementation must support locked transactions.

Note

AXI4 removes support for locked transactions because:

- The majority of components do not require locked transactions.
- The implementation of locked transactions has a significant effect on:
  - The complexity of the interconnect
  - The ability to make QoS guarantees

In this specification, **AxLOCK** indicates **ARLOCK** or **AWLOCK**.

When a master uses the **AxLOCK** signals for a transaction to show that it is a locked transaction then the interconnect must ensure that only that master can access the targeted slave region, until an unlocked transaction from the same master completes. An arbiter within the interconnect must enforce this restriction.

Before a master starts a locked sequence of either read or write transactions it must ensure that it has no other transactions waiting to complete.

Any transaction with **AxLOCK** indicating a locked transaction forces the interconnect to lock the following transaction. Therefore, a locked sequence must always complete with a final transaction that does not have **AxLOCK** indicating a locked transaction. This final transaction is included in the locked sequence and effectively removes the lock.

When completing a locked sequence, before issuing the final unlocking transaction, a master must ensure that all previous locked transactions are complete. It must then ensure that the final unlocking transaction has completed before it starts any further transactions.

The master must ensure that all transactions in a locked sequence have the same **AxID** value.

Note

Locked accesses require the interconnect to prevent any other transactions occurring while the locked sequence is in progress, and can therefore have an impact on the interconnect performance. This specification recommends that locked accesses are only used to support legacy devices.

This specification recommends the following restrictions, but they are not mandatory:

- Keep any locked transaction sequence within a single 4 KB address region.
- Limit any locked transaction sequence to two transactions.

A7.4 Atomic access signaling

In AXI3 the **AxLOCK** signals specify normal, exclusive, and locked accesses. Table A7-1 shows the AXI3 encoding of the **AxLOCK** signals.

Table A7-1 AXI3 atomic access encoding

AxLOCK[1:0]	Access type
0b00	Normal access
0b01	Exclusive access
0b10	Locked access
0b11	Reserved

AXI4 removes the support for locked transactions and uses only a 1-bit lock signal. Table A7-2 shows the AXI4 signal encoding of the **AxLOCK** signals.

Table A7-2 AXI4 atomic access encoding

AxLOCK	Access type
0b0	Normal access
0b1	Exclusive access

A7.4.1 Legacy considerations

In an AXI4 environment, any AXI3 locked transaction is converted as follows:

- **AWLOCK[1:0]** = 0b10 is converted to a normal write transaction, **AWLOCK** = 0b0
- **ARLOCK[1:0]** = 0b10 is converted to a normal read transaction, **ARLOCK** = 0b0

This specification recommends that any component performing such a conversion, typically an interconnect, includes an optional mechanism to detect and flag that such a translation has occurred.

Any component that cannot operate correctly if this translation is performed cannot be used in an AXI4 environment.

Note

For many legacy cases that use locked transactions, such as the execution of a SWP instruction, a software change might be required to prevent the use of any instruction that forces a locked transaction.

Chapter A8
AMBA 4 Additional Signaling

This chapter describes the additional signaling that is introduced in AMBA 4 to extend the application of the AXI interface. It contains the following sections:

A8.1 QoS signaling

This section describes the additional signaling in the AXI4 protocol to support *Quality of Service* (QoS).

A8.1.1 QoS interface signals

The AXI4 signal set is extended to support two 4-bit QoS identifiers:

AWQOS A 4-bit QoS identifier, sent on the write address channel for each write transaction.

ARQOS A 4-bit QoS identifier, sent on the read address channel for each read transaction.

In this specification, **AxQOS** indicates **AWQOS** or **ARQOS**.

The protocol does not specify the exact use of the QoS identifier. This specification recommends that **AxQOS** is used as a priority indicator for the associated write or read transaction. A higher value indicates a higher priority transaction.

A default value of `0b0000` indicates that the interface is not participating in any QoS scheme.

Note

Additional interpretations of the QoS identifier can be used.

A8.1.2 Master considerations

A master can produce its own **AxQOS** values, and if it can produce multiple streams of traffic it can choose different QoS values for the different streams.

Support for QoS requires a system-level understanding of the QoS scheme in use, and collaboration between all participating components. For this reason, this specification recommends that a master component includes some programmability that can be used to control the exact QoS values that are used for any given scenario.

If a master component does not support a programmable QoS scheme, it can use QoS values that represent the relative priorities of the transactions it generates. These values can then be mapped to alternative system level QoS values if appropriate.

A master that cannot produce its own **AxQOS** values must use the default value.

Note

This specification expects that many interconnect component implementations will support programmable registers that can be used to assign QoS values to connected masters. These values replace the QoS values, either programmed or default, supplied by the masters.

A8.1.3 System considerations

QoS signaling, as defined in AXI4, can be used with any compatible system-level QoS methodology.

The default system-level implementation of QoS is that any component with a choice of more than one transaction to process selects the transaction with the higher QoS value to process first. This selection only occurs when there is no other AXI constraint that requires the transactions to be processed in a particular order.

Note

This means that the AXI ordering rules take precedence over ordering for QoS purposes.

More sophisticated QoS schemes that are compatible with this default scheme can be implemented.

A8.2 Multiple region signaling

This section describes the optional additional signaling in the AXI4 protocol to support multiple region interfaces.

A8.2.1 Additional interface signals

Optionally, the AXI4 interface signal set can be extended to support two 4-bit region identifiers:

AWREGION A region identifier, sent on the write address channel for each write transaction.

ARREGION A region identifier, sent on the read address channel for each read transaction.

In this specification, **AxREGION** indicates **AWREGION** or **ARREGION**.

The 4-bit region identifier can be used to uniquely identify up to 16 different regions. The region identifier can provide a decode of higher-order address bits. The region identifier must remain constant within any 4K-byte address space.

The use of region identifiers means that a single physical interface on a slave can provide multiple logical interfaces, each with a different location in the system address map. The use of the region identifier means that the slave does not have to support the address decode between the different logical interfaces.

This protocol expects an interconnect to produce **AxREGION** signals when performing the address decode function for a single slave that has multiple logical interfaces. If a slave only has a single physical interface in the system address map, the interconnect must use the default **AxREGION** values. See Chapter A9 *Default Signaling and Interoperability*.

There are a number of usage models for the region identifier including, but not limited to, the following:

- A peripheral can have its main data path and control registers at different locations in the address map, and be accessed through a single interface without the need for the slave to perform an address decode.
- A slave can exhibit different behaviors in different memory regions. For example, a slave might provide read and write access in one region, but read only access in another region.

A slave must ensure the correct protocol signaling and the correct ordering of transactions are maintained. A slave must ensure that it provides the responses to two transactions to different regions with the same AXI ID in the correct order.

A slave must also ensure the correct protocol signaling for any values of **AxREGION**. If a slave implements fewer than sixteen regions, then the slave must ensure the correct protocol signaling on any attempted access to an unsupported region. How this is achieved is IMPLEMENTATION DEFINED. For example, the slave might ensure this by:

- Providing an error response for any transaction that accesses an unsupported region.
- Aliasing supported regions across all unsupported regions, to ensure that a protocol-compliant response is given for all accesses.

The **AxREGION** signals only provide an address decode of the existing address space that can be used by slaves to remove the need for an address decode function. The signals do not create new independent address spaces. **AxREGION** must only be present on an interface that is downstream of an address decode function.

A8.3 User-defined signaling

User-defined signaling was introduced in AMBA 4. For AMBA 5, the usage and configuration of these signals is further clarified and unified across protocols.

An AXI interface can include a set of user-defined signals, called the User signals. The signals can be used to augment information to a transaction, where there is a requirement that is not covered by the existing AMBA specification.

Information can be added to:

- A transaction request
- A transaction response
- Each beat of read or write data within a transaction

Generally, this specification recommends not to use User signals. The AXI protocol does not define the functions of these signals, which can lead to interoperability issues if two components use the same User signals in an incompatible manner.

A8.3.1 User-defined signaling configuration

The presence and width of User signals is specified by the properties in Table A8-1:

Table A8-1 User signals properties

Name	Min	Max	Applies to
USER_REQ_WIDTH	0	128	**AWUSER**, **ARUSER**
USER_DATA_WIDTH	0	DATA_WIDTH/2	**WUSER**, **RUSER**
USER_RESP_WIDTH	0	16	**BUSER**, **RUSER**

If a property has a value of zero, then the associated signals are not present on the interface.

The maximum values are for guidance only, in order to set a reasonable maximum for configurable interfaces.

The following interface types can include User signals:

- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP
- AXI5
- AXI5-Lite

A8.3.2 Signaling

The User signal names that are defined for each channel are shown in Table A8-2:

Table A8-2 User-defined signals

Signal	Width	Description
ARUSER	USER_REQ_WIDTH	User-defined read request attribute. This has the same validity requirements as the other AR channel signals.
AWUSER	USER_REQ_WIDTH	User-defined write request attribute. This has the same validity requirements as the other AW channel signals.
WUSER	USER_DATA_WIDTH	User-defined write data attribute. This has the same validity requirements as the other W channel signals.
RUSER	USER_DATA_WIDTH + USER_RESP_WIDTH	User-defined read data and response attribute. This has the same validity requirements as the other R channel signals.
BUSER	USER_RESP_WIDTH	User-defined write response attribute. This has the same validity requirements as the other B channel signals.

A8.3.3 Usage considerations

Where User signals are implemented, it is not required that User signals are supported on all channels. The design decision whether to include User signals is made independently for request, data, and response channels.

To assist with data width and protocol conversion, this specification recommends that:

- USER_DATA_WIDTH is an integer multiple of the width of the data buses in bytes.
- User response bits are the same value for every beat of a read or write response.
- The lower bits of **RUSER** are used to transport per-transaction response information.
- The upper bits of **RUSER** are used to transport per-beat read data information.

Chapter A9
Default Signaling and Interoperability

This chapter describes the default signaling and interoperability of the AXI interface.

The AXI protocol does not require a component to use the full set of signals available on an AXI interface. To assist in the connection of components that do not use every signal, this chapter defines the major categories of interfaces together with the restrictions that apply to each category. It contains the following sections:

A9.1 Interoperability principles

The following interoperability principles apply to both AXI3 and AXI4 components.

As a general principle, components must support all combinations of inputs, but do not have to generate all combinations of outputs. For example, a slave must support all the different possible lengths of burst, but a master only has to generate the types of burst that it uses. This policy ensures that all components work with all other components.

The conditions that a signal can be omitted from an AXI interface are:

Optional Outputs

If a component might require a value that does not match the default value, then the component must have the output signal present.

If a component always requires the value that matches the default value, specified in *Default signal values* on page A9-110, then it is not required that the component has the signal present.

Optional Inputs

An input signal can be omitted if the master or slave does not need to observe the input signal for correct functional operation.

Note

Interconnect components can also omit signals when appropriate. For example, when a signal is only ever driven to its default value, there is no requirement to transport that signal across the interconnect. The signal can be created at its destination. Similarly, if a signal is not used at any destination then there is no requirement to transport it across the interconnect.

A9.2 Major interface categories

The following sections describe the major interface categories.

A9.2.1 Read/write interface

A read write interface includes the following AXI channels:

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

A9.2.2 Read-only interface

A read-only interface supports only read transactions and includes the following AXI channels:

AR Read address channel
R Read data channel

Note

A read-only interface does not support exclusive accesses.

A9.2.3 Write-only interface

A write-only interface supports only write transactions and includes the following AXI channels:

AW Write address channel
W Write data channel
B Write response channel

Note

A write-only interface does not support exclusive accesses.

A9.2.4 Memory slaves and peripheral slaves

AXI slaves are classified as Memory slaves or Peripheral slaves.

A memory slave must handle all transaction types correctly.

Peripheral Slaves are expected to have a defined method of access that establishes the types of transaction that can be used to access a device, and if there are any restrictions on how the device is accessed. Typically, the defined method of access is described in the data sheet for the component. Any access that is not a defined method of access might cause the peripheral slave to fail but is expected to complete in a protocol-compliant fail-safe manner, to prevent system deadlock. Continued correct operation of the peripheral slave is not required.

Because a peripheral slave is required to work correctly only for its defined method of access, a peripheral slave can have a significantly reduced set of interface signals.

Note

All peripherals are expected to support a subset of transactions that permit the peripheral to be controlled using accesses that can be specified in C code. For example, single 8-bit, single 16-bit or single 32-bit aligned transactions might be supported.

No minimum subset is required, because the subset of supported transactions can differ between peripherals. For example, one peripheral might only support 16-bit accesses and another peripheral might only support 32-bit accesses.

A9.3 Default signal values

This specification suggests that, in general, for maximum IP reuse, an AXI component interface includes all signals. The presence of all signals reduces the risk of error at the system integration phase of the design flow and it can also help support some design flows that do not effectively support default values for absent signals.

The following tables show the AXI required and optional signals, and the default signals values that apply when an optional signal is not implemented:

- Table A9-1 shows the master interface write channel signals.
- Table A9-2 on page A9-112 shows the memory slave interface write channel signals.
- Table A9-3 on page A9-113 shows the master interface read channel signals.
- Table A9-4 on page A9-114 shows the memory slave interface read channel.

The following sections give more information about the default signal requirements:

- *Master addresses* on page A9-114
- *Slave addresses* on page A9-115
- *Memory slaves* on page A9-115
- *Write transactions* on page A9-115
- *Read transactions* on page A9-115
- *Response signaling* on page A9-115
- *Non-secure and Secure accesses* on page A9-115

Table A9-1 Master interface write channel signals and default signal values

Signal	Direction	Required?	Default
ACLK	Input	Required	-
ARESETn	Input	Required	-
AWID	Output	Optional	All zeros
AWADDR	Output	Required	-
AWREGION	Output	Optional	All zeros
AWLEN	Output	Optional	All zeros, Length 1
AWSIZE	Output	Optional	Data bus width
AWBURST	Output	Optional	`0b01`, INCR
AWLOCK	Output	Optional	All zeros, Normal access
AWCACHE	Output	Optional	`0b0000`
AWPROT	Output	Required	-
AWQOS	Output	Optional	`0b0000`
AWVALID	Output	Required	-
AWREADY	Input	Required	-
WDATA	Output	Required	-
WSTRB	Output	Optional	All ones
WLAST	Output	Required	-
WVALID	Output	Required	-
WREADY	Input	Required	-

Table A9-1 Master interface write channel signals and default signal values (continued)

Signal	Direction	Required?	Default
BID	Input	Optional	-
BRESP	Input	Optional	-
BVALID	Input	Required	-
BREADY	Output	Required	-

Table A9-2 Memory slave interface write channel signals and default signal values

Signal name	Direction	Required?	Default
ACLK	Input	Required	-
ARESETn	Input	Required	-
AWID	Input	Required	-
AWADDR	Input	Required	-
AWREGION	Input	Optional	-
AWLEN	Input	Required	-
AWSIZE	Input	Required	-
AWBURST	Input	Required	-
AWLOCK	Input	Optional	-
AWCACHE	Input	Optional	-
AWPROT	Input	Optional	-
AWQOS	Input	Optional	-
AWVALID	Input	Required	-
AWREADY	Output	Required	-
WDATA	Input	Required	-
WSTRB	Input	Required	-
WLAST	Input	Optional	-
WVALID	Input	Required	-
WREADY	Output	Required	-
BID	Output	Required	-
BRESP	Output	Optional	`0b00`, OKAY
BVALID	Output	Required	-
BREADY	Input	Required	-

Table A9-3 Master interface read channel signals and default signals values

Signal name	Direction	Required?	Default
ARID	Output	Optional	All zeros
ARADDR	Output	Required	-
ARREGION	Output	Optional	`0x0`
ARLEN	Output	Optional	All zeros, Length 1
ARSIZE	Output	Optional	Data bus width
ARBURST	Output	Optional	`0b01`, INCR
ARLOCK	Output	Optional	All zeros, Normal access
ARCACHE	Output	Optional	`0b0000`
ARPROT	Output	Required	-
ARQOS	Output	Optional	`0b0000`
ARVALID	Output	Required	-
ARREADY	Input	Required	-
RID	Input	Optional	-
RDATA	Input	Required	-
RRESP	Input	Optional	-
RLAST	Input	Optional	-
RVALID	Input	Required	-
RREADY	Output	Required	-

Table A9-4 Memory slave interface read channel signals and default signals values

Signal name	Direction	Required?	Default
ARID	Input	Required	-
ARADDR	Input	Required	-
ARREGION	Input	Optional	-
ARLEN	Input	Required	-
ARSIZE	Input	Required	-
ARBURST	Input	Required	-
ARLOCK	Input	Optional	-
ARCACHE	Input	Optional	-
ARPROT	Input	Optional	-
ARQOS	Input	Optional	-
ARVALID	Input	Required	-
ARREADY	Output	Required	-
RID	Output	Required	-
RDATA	Output	Required	-
RRESP	Output	Optional	`0b00`, OKAY
RLAST	Output	Required	-
RVALID	Output	Required	-
RREADY	Input	Required	-

A9.3.1 Master addresses

AxADDR There is no minimum requirement for the number of address bits supplied by a master.

If the system that the master is connected to has a different address bus width than that provided by the master:

- If the system address is wider than is provided by the master then the default value of all zeros must be used for the additional high-order address bits.
- If the system address is narrower than is provided by the master then the high-order address bits from the master must be left unconnected.

Note

Typically a master supplies 32-bits of addressing, optionally a master can support up to 64-bits of addressing.

A9.3.2 Slave addresses

AxADDR There is no minimum requirement for the number of address bits used by a slave.

A slave is not required to have low-order address bits to support decoding within the width of the system data bus and can assume that such low-order address bits have a default value of all zeros. If the slave has more address bits than supplied by the interconnect, the higher order address bits use a default value of all zeros.

Typically a memory slave has at least enough address bits to fully decode a 4KB address range.

A9.3.3 Memory slaves

AxLOCK A memory slave is not required to use the **AxLOCK** inputs. However, a memory slave that supports exclusive accesses requires these signals.

AxCACHE A memory slave is not required to make use of the **AxCACHE** inputs. A memory slave does not require these signals if either:

- It has no caching behavior.
- It caches all transactions in the same way.

A9.3.4 Write transactions

WSTRB A master is not required to use the write strobe signals **WSTRB** if it always performs full data bus width write transactions. The default value for write strobes is all signals asserted.

WLAST A slave is not required to use the **WLAST** signal. Since the length of a write burst is defined, a slave can calculate the last write data transfer from the burst length **AWLEN[7:0]** signals.

A9.3.5 Read transactions

RLAST A master is not required to use the **RLAST** signal. Since the length of a read burst is defined, a master can calculate the last read data transfer from the burst length **ARLEN[7:0]** signals.

A9.3.6 Response signaling

RRESP, BRESP A master does not require the **RRESP** and **BRESP** inputs if it both:

- Does not perform exclusive accesses
- Does not require notification of transaction errors

A slave does not require the **RRESP** and **BRESP** outputs if it both:

- Does not support exclusive accesses
- Does not generate error responses

A9.3.7 Non-secure and Secure accesses

AxPROT A slave that is not required to differentiate between Non-secure and Secure accesses, and that does not require any additional protection support, does not require the **AxPROT** input signals.

Caution

Take great care with the **AxPROT** signals. The **AxPROT[1]** signals indicate the Secure or Non-secure nature of the transactions, and incorrect assignment of these bits can lead to incorrect system behavior.

Part B

AMBA AXI4-Lite Interface Specification

Chapter B1
AMBA AXI4-Lite

This chapter defines the AXI4-Lite interface and associated protocol. AXI4-Lite is suitable for simpler control register-style interfaces that do not require the full functionality of AXI4.

This chapter contains the following sections:

B1.1 Definition of AXI4-Lite

This section defines the functionality and signal requirements of AXI4-Lite components.

The key functionality of AXI4-Lite operation is:

- All transactions are of burst length 1.
- All data accesses use the full width of the data bus: AXI4-Lite supports a data bus width of 32-bit or 64-bit.
- All accesses are Non-modifiable, Non-bufferable.
- Exclusive accesses are not supported.

B1.1.1 Signal list

Table B1-1 shows permitted signals on an AXI4-Lite interface. Some signals are optional, See *Default signal values on page A9-110*.

Table B1-1 AXI4-Lite interface signals

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	**AWVALID**	**WVALID**	**BVALID**	**ARVALID**	**RVALID**
ARESETn	**AWREADY**	**WREADY**	**BREADY**	**ARREADY**	**RREADY**
–	**AWADDR**	**WDATA**	**BRESP**	**ARADDR**	**RDATA**
–	**AWPROT**	**WSTRB**	–	**ARPROT**	**RRESP**

AXI4 signals modified in AXI4-Lite

The AXI4-Lite interface does not fully support the following signals:

RRESP, BRESP

The EXOKAY response is not supported on the read data and write response channels.

AXI4 signals not supported in AXI4-Lite

The AXI4-Lite interface does not support the following signals:

AWLEN, ARLEN The burst length is defined to be 1, equivalent to an **AxLEN** value of zero.

AWSIZE, ARSIZE All accesses are defined to be the width of the data bus.

Note

AXI4-Lite requires a fixed data bus width of either 32-bit or 64-bit.

AWBURST, ARBURST

The burst type has no meaning because the burst length is 1.

AWLOCK, ARLOCK

All accesses are defined as Normal accesses, equivalent to an **AxLOCK** value of zero.

AWCACHE, ARCACHE

All accesses are defined as Non-modifiable, Non-bufferable, equivalent to an **AxCACHE** value of `0b0000`.

WLAST, RLAST All bursts are defined to be of length 1, equivalent to a **WLAST** or **RLAST** value of 1.

B1.1.2 Bus width

AXI4-Lite has a fixed data bus width and all transactions are the same width as the data bus. The data bus width must be, either 32-bits or 64-bits.

- The majority of components use a 32-bit interface.
- Only components requiring 64-bit atomic accesses use a 64-bit interface.

A 64-bit component can be designed for access by 32-bit masters, but the implementation must ensure that the component sees all transactions as 64-bit transactions.

Note

This interoperability can be achieved by including, in the register map of the component, locations that are suitable for access by a 32-bit master. Typically, such locations would use only the lower 32 bits of the data bus.

B1.1.3 Write strobes

The AXI4-Lite protocol supports write strobes. This means multi-sized registers can be implemented and also supports memory structures that require support for 8-bit and 16-bit accesses.

All master interfaces and interconnect components must provide correct write strobes. A slave is permitted to:

- To make full use of the write strobes
- To ignore the write strobes and treat all write accesses as being the full data bus width
- To detect write strobe combinations that are not supported and provide an error response

A slave that provides memory access must fully support write strobes. Other slaves in the memory map might support a more limited write strobe option.

When converting from full AXI to AXI4-Lite, a write transaction can be generated on AXI4-Lite with all write strobes deasserted. Automatic suppression of such transactions is permitted but not required. See *Conversion, protection, and detection* on page B1-125.

B1.1.4 Optional signaling

AXI4-Lite supports multiple outstanding transactions, but a slave can restrict this by the appropriate use of the handshake signals.

AXI4-Lite does not support AXI IDs. This means that all transactions must be in order, and all accesses use a single fixed ID value.

Note

Optionally, an AXI4-Lite slave can support AXI ID signals, so that it can be connected to a full AXI interface without modification. See *Interoperability* on page B1-122.

AXI4-Lite does not support data interleaving, the burst length is defined as 1.

B1.2 Interoperability

This section describes the interoperability of AXI and AXI4-Lite masters and slaves. Table B1-2 shows the possible combinations of interface, and indicates that the only case requiring special consideration is an AXI master connecting to an AXI4-Lite slave.

Table B1-2 Full AXI and AXI4-Lite interoperability

Master	Slave	Interoperability
AXI	AXI	Fully operational.
AXI	AXI4-Lite	AXI ID reflection is required. Conversion might be required.
AXI4-Lite	AXI	Fully operational.
AXI4-Lite	AXI4-Lite	Fully operational.

B1.2.1 Bridge requirements of AXI4-Lite slaves

As Table B1-2 shows, the only interoperability case that requires special consideration is the connection of an AXI4-Lite slave interface to a full AXI master interface.

This connection requires AXI ID reflection. The AXI4-Lite slave must return the AXI ID associated with the address of a transaction with the read data or write response for that transaction. This is required because the master requires the returning ID to correctly identify the transaction response.

If an implementation cannot ensure that the AXI master interface only generates transactions in the AXI4-Lite subset, then some form of adaptation is required. See *Conversion, protection, and detection* on page B1-125.

B1.2.2 Direct connection requirements of AXI4-Lite slaves

An AXI4-Lite slave can be designed to include ID reflection logic. This means that the slave can be used directly on a full AXI connection, without a bridge function, in a system that guarantees that the slave is accessed only by transactions that comply with the AXI4-Lite subset.

Note

This specification recommends that the ID reflection logic uses **AWID**, instead of **WID**, to ensure compatibility with both AXI3 and AXI4.

B1.3 Defined conversion mechanism

This section defines the requirements to convert any legal AXI transaction for use on an AXI4-Lite component. *Conversion, protection, and detection* on page B1-125 discusses the advantages and disadvantages of the various approaches that can be used.

B1.3.1 Conversion rules

Conversion requires that the AXI data width is equal to or greater than the AXI4-Lite data width. If not then the AXI data width must first be converted to the AXI4-Lite data width.

Note

AXI4-Lite does not support EXOKAY responses, so the conversion rules do not consider this response.

The rules for conversion from a full AXI interface are as follows:

- If a transaction has a burst length greater than 1, then the burst is broken into multiple transactions of burst length 1. The number of transactions that are created depends on the burst length of the original transaction.
- When generating the address for subsequent beats of a burst, the conversion of bursts with a length greater than 1 must take into consideration the burst type. An unaligned start address must be incremented and aligned for subsequent beats of an INCR or WRAP burst. For a FIXED burst the same address is used for all beats.
- Where a write burst with length greater than 1 is converted into multiple write transactions, the component responsible for the conversion must combine the responses for all of the generated transactions, to produce a single response for the original burst. Any error response is sticky. That is, an error response received for any of the generated transactions is retained, and the single combined response indicates an error. If both a SLVERR and a DECERR are received then the first response received is the one that is used for the combined response.
- A transaction that is wider than the destination AXI4-Lite interface is broken into multiple transactions of the same width as the AXI4-Lite interface. For transactions with an unaligned start address, the breaking up of the burst occurs on boundaries that are aligned to the width of the AXI4-Lite interface.
- Where a wide transaction is converted to multiple narrower transactions, the component responsible for the conversion must combine the responses to all of the narrower transactions, to produce a single response for the original transaction. Any error response is sticky. If both a SLVERR and a DECERR are received then the first response received is used for the combined response.
- Transactions that are narrower than the AXI4-Lite interface are passed directly and are not converted.
- Write strobes are passed directly, unmodified.
- Write transactions with no strobes are passed directly.

  Note

  The AXI4-Lite protocol does not require these transactions to be suppressed.

- The **AxLOCK** signals are discarded for all transactions. For a sequence of locked transactions any lock guarantee is lost. However, the locked nature of the transaction is lost only at any downstream arbitration. For an exclusive sequence, the AXI signaling requirements mean that any exclusive write access must fail.
- The **AxCACHE** signals are discarded. All transactions are treated as Non-modifiable and Non-bufferable.

  Note

  This is acceptable because AXI permits Modifiable accesses to be treated as Non-modifiable, and Bufferable accesses to be treated as Non-bufferable.

- The **AxPROT** signals are passed directly, unmodified.

- The **WLAST** signal is discarded.
- The **RLAST** signal is not required, and is considered asserted for every transfer on the read data channel.

B1.4 Conversion, protection, and detection

Connection of an AXI4-Lite slave to an AXI4 master requires some form of adaptation if it cannot be ensured that the master only issues transactions that meet the AXI4-Lite requirements.

This section describes techniques that can be adopted in a system design to aid with the interoperability of components and the debugging of system design problems. These techniques are:

Conversion This requires the conversion of all transactions to a format that is compatible with the AXI4-Lite requirements.

Protection This requires the detection of any non-compliant transaction. The non-compliant transaction is discarded, and an error response is returned to the master that generated the transaction.

Detection This requires observing any transaction that falls outside the AXI4-Lite requirements and:

- Notifying the controlling software of the unexpected access.
- Permitting the access to proceed at the hardware interface level.

B1.4.1 Conversion and protection levels

Different levels of conversion and protection can be implemented:

Full conversion

This converts all AXI transactions, as described in *Defined conversion mechanism* on page B1-123.

Simple conversion with protection

This propagates transactions that only require a simple conversion, but suppresses and error reports transactions that require a more complex task.

Examples of transactions that are propagated are the discarding of one or more of **AxLOCK** and **AxCACHE**.

Examples of transactions that are discarded and generate an error report are burst length or data width conversions.

Full protection

Suppress and generate an error for every transaction that does not comply with the AXI4-Lite requirements.

B1.4.2 Implementation considerations

A protection mechanism that discards transactions must provide a protocol-compliant error response to prevent deadlock. For example, in the full AXI protocol, read burst transactions require an error for each beat of the burst and a correctly asserted **RLAST** signal.

Using a combination of detection and conversion permits hardware implementations that:

- Do not prevent unexpected accesses from occurring
- Provide a mechanism for notifying the controlling software of the unexpected access, so speeding up the debug process

In complex designs, the advantage of combining conversion and detection is that unforeseen future usage can be supported. For example, at design time it might be considered that only the processor programs the control register of a peripheral, but in practice, the peripheral might need to be programmed by other devices, for example a DSP or a DMA controller, that cannot generate exactly the required AXI4-Lite access.

The advantages and disadvantages of the different approaches are:

- Protection requires a lower gate count.
- Conversion ensures the interface can operate with unforeseen accesses.
- Conversion increases the portability of software from one system to another.
- Conversion might provide more efficient use of the AXI infrastructure. For example, a burst of writes to a FIFO can be issued as a single burst, rather than needing to be issued as a set of single transactions.

- Conversion might provide more efficient use of narrow links, where the address and data payload signals are shared.
- Conversion might provide more flexibility in components that can be placed on AXI4-Lite interfaces. By converting bursts and permitting sparse strobes, memory can be placed on AXI4-Lite, with no burst conversion required in the memory device. This is, essentially, a sharing of the burst conversion logic.

Part C

AMBA AXI5 and AXI5-Lite Interface Specification

Chapter C1
AMBA AXI5

This chapter specifies the new capabilities in the AXI5 protocol specification. It contains the following sections:

C1.1 About the AXI5 protocol

AXI5 extends the capabilities of the AXI4 protocol that is specified in Part A *AMBA AXI Protocol Specification*.To maintain compatibility, a property is used to declare each new capability: Table C1-1 summarizes the AXI5 properties.

Table C1-1 Properties that can be set on an AXI5 interface

Property	Description
Atomic_Transactions	Adds Atomic transactions that perform more than just a single access, and have some form of operation that is associated with them. See *Atomic transactions* on page E1-342.
Check_Type	Adds data checking signaling that is used to detect, and potentially correct, data bytes that might have been corrupted. See Chapter E2 *Interface and data protection*.
Poison	Adds Poison signaling that is used to indicate that a set of data bytes have been previously corrupted. See Chapter E2 *Interface and data protection*.
QoS_Accept	Adds two additional QoS interface signals that enable a slave to indicate the QoS value of transactions that it will accept. See *QoS Accept signaling* on page E1-360.
Trace_Signals	Adds a Trace signal, which is associated with each channel, to support the debugging, tracing, and performance measurement of systems. See *Trace signals* on page E1-357.
Loopback_Signals	Adds loopback signaling that permits an agent that is issuing transactions to store information relating to the transaction in an indexed table. See *User Loopback signaling* on page E1-359.
Wakeup_Signals	Adds wakeup signaling that is used to indicate that there is activity that is associated with the interface. See *Wake-up Signaling* on page E1-362.
Untranslated_Transactions	Adds untranslated transaction support and permits different transactions on the same interface to use different translation schemes. See *Untranslated transactions* on page E1-369.
NSAccess_Identifiers	Adds Non-secure access identifiers that support the storage and processing of protected data. See *Non-secure access identifiers* on page E1-376.
MPAM_Support	Adds indicatation of interface supporting MPAM. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385.
Unique_ID_Support	Adds indicatation of interface supporting the Unique ID Indicator: See *Unique ID indicator* on page E1-383.
Read_Interleaving_Disabled	Adds indicatation of interface supporting the interleaving of read data beats from different transactions. See *Read interleaving property* on page E1-382.
Read_Data_Chunking	Adds indicatation of interface supporting the return of read data in reorderable chunks. See *Read data chunking* on page E1-378.
MTE_Support	Used to indicate that a component supports the Memory Tagging Extension. See *Memory tagging* on page E1-387.
Regular_Transactions_Only	Used to define whether a master issues only Regular type transactions and if a slave only supports Regular transactions. See *Regular transactions property* on page A3-53.

Table C1-1 Properties that can be set on an AXI5 interface (continued)

Property	Description
Exclusive_Accesses	Used to define whether a master issues exclusive accesses or whether a slave supports them. See *Exclusive accesses* on page E1-399.
Max_Transaction_Bytes	Defines the maximum size of a transaction in bytes. See *Maximum transaction size and boundary* on page E1-400.
Consistent_DECERR	Used to define whether a slave signals DECERR consistently across all beats of read and write response. See *Consistent DECERR response* on page E1-401.

C1.2 Signal Descriptions

This section introduces the additional AXI5 interface signals that support the new capabilities. It contains the following subsections:

- *Additions to existing AXI channels*
- *Additional signaling* on page C1-137

See Chapter A8 *AMBA 4 Additional Signaling* for details of the AXI4 interface signals.

C1.2.1 Additions to existing AXI channels

Depending on the interface properties, signals might be added on the following AXI channels:

- *Write address channel signals*
- *Write data channel signals* on page C1-134
- *Write response channel signals* on page C1-135
- *Read address channel signals* on page C1-135
- *Read data channel signals* on page C1-136
- *Parity check signals* on page E2-411

Write address channel signals

Table C1-2 shows the additional write address channel signals:

Table C1-2 Write address channel signals

Signal	Source	Property	Description
AWATOP	Master	Atomic_Transactions	Indicates the type and endianness of atomic transactions. See *Atomic transactions* on page E1-342.
AWTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
AWLOOP	Master	Loopback_Signals	Loopback value for a write transaction. See *User Loopback signaling* on page E1-359.
AWMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSID	Master	Untranslated_Transactions	Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSSIDV	Master	Untranslated_Transactions	Indicates if the AWMMUSSID signal is valid. See *Untranslated transactions* on page E1-369.
AWMMUSSID	Master	Untranslated_Transactions	Substream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUATST	Master	Untranslated_Transactions	Indicates whether a write transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.
AWMMUFLOW	Master	Untranslated_Transactions	Indicates the SMMU flow for managing translation faults. See *Untranslated transactions* on page E1-369.

Table C1-2 Write address channel signals (continued)

Signal	Source	Property	Description
AWNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a write transaction. See *Non-secure access identifiers* on page E1-376.
AWMPAM	Master	MPAM_Support	Write address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
AWIDUNQ	Master	Unique_ID_Support	Write address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
AWTAGOP	Master	MTE_Support	Write request tag operation.See *MTE signaling* on page E1-388

Write data channel signals

Table C1-3 shows the additional write data channel signals:

Table C1-3 Write data channel signals

Signal	Source	Property	Description
WPOISON	Master	Poison	Indicates that the write data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
WTRACE	Master	Trace_signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
WTAG	Master	MTE_Support	Indicates the tag that is associated with write data. See *MTE signaling* on page E1-388.
WTAGUPDATE	Master	MTE_Support	Indicates which tags must be written to memory in an Update operation. See *MTE signaling* on page E1-388

Write response channel signals

Table C1-4 shows the additional write response channel signals:

Table C1-4 Write response channel signals

Signal	Source	Property	Description
BTRACE	Interconnect	Trace_signals	Supports the tracing of specific write transactions through the system. See *Trace signals* on page E1-357.
BLOOP	Interconnect	Loopback_Signals	Loopback value for a write response. See *User Loopback signaling* on page E1-359.
BIDUNQ	Slave	Unique_ID_Support	Write response channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
BTAGMATCH	Slave	MTE_Support	Indicates the result of a tag comparison on a write transaction. See *MTE signaling* on page E1-388.
BCOMP	Slave	CMO_On_Write, Persist_CMO, MTE_Support	Indicates that a write is observable. See *PCMO response on the B channel* on page D7-275.

Read address channel signals

Table C1-5 shows the additional read address channel signals:

Table C1-5 Read address channel signals

Signal	Source	Property	Description
ARTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
ARLOOP	Master	Loopback_Signals	Loopback value for a read transaction. Reflected back on **RLOOP**. See *User Loopback signaling* on page E1-359.
ARMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSID	Master	Untranslated_Transactions	Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSSIDV	Master	Untranslated_Transactions	Indicates whether the ARMMUSSID signal is valid. See *Untranslated transactions* on page E1-369.
ARMMUSSID	Master	Untranslated_Transactions	Substream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUATST	Master	Untranslated_Transactions	Indicates whether a read transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.

Table C1-5 Read address channel signals (continued)

Signal	Source	Property	Description
ARMMUFLOW	Master	Untranslated_Transactions	Indicates the SMMU flow for managing translation faults. See *Untranslated transactions* on page E1-369.
ARNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a read transaction. See *Non-secure access identifiers* on page E1-376.
ARMPAM	Master	MPAM_Support	Read address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
ARIDUNQ	Master	Unique_ID_Support	Read address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
ARCHUNKEN	Master	Read_Data_Chunking	Read data chunking enable. See *Read data chunking* on page E1-378.
ARTAGOP	Master	MTE_Support	Read request tag operation. See *MTE signaling* on page E1-388

Read data channel signals

Table C1-6 shows the additional read data channel signals:

Table C1-6 Read data channel signals

Signal	Source	Property	Description
RPOISON	Interconnect	Poison	Indicates that the read data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
RTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
RLOOP	Interconnect	Loopback_Signals	Loopback value for a read response. See *User Loopback signaling* on page E1-359.
RIDUNQ	Slave	Unique_ID_Support	Read data channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
RCHUNKV	Slave	Read_Data_Chunking	Valid signal of **RCHUNKNUM** and **RCHUNKSTRB**. See *Read data chunking* on page E1-378.
RCHUNKNUM	Slave	Read_Data_Chunking	Read data chunk number. See *Read data chunking* on page E1-378.
RCHUNKSTRB	Slave	Read_Data_Chunking	Read data chunk strobe. See *Read data chunking* on page E1-378.
RTAG	Slave	MTE_Support	The tag that is associated with read data. See *MTE signaling* on page E1-388

C1.2.2 Additional signaling

The following ancillary signaling is optional on the AXI5 interface to support the new capabilities.

QoS accept

Table C1-7 shows the additional QoS accept signaling.

Table C1-7 QoS accept signals

Signal	Source	Property	Description
VAWQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for write transactions. See *QoS Accept signaling* on page E1-360.
VARQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for read transactions. See *QoS Accept signaling* on page E1-360.

Low-power signals

Table C1-8 shows the additional wakeup low-power signaling.

Table C1-8 Wakeup low-power signals

Signal	Source	Property	Description
AWAKEUP	Master	Wakeup_Signals	Indicates that activity is initiated on the write or read address channels. See *Wake-up Signaling* on page E1-362.

Chapter C2
AMBA AXI5-Lite

This chapter specifies the new capabilities in the AXI5-Lite protocol specification. It contains the following sections:

C2.1 Definition of AXI5-Lite

AXI5-Lite is a subset of AXI5, where all transactions are completed in a single beat. It is intended for communication with register-based components and simple memories when bursts of data transfer are not advantageous.

AXI5-Lite extends the definition of AXI4-Lite, adding more flexibility on bus width and ordering. These features enable the interface to be used for peripherals that are closely coupled to high-performance processors when it is important to minimize response latency. For example, an AXI5-Lite master can issue multiple requests to peripherals with different response latencies, without the slower peripherals affecting the latency of faster ones.

Figure C2-1 shows an example where AXI5-Lite might be used.

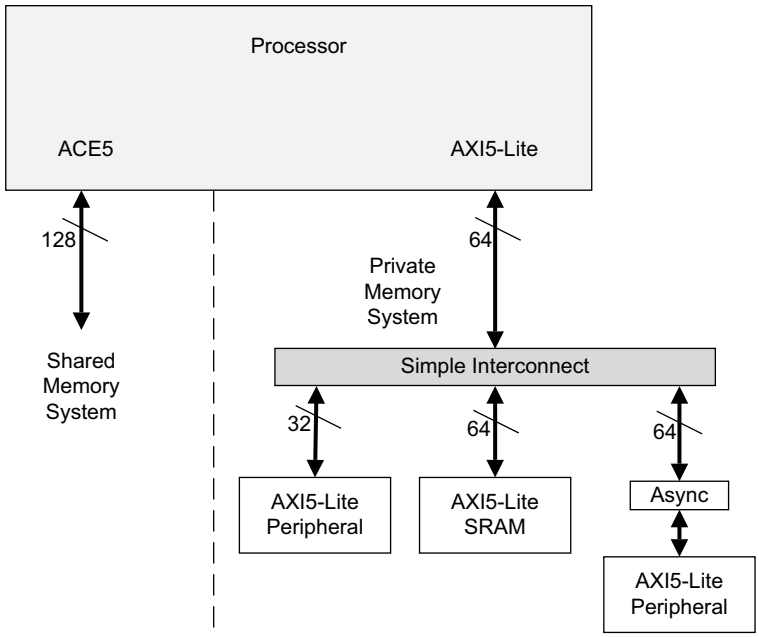


Figure C2-1 Example of memory system that uses AXI5-Lite

The key functionality of AXI5-Lite operation is:

- All transactions have burst length 1.
- Reordering of responses is permitted when requests have different IDs.
- All accesses are considered Device Non-bufferable.
- Exclusive accesses are not supported.

C2.2 AXI5-Lite compared with other interfaces

If a component does not benefit from burst access, AXI5-Lite is a better choice of interface than AXI5. Compared with AXI5, an AXI5-Lite interface is simpler to implement and verify.

Compared with AXI4-Lite, AXI5-Lite permits any data width and responses can be reordered. Flexibility in data width enables an AXI5-Lite slave to be easily connected to the main memory system interconnect. If data width conversion is required, performing it in the AXI5-Lite domain, is less complex than in AXI. Response reordering is optional, but can improve performance when communicating with slaves with differing response latencies.

Table C2-1 shows the summary of differences:

Table C2-1 AXI5-Lite interface comparison

	AXI5	AXI5-Lite	AXI4-Lite
Number of interface wires (32-bit address and data)	224	175	160
Data width (bits)	Up to 1024	Up to 1024	32 or 64
Transaction length	Up to 256	1	1
Transaction size	Up to bus width	Up to bus width	Full bus width
Address buses	Read and write	Read and write	Read and write
Memory types	Any	Device Non-bufferable	Device Non-bufferable
Write strobes	Mandatory	Mandatory	optional
Response Ordering	In-order or out-of-order	In-order or out-of-order	In-order
IDs	Mandatory	Mandatory	optional
Exclusive accesses	Supported	No	No
Check_Type	optional	optional	No
Poison	optional	optional	No
Trace_Signals	optional	optional	No
Wakeup_Signals	optional	optional	No
Unique_ID_Support	optional	optional	No

C2.3 Interoperability

This section describes the interoperability of AXI5-Lite with AXI5 and AXI4-Lite components.

Table C2-2 shows combinations of interface, and indicates that special consideration must be given when an AXI5 master is connected to an AXI5-Lite slave and an AXI5-Lite master is connected to an AXI4-Lite slave.

Table C2-2 Interoperability of AXI5-Lite with AXI5 and AXI4-Lite

Master	Slave	Interoperability
AXI5	AXI5-Lite	Can connect directly if master uses AXI5-Lite subset of transactions. Otherwise needs conversion, protection, or detection. See *Conversion, protection, and detection* on page B1-125
AXI5-Lite	AXI5	Fully operational.
AXI4-Lite	AXI5-Lite	Fully operational.
AXI5-Lite	AXI4-Lite	AXI ID reflection is required on slave. Can connect directly if master uses bus-width transactions. Otherwise needs conversion, protection, or detection. See *Conversion, protection, and detection* on page B1-125

C2.4 Conversion from AXI5 to AXI5-Lite

If an AXI5 master uses transactions that are not within the AXI5-Lite subset, a bridge can be used to convert the AXI5 transactions into those suitable for an AXI5-Lite slave.

The rules for conversion are as follows:

- If a transaction has a burst length greater than 1, then the burst is broken into multiple transactions of burst length 1. The number of transactions that are created depends on the burst length of the original transaction.
- When generating the address for subsequent beats of a burst, the conversion of bursts with a length greater than 1 must consider the burst type. An unaligned start address must be incremented and aligned for subsequent beats of an INCR or WRAP burst. For a FIXED burst, the same address is used for all beats.
- Where a write burst with length greater than 1 is converted into multiple write transactions, the component responsible for the conversion must combine the responses for all the generated transactions to produce a single response for the original burst. Any error response is sticky. That is, an error response that is received for any of the generated transactions is retained, and the single combined response indicates an error. If both a SLVERR and a DECERR are received, then the first response that is received is the one that is used for the combined response.
- A transaction that is wider than the destination AXI5-Lite interface is broken into multiple transactions of the same width as the AXI5-Lite interface. For transactions with an unaligned start address, the breaking up of the burst occurs on boundaries that are aligned to the width of the AXI5-Lite interface.
- Where a wide transaction is converted to multiple narrower transactions, the component responsible for the conversion must combine the responses to all the narrower transactions, to produce a single response for the original transaction. Any error response is sticky. If both an SLVERR and a DECERR are received then the first response received is used for the combined response.
- Transactions that are narrower than the AXI5-Lite interface are passed directly and are not converted.
- Transaction IDs are passed directly, unmodified.
- Write strobes are passed directly, unmodified.
- For an exclusive sequence, the AXI signaling requirements mean that any exclusive write access must fail.
- The **AxCACHE** signals are discarded. All transactions are treated as Non-modifiable and Non-bufferable.
- The **AxPROT** signals are passed directly, unmodified.
- The **WLAST** signal is discarded.
- The **RLAST** signal is not required, and is considered asserted for every transfer on the read data channel.

C2.5 Upgrading an AXI4-Lite master to AXI5-Lite

An AXI4-Lite master can be upgraded to AXI5-Lite by doing the following:

- If not already present, add ID signals. If the master supports only in-order responses, then use a single-bit ID and tie off **ARID** and **AWID** to `0b0`.
- Add **AWSIZE** and **ARSIZE** outputs. An AXI4-Lite master only generates transactions that are full bus width, so these signals can be tied off to `0b010` for a 32-bit bus or `0b011` for a 64-bit bus.

C2.6 Upgrading an AXI4-Lite slave to AXI5-Lite

An AXI4-Lite slave can be upgraded to AXI5-Lite by doing the following:

- If not already present, add ID signals. The slave must mirror **ARID** onto **RID** and **AWID** onto **BID**. Responses can continue to be provided in-order, or out-of-order capability can be added.
- Add the **AWSIZE** input. It can be decided whether to use this, or use **WSTRB** to determine which bytes to write.
- Modify the slave to fully support **WSTRB**, if it does not already. The slave must only write those bytes indicated by the relevant **WSTRB** bits. A write with no strobes asserted must be supported.
- Add the **ARSIZE** input. The slave can choose to use this input to drive only the active bytes in the transfer, or it can continue to drive the full bus width of read data.

C2.7 AXI5-Lite signal list

Table C2-3 lists the signals available on each channel with AXI5-Lite. Some signals are optional or conditional on interface properties, see Table G2-2 on page G2-461.

Table C2-3 AXI5-Lite signals

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	**AWVALID**	**WVALID**	**BVALID**	**ARVALID**	**RVALID**
ARESETn	**AWREADY**	**WREADY**	**BREADY**	**ARREADY**	**RREADY**
AWAKEUP	**AWADDR**	**WDATA**	-	**ARADDR**	**RDATA**
-	**AWPROT**[a]	**WSTRB**	**BRESP**	**ARPROT**[a]	**RRESP**
-	**AWID**	-	**BID**	**ARID**	**RID**
-	**AWIDUNQ**	-	**BIDUNQ**	**ARIDUNQ**	**RIDUNQ**
-	**AWSIZE**	-	-	**ARSIZE**	-
-	**AWUSER**	**WUSER**	**BUSER**	**ARUSER**	**RUSER**
-	**AWTRACE**	**WTRACE**	**BTRACE**	**ARTRACE**	**RTRACE**
-	-	**WPOISON**	-	-	**RPOISON**

a. **AxPROT** is defined as 3 bits wide, but only **AxPROT[1]** (secure) is used by the interface.

Interface parity signals can also be included on an AXI5-Lite interface, see *Parity check signals* on page E2-411.

Part D

AMBA ACE and ACE-Lite Protocol Specification

Chapter D1
About ACE

This chapter gives an overview of system level coherency and the ACE protocol that supports it. It contains the following sections:

D1.1 Coherency overview

System level coherency enables the sharing of memory by system components without the software requirement to perform software cache maintenance to maintain coherency between caches.

Regions of memory are coherent if writes to the same memory location by two components are observable in the same order by all components.

The ACE protocol enables:

- Correctness to be maintained when sharing data across caches
- Components with different characteristics to interact
- The maximum reuse of cached data
- A choice between high performance and low power

The ACE protocol provides a framework for system level coherency. The system designer can determine:

- The ranges of memory that are coherent
- The memory system components that implement the coherency extensions
- The software models that are used to communicate between system components

D1.1.1 ACE revisions

Issue D of the ACE Protocol Specification first described the *AXI Coherency Extensions*.

Issue E of the specification adds clarifications, recommendations, and new capabilities to the ACE Protocol Specification described in Issue D. To maintain compatibility, a property is used to declare a new capability.

Issue F of the *ACE Protocol Specification* describes extensions to the ACE protocol. The protocol now has four variants:

- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

References to the Low-Power Interface have been removed, this content has been superseded by the *AMBA Low Power Interface Specification* (ARM IHI 0068). New appendixes have been added to summarize transaction names and signal lists.

D1.1.2 Usage cases

The ACE protocol enables system architects to select the most appropriate technique for sharing data between system components. The protocol does not define specific usage cases, but typical usage cases are:

- The coherent connection of system components
- The coherent connection of subsystems that have non-uniform memory resources
- The coherent connection of components that have a highly optimized local coherency system
- The filtering of coherency communications
- The coherent connection of components that support different coherency protocols, such as MESI, ESI, MEI, and MOESI
- Wrapping of components that do not support coherency natively, enabling them to be used effectively within a coherent system level design
- Support for cached components that might include multiple levels of cache, and non-cached components
- Support for components that store coherency information at different granularities, including cache line granularity and large buffer granularity
- Implementations that facilitate optimization of:
  - The primary interconnect within a system
  - Multiple subsystems

D1.1.3 ACE terminology

Terminology on page A1-30 introduces terminology that is used throughout the AXI and ACE specifications, and indicates that:

- This specification does not define standard cache terminology, as defined in any reference work on caching.
- The *Glossary* defines terms that are used in the specifications.

ACE introduces additional terms, particularly relating to caching, and to memory operations performed by system masters. The following subsections summarize those terms. Where appropriate, terms that are listed in this section link to the corresponding glossary definition.

AXI components and topology

The following terms describe components in an AXI4 system. Some terms apply, more specifically, to caches on those components:

- *Caching master*, *Initiating master*, and *Snooped master*
- *Downstream cache*, *Local cache*, *Peer cache*, and *Snooped cache*
- *Main memory* and *Snoop filter*

Cache state terminology

Cache state model on page D1-153 defines the possible states of a cache entry.

Actions and permissions

The following terms relate to actions that can be performed by a *Master component*, and the permissions to perform such actions:

- *Load*, *Speculative read*, and *Store*
- *Permission to store* and *Permission to update main memory*

Temporal descriptions

The AXI specification defines *in a timely manner*. The ACE specification requires the additional concept of *At approximately the same time*.

D1.2 Protocol overview

This section introduces the ACE protocol.

D1.2.1 About the ACE protocol

The ACE protocol extends the AXI4 protocol and provides support for hardware-coherent caches. The ACE protocol is realized using:

- A five state cache model to define the state of any cache line in the coherent system. The cache line state determines what actions are required during access to that cache line.
- Additional signaling on the existing AXI4 channels that enable new transactions and information to be conveyed to locations that require hardware coherency support.
- Additional channels that enable communication with a cached master when another master is accessing an address location that might be shared.

The ACE protocol also provides:

- Barrier transactions that guarantee transaction ordering within a system, see *Barriers* on page D1-166. Barrier transactions are not supported in ACE5 and ACE5-Lite variant interfaces.
- *Distributed Virtual Memory* (DVM) functionality to manage virtual memory, see *Distributed Virtual Memory* on page D1-167.

D1.2.2 Coherency model

Figure D1-1 shows an example coherent system that includes three master components, each with a local cache. The ACE protocol permits cached copies of the same memory location to reside in the local cache of one or more master components.

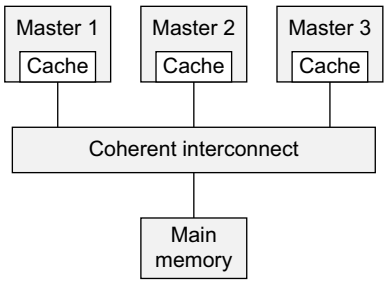


Figure D1-1 Example coherent system

The ACE coherency protocol ensures that all masters observe the correct data value at any given address location by enforcing that only one copy exists whenever a store occurs to the location. After each store to a location, other masters can obtain a new copy of the data for their own local cache, allowing multiple copies to exist.

A cache line is defined as a cached copy of a number of sequentially byte addressed memory locations, with the first address aligned to the total size of the cache line.

There is no requirement to keep main memory up to date at all times. Main memory is only required to be updated before a copy of the memory location is no longer held in any cache.

Note

Although not a requirement, it is acceptable to update main memory while cached copies still exist.

The ACE protocol enables master components to determine if a cache line is the only copy of a particular memory location, or if there might be other copies of the same location, so that:

- If a cache line is the only copy, a master component can change the value of the cache line without notifying any other master components in the system.
- If a cache line might also be present in another cache, a master component must notify the other caches, using an appropriate transaction.

D1.2.3 Cache state model

To determine whether an action is required when a component accesses a cache line, the ACE protocol defines cache states. Each cache state is based on a cache line characteristic.

The cache line characteristics are:

Valid, Invalid When valid, the cache line is present in the cache. When invalid, the cache line is not present in the cache.

Unique, Shared When unique, the cache line exists only in one cache. When shared, the cache line might exist in more than one cache, but this is not guaranteed.

Clean, Dirty When Clean, the cache does not have responsibility for updating main memory. When Dirty, the cache line has been modified with respect to main memory, and this cache must ensure that main memory is eventually updated.

Figure D1-2 shows the ACE five state cache model and Table D1-1 on page D1-154 provides more information about each state:

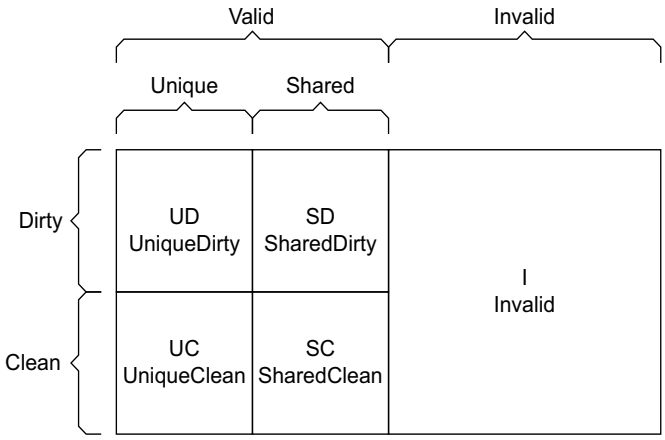


Figure D1-2 ACE cache state model

Table D1-1 ACE cache states

State	Abbreviation	Description
Invalid	I	The cache line does not exist in this cache.
UniqueClean	UC	The following rules apply to a cache line that is in the UniqueClean state: • The cache line is held only in this cache and it has not been modified with respect to main memory. • A component can perform a store to the cache line without notifying other caches.
UniqueDirty	UD	The following rules apply to a cache line that is in the UniqueDirty state: • The cache line is held only in this cache. • The cache line has been modified with respect to main memory and this cache must ensure that the changes are subsequently notified to main memory. • A component can perform subsequent stores to the cache line without notifying other caches.
SharedClean	SC	The following rules apply to a cache line that is in the SharedClean state: • The cache line might be shared with another cache. • It is not known if the cache line is modified with respect to main memory, but this component is not responsible for updating main memory. • A component must notify other caches before performing a store to the cache line.
SharedDirty	SD	The following rules apply to a cache line that is in the SharedDirty state: • The cache line might be shared with another cache. • The cache line has been modified with respect to main memory and this cache must ensure that the changes are subsequently notified to main memory. • A component must notify other caches before performing a store to the cache line.

Cache state rules

The rules that apply to the cache states are:

- A line in a Unique state must only be in one cache.
- A line that is in more than one cache must be in a Shared state in every cache it is in.
- When a cache obtains a new copy of a line, other caches that also have a copy of the line must be notified. This copy might have the line in a Unique state and must be notified to hold the line in a Shared state.
- When a cache discards a copy of a line, there is no requirement to inform other caches that also have a copy of the line. This requirement means that a line in a Shared state might be held in only one cache.
- A line that has been updated, relative to main memory, must be in a Dirty state in one cache.
- A line that has been updated relative to main memory and is in more than one cache, must be in a Dirty state in only one cache.

D1.3 Channel overview

This section introduces the signals that the ACE protocol provides, and where appropriate, describes their relationship to the existing AXI4 channels. The ACE protocol defines:

- Signaling on existing AXI4 channels, see *Changes to existing AXI4 channels*
- Signaling on ACE-specific channels, see *Additional channels defined by ACE*
- Acknowledge signaling, see *Acknowledge signaling* on page D1-156

Channel usage examples on page D1-157 gives examples of the use of the ACE signaling.

D1.3.1 Changes to existing AXI4 channels

Table D1-2 shows the ACE signals provided on existing AXI4 channels.

Table D1-2 ACE signals provided on existing AXI4 channels

AXI4 channel	Signal	Source	Description
Read address	**ARDOMAIN[1:0]**	Master	See *Read address channel (AR) signals* on page D2-170.
	ARSNOOP[3:0]	Master	
	ARBAR[1:0]	Master	
Write address	**AWDOMAIN[1:0]**	Master	See *Write address channel (AW) signals* on page D2-170.
	AWSNOOP[2:0]	Master	
	AWBAR[1:0]	Master	
	AWUNIQUE[a]	Master	
Read data	**RRESP[3:2]**	Interconnect	See *Read data channel (R) signals* on page D2-171.

a. The **AWUNIQUE** signal is only required by a component that supports the WriteEvict transaction.

Note

There are no additional signals on the write data or write response channels.

D1.3.2 Additional channels defined by ACE

Three new channels are supported, these are:

- Snoop address channel
- Snoop data channel
- Snoop response channel

The snoop address (AC) channel is an input to a cached master that provides the address and associated control information for snoop transactions.

The snoop response (CR) channel is an output channel from a cached master that provides a response to a snoop transaction. Every snoop transaction has a single response that is associated with it. The snoop response indicates that an associated data transfer on the CD channel is expected.

The snoop data (CD) channel is an optional output channel that passes snoop data out from a master. Typically, this output occurs for a read or clean snoop transaction when the master being snooped has a copy of the data available to return.

Table D1-3 shows the signals provided on the ACE-specific channels.

Table D1-3 ACE signals provided on ACE-specific channels

ACE-specific channel	Signal	Source	Description
Snoop address	**ACVALID**	Interconnect	See *Snoop address channel (AC) signals* on page D2-172
	ACREADY	Master	
	ACADDR[ac-1:0][a]	Interconnect	
	ACSNOOP[3:0]	Interconnect	
	ACPROT[2:0]	Interconnect	
Snoop response	**CRVALID**	Master	See *Snoop response channel (CR) signals* on page D2-172
	CRREADY	Interconnect	
	CRRESP[4:0]	Master	
Snoop data	**CDVALID**	Master	See *Snoop data channel (CD) signals* on page D2-173
	CDREADY	Interconnect	
	CDDATA[cd-1:0][b]	Master	
	CDLAST	Master	

a. ac is the width of the snoop address bus.
b. cd is the width of the snoop data bus.

D1.3.3 Acknowledge signaling

ACE supports two additional acknowledge signals. These signals indicate that a master has completed a read or write transaction.

Table D1-4 shows the acknowledge signals.

Table D1-4 ACE read and write acknowledge signals

Signal	Source	Description
RACK	Master	See *Read acknowledge signal* on page D2-174
WACK	Master	See *Write acknowledge signal* on page D2-174

D1.3.4 Channel usage examples

This section describes different examples of how the ACE channels are used when performing load and store operations.

Performing load operations from Shareable locations

The following procedure is an example of a master component loading data from a Shareable address location, where the master component does not already have a copy of this location in its local cache:

1. The master component issues a read transaction on the read address channel.
2. The interconnect determines whether any other cache holds a copy of the location by passing the Shareable address to other caching master components that can hold a copy on the snoop address channel. In this context, these are snooped master components.
3. The snooped masters respond on the snoop response channel and optionally provide data on the snoop data channel.
4. If a master has provided data, the interconnect can respond to the initiating master on the read data channel.
5. If no snooped master has provided data:
   a. The interconnect initiates a transaction to main memory, effectively passing on the transaction from the initiating master component.
   b. The read data is supplied back to the master on the read data channel, as for standard transactions.
6. The master component indicates that the transaction has completed, using the **RACK** signal.

If neither the initiating master or the snooped cache takes responsibility for writing a Dirty cache line back to main memory at a later point in time, the interconnect might have to write data back to main memory at the same time that it is passed to the initiating master component.

If this occurs, then the interconnect must generate the transaction address and write the Dirty data that is returned from a snooped master component.

See *Transactions for performing load operations from Shareable locations* on page D1-160 for more information.

Performing store operations to Shareable locations

When a master stores to a cache line, to a Shareable location, it removes all other copies of the cache line. This ensures that the master component has a unique copy of the cache line when it performs the store. The new value of the cache line at that location is propagated to other caches when respective caching master components subsequently read the cache line.

This section describes:

- *Store operations for a partial cache line*
- *Store operations for an entire cache line* on page D1-158
- *Store operations where the cache line is already cached* on page D1-159
- *Overlapping store operations* on page D1-159

See *Transactions for performing store operations to Shareable locations* on page D1-161 and *Transactions for accessing Shareable locations when no cached copy is required* on page D1-161 for more information.

Store operations for a partial cache line

A master component storing only a partial cache line must obtain a current copy of the cache line before performing the store. An example sequence is:

1. The initiating master component obtains a pre-store form of the cache line, and requests that other copies are removed, by issuing a ReadUnique transaction on the read address channel.
2. The interconnect passes the transaction to other caches on the snoop address channel.

3. Where applicable, a snooped master component responds to the transaction using the snoop response channel to indicate that it has the requested cache line. It also provides the cache line to the interconnect, using the snoop data channel.

4. The interconnect passes the cache line, together with a response, to the initiating master, using the read data channel.

   Note

   If no copies of the cache line are found during the snoop, a read of main memory is performed. The interconnect then passes the cache line and a response to the initiating master component, on the read data channel.

5. The master component performs the store and uses the **RACK** signal to indicate that the transaction has completed.

Note

While the cache line remains unique, loads and stores can be performed with no need for transactions to be broadcast to other caches.

Store operations for an entire cache line

A master component that is storing an entire cache line does not have to obtain data before storing the cache line. An example sequence is:

1. The initiating master component requests a unique copy of the cache line by issuing a MakeUnique transaction on the read address channel. This removes all other copies of the cache line.

2. The interconnect passes the transaction to other caches on the snoop address channel.

3. Snooped master components respond to the transaction using the snoop response channel to indicate that the cache line has been successfully removed.

4. A response is provided to the initiating master component, using the read data channel.

   Note

   Only the response fields are valid. No data transfer occurs.

5. The master component performs the store and uses the **RACK** signal to indicate that the transaction has completed.

Store operations where the cache line is already cached

For a master component that already has a shared copy of the cache line, an example store sequence is:

1. The initiating master component requests a unique copy of the cache line by issuing a CleanUnique transaction on the read address channel. This removes all other copies of the cache line and writes any Dirty copy to main memory.

   Note

   This transaction does not return the cache line to the initiating master component.

2. The interconnect passes the transaction to other caches on the snoop address channel. Snooped master components respond to the transaction using the snoop response channel to indicate:
   - That the cache line has been successfully removed
   - Whether a Dirty cache line must be written to main memory by the interconnect
3. If a Dirty cache line is being written to main memory, the appropriate snooped master provides the Dirty cache line to the interconnect, using the snoop data channel. The interconnect then constructs the transaction to write the Dirty cache line back to main memory.
4. A response is provided to the initiating master component, using the read data channel.

   Note

   Only the response fields are valid. No data transfer occurs.

5. The master component performs the store and uses the **RACK** signal to indicate that the transaction has completed.

Overlapping store operations

If two master components attempt simultaneous Shareable store operations to the same cache line, the interconnect determines the order that the transactions occur. This section uses the convention:

- Master1 is the component that the interconnect selects to proceed first.
- Master2 is the component that the interconnect selects to proceed second.

Master 1 proceeds with the operation as described in *Performing store operations to Shareable locations* on page D1-157.

Master 2 uses its snoop port to observe the Master1 store operation, and the following rules apply:

- If Master2 requires data, it receives the data when its own transaction completes, when it can proceed as normal with its own store operations.
- If Master2 is performing a full cache line store, it removes any original copy of the data when it observes the snoop transaction relating to the Master1 store. However, Master2 can then proceed with its own full cache line store when its own transaction completes.
- If Master2 is performing a partial line store, and originally had a copy of the cache line and therefore does not request a copy of the data then special consideration is required. In this case, when Master2 observes the snoop transaction relating to the Master1 store operation, it must remove its original copy of the data. Master 2 can then take one of the following options when its transaction completes:
  - For Master2 to retain the cache line in its cache, it must issue a new transaction to request a copy of the data, enabling it to complete the store operation.
  - Master 2 can perform a partial line write to main memory, ensuring the line is updated correctly, but the master does not retain a copy of the cache line in its cache. To access the cache line at a later point in time, it must fetch the data again.

See *Sequencing transactions* on page D6-249 for more information.

D1.4 Transaction overview

This section introduces the different transaction types. It provides information about when the transactions are used and the required behavior of the various system components. The section describes:

- *Non-snooping transactions*
- *Coherent transactions*
- *Memory update transactions* on page D1-161
- *Cache maintenance transactions* on page D1-162
- *Snoop transactions* on page D1-162
- *Barrier transactions* on page D1-163
- *Distributed virtual memory transactions* on page D1-163

D1.4.1 Non-snooping transactions

Non-snooping transactions are used to access memory locations that are not in the caches of other master components. These transactions do not cause snoop transactions to be performed and are used for the following transaction types:

- Non-shareable.
- Device.

Two forms of non-snooping transaction are provided, ReadNoSnoop and WriteNoSnoop.

Note

Within the context of coherency, ReadNoSnoop and WriteNoSnoop transactions are also referred to as Read and Write transactions. The extended form of the name can be used to differentiate between this transaction type and the more generic set of all read or write transactions.

D1.4.2 Coherent transactions

In general, coherent transactions are used to access Shareable address locations, which might be held in the coherent caches of other components.

Transactions for performing load operations from Shareable locations

When a master is required to perform a load operation from a location in a Shareable area of memory, the following snoop transactions can be used, all of which permit the current holders of the cache line to retain their copy:

ReadClean The ReadClean transaction indicates that the master component requesting the read can only accept a cache line that is Clean, that is, it cannot accept responsibility for a Dirty line that it must subsequently write back to memory. Typically, the ReadClean transaction is used by master components that do not have the ability to accept a Dirty cache line, or have a Write-Through cache.

ReadNotSharedDirty The ReadNotSharedDirty transaction indicates that the master requesting the read can accept a line that is in any state except SharedDirty. This means that the line can be passed as Clean (either unique or shared) or the line can be passed as unique and Dirty.

ReadShared The ReadShared transaction indicates that the master component requesting the read can accept a cache line in any state.

For each of these transactions, it is acceptable for a cache that is being snooped to pass a cache line as Dirty, even if it cannot be accepted by the master component that is requesting the cache line. In this situation, the interconnect is responsible for writing back the Dirty line to main memory.

If a cache that receives one of these snoop transactions has a copy of the data, this specification recommends that it provides the data to complete the snoop transaction. The interconnect must pass the data back to the initiating master component.

If the cache that provides the data originally held the line in a Unique state then to retain the copy, it must move the cache line to a Shared state after the operation.

Transactions for performing store operations to Shareable locations

When a master is required to perform a store to a location in a Shareable area of memory, the following transactions can be used. All the following transactions ensure that there are no other copies of the location when the store operation occurs.

ReadUnique A master component uses the ReadUnique transaction when performing a partial cache line store, storing only some of the bytes of the cache line. The partial store occurs in cases where the master does not already have a copy of the cache line. The ReadUnique transaction obtains a copy of the data and ensures that no other copies exist.

CleanUnique A master component uses a CleanUnique transaction when performing a partial cache line store, in cases where it already has a copy of the cache line. The CleanUnique transaction removes all other copies of the cache line, but if it finds a cache that holds the line in a Dirty state then the transaction ensures that the Dirty cache line is written to main memory.

MakeUnique A master component uses the MakeUnique transaction when performing a full cache line store. The MakeUnique transaction invalidates all other copies of the cache line.

Transactions for accessing Shareable locations when no cached copy is required

When a master is required to access a Shareable memory location but the issuing master is not going to keep a cached copy of the address, either because it does not want to allocate that cache line or because it does not have a cache, the following transactions can be used:

ReadOnce A master component uses the ReadOnce transaction to obtain a snapshot of data that it does not require to copy to its cache. For the ReadOnce transaction, if the cache that provides the data holds the cache line in a Unique state, there is no requirement to change the cache line to a Shared state after the ReadOnce transaction.

WriteUnique

A WriteUnique transaction can be used to remove all copies of a cache line before issuing a write transaction. The WriteUnique transaction can be used when writing a full or partial cache line, and ensures that dirty data is written to memory before performing the write transaction.

WriteLineUnique

A WriteLineUnique transaction can be used to remove all copies of a cache line before issuing a write transaction. The WriteLineUnique transaction must be used only when writing a full cache line, where all bytes within the cache line are written by the transaction.

Note

Unlike other transactions to Shareable memory, ReadOnce and WriteUnique transactions issued by a master component are not required to be a full cache line size. However, WriteLineUnique transactions are required to be a full cache line size.

D1.4.3 Memory update transactions

The following transactions are used to update main memory:

WriteBack A master component cache uses a WriteBack transaction to write back a Dirty line to main memory to free a cache line that is to be used for a different address. The master component does not retain a copy of the cache line.

WriteClean A master component cache uses a WriteClean transaction to write a Dirty line to main memory, while permitting that master component to retain a copy of the cache line.

WriteEvict A WriteEvict transaction can be used to evict a Clean cache line. The transaction is used to write the line to a lower level of the cache hierarchy, such as an L3 or system level cache. The WriteEvict transaction is not required to update main memory.

Evict A master component uses an Evict transaction to indicate the address of a cache line that is evicted from its local cache when no main memory update is required. The transaction enables cache lines in a particular component to be tracked, and can be used for applications such as constructing snoop filters. No data transfer is associated with Evict transactions.

Note

The WriteBack, WriteClean, WriteEvict, and Evict transactions do not result in snoop transactions to other caches. Other caches are not required to know whether the cache line has been written to main memory. WriteBack, WriteClean, WriteEvict, and Evict transactions are not serialized in the same way as other snoop transactions.

D1.4.4 Cache maintenance transactions

Master components use broadcast cache maintenance transactions to access and maintain the caches of other master components in a system. In particular, cache maintenance transactions enable master components to view the effect of load and store operations on system caches that cannot otherwise be accessed. This process is typically referred to as Software Cache Maintenance. Broadcast cache maintenance operations can also propagate to downstream caches, permitting all caches in a system to be maintained.

Note

A master component that initiates a cache maintenance transaction is also responsible for performing the appropriate operation on its own local cache.

The following transactions are used to maintain caches:

CleanShared A master component uses a CleanShared transaction to perform a clean operation on the caches of other components in the system. If a snooped cache that holds a Dirty cache line receives a CleanShared transaction, it must provide that cache line so that the cache line can be written to main memory. The snooped cache can retain its local copy of the cache line.

CleanInvalid A master component uses a CleanInvalid transaction to perform a clean and invalidate operation on the caches of other components in the system. If a snooped cache that holds a Clean cache line receives a CleanInvalid transaction, it must remove its local copy of the cache line. If a snooped cache that holds a Dirty cache line receives a CleanInvalid transaction, it must provide that cache line so that the cache line can be written to main memory and remove its local copy of the cache line.

MakeInvalid A master component uses a MakeInvalid transaction to perform an invalidate operation on the caches of other components in the system. If a snooped cache receives a MakeInvalid transaction, it must remove its local copy, but it is not required to provide any data, even if the cache line is Dirty.

D1.4.5 Snoop transactions

Snoop transactions are transactions that use the snoop address, snoop response, and snoop data channels. Snoop transactions are a subset of coherent transactions and cache maintenance transactions.

D1.4.6 Barrier transactions

Barrier transactions provide guarantees about the ordering and observation of transactions in a system. Barrier transactions are not supported in ACE5 and ACE5-Lite variant interfaces.

ACE supports the following types of barrier:

- Memory barrier
- Synchronization barrier

A master component issues a memory barrier to guarantee that if another master in the appropriate domain can observe any transaction after the barrier it must be able to observe every transaction prior to the barrier.

A master component issues a synchronization barrier to determine when all transactions issued before the barrier are observable by every master in a particular domain. Some synchronization barriers also have a requirement that all transactions that are issued before the barrier transaction must have reached the destination slave component before the barrier completes.

See Chapter D8 *Barrier Transactions* for more information.

D1.4.7 Distributed virtual memory transactions

Distributed Virtual Memory (DVM) transactions are used for virtual memory system maintenance, and typically pass messages between components within distributed virtual memory systems.

See Chapter D13 *Distributed Virtual Memory Transactions* for more information.

D1.5 Transaction processing

When a master component issues a transaction, the coherency signaling indicates that the transaction is to a memory location that is Shareable by more than one component, and therefore requires coherency support. Typically, transactions are processed as follows:

1. The initiating master component issues the transaction.
2. Depending on whether coherency support is required, the transaction is either:
   - Passed directly to a slave component, subject to the address decode scheme being used
   - Passed to the coherency support logic within the interconnect
3. A coherent transaction is checked against subsequent transactions from other master components, to ensure correct processing order.
4. The interconnect determines the snoop transactions that are required.
5. Each cached master that receives a snoop transaction must provide a snoop response. Some masters might provide snoop data to complete the snoop transaction.
6. The interconnect determines whether a main memory access is required.
7. The interconnect collates snoop responses and any required data.
8. The initiating master component completes the transaction.

D1.6 Concepts required for the ACE specification

The ACE specification defines the following concepts:

- *Domains*
- *Barriers* on page D1-166
- *Distributed Virtual Memory* on page D1-167

D1.6.1 Domains

The ACE protocol uses a concept that is called shareability domains. A shareability domain is a set of master components that enables a master component to determine which other master components to include when issuing coherency or barrier transactions.

For coherent transactions, a master component uses the shareability domain to determine which other master components might have a copy of the addressed location in their local cache. The interconnect component uses this information to determine what other master components must be snooped to complete the transaction.

For barrier transactions, a master component uses the shareability domain to determine which other master components the barrier is establishing an ordering relationship with. The domain of a barrier transaction can be used to determine how far a barrier transaction must propagate, and the blocking properties necessary to establish the required ordering.

The ACE protocol defines the following levels of shareability domain:

Non-shareable The domain contains a single master component.

Inner Shareable The domain can include additional master components.

Outer Shareable The domain contains at least all master components in the Inner domain, but can include additional master components.

System The domain includes all master components in the system.

Figure D1-3 shows an example set of shareability domains for a system that includes master components 0-9:

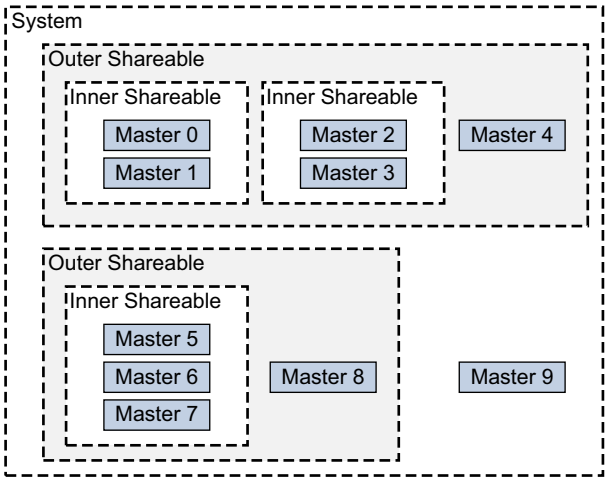


Figure D1-3 Shareability domains

Although multiple Non-shareable, Inner Shareable, and Outer Shareable domains can exist in a system, there must be a single consistent definition of the master components that are contained in each domain. For example, in Figure D1-3, because master 0 has Master1 in its Inner Shareable domain, then Master1 must have master 0 in its Inner Shareable domain.

Domains are defined as non-overlapping. For each master component in an Outer Shareable domain, all the other master components in the Inner Shareable domain that includes that master component must also be included in the same Outer Shareable domain.

For transactions that must be visible to all other master components in the system, the System domain is used. Because System domain transactions include master components that do not have hardware-coherent caches, these transactions must not be cached at any level.

In Figure D1-4, from the perspective of Master1:

- The cache of Master1 is a local cache.
- The caches of masters 2-6 are peer caches.
- Caches of masters 1-3 are in the Inner Shareable domain.
- Caches of masters 1-6 are in the Outer Shareable domain.
- Cache 1 is a downstream cache.

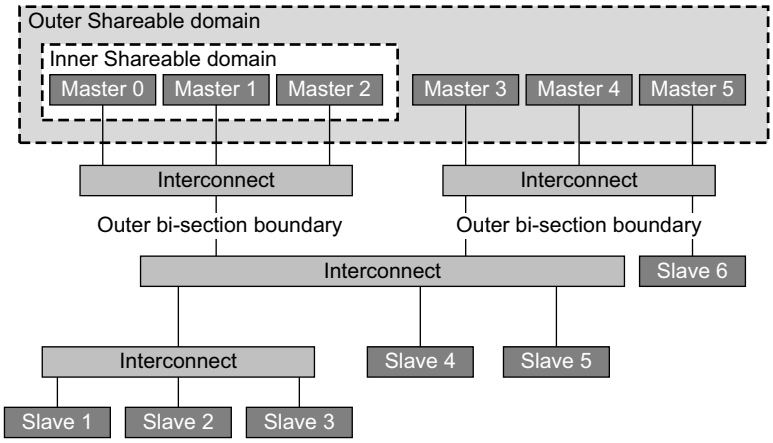


Figure D1-4 Example system using shareability domains

D1.6.2 Barriers

Barrier transactions provide guarantees about the ordering and observation of transactions in a system. The following types of barrier transaction are supported:

Memory barriers

A master component issues a memory barrier to guarantee that if another master component in the appropriate domain can observe any transaction after the barrier it must be able to observe every transaction prior to the barrier.

Synchronization barriers

A master component issues a synchronization barrier to determine when every master component in the appropriate domain can observe all transactions that preceded the barrier transaction. For System domain synchronization barriers, all transactions that are issued before the barrier transaction must have reached the destination slave components before the barrier transaction completes.

A barrier transaction has an address phase and a response, but no data transfer occurs. A master component must issue a barrier transaction on both the read address channel and the write address channel.

Barriers can enforce ordering because a master component must not issue any read or write transaction that must be ordered after the barrier until the master component has received a response for the barrier on both read data and write response channels.

See Chapter D8 *Barrier Transactions* for more information.

D1.6.3 Distributed Virtual Memory

ACE supports *Distributed Virtual Memory* (DVM) and includes transactions that permit virtual memory system management. Figure D1-5 shows the basic parts of a virtual memory system.

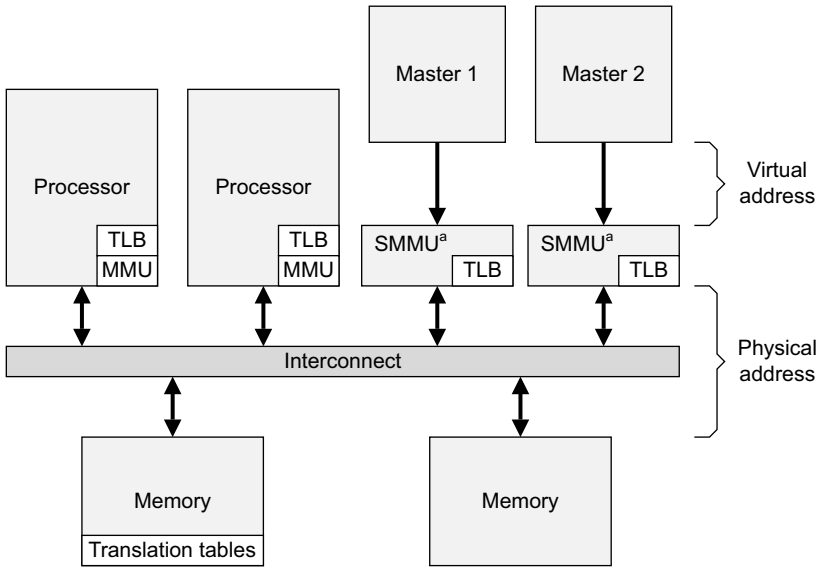


Figure D1-5 Virtual memory system

In Figure D1-5, the *System Memory Management Units* (SMMUs) translate addresses in the virtual address space to addresses in the physical address space. Although all components in the system must use a single physical address space, SMMU components enable different master components to operate in their own independent virtual address or intermediate physical address space.

A typical process in the virtual memory system that is shown in Figure D1-5 might operate as follows:

1. A master component operating in a *virtual address* (VA) space issues a transaction that uses a VA.
2. The SMMU receives the VA, for translation to a *physical address* (PA):
   - If the SMMU has recently performed the requested translation, then it might obtain a cached copy of the translation from its TLB.
   - Otherwise, the SMMU must perform a *translation table walk*, accessing translation table in memory to obtain the required VA to PA translation.
3. The SMMU uses the PA to issue the transaction for the requesting component.

At step 2 of this process, the translation for the required VA might not exist. In this case, the translation table walk generates a fault, that must be notified to the agent that maintains the translation tables. For the required access to proceed, that agent must then provide the required VA to PA translation. Typically, it will update the translation tables with the required information.

Maintaining the translation tables can require changes to translation table entries that are cached in TLBs. To prevent the use of these entries, a DVM message can be used to issue a TLB invalidate operation.

When the translation tables have been updated, and the required TLB invalidations performed, a DVM Sync transaction is used to ensure that all required transactions have completed.

See Chapter D13 *Distributed Virtual Memory Transactions* for more information.

D1.7 Protocol errors

The protocol defines two categories of protocol errors, a software protocol error and a hardware protocol error.

D1.7.1 Software protocol error

A software protocol error occurs when multiple accesses to the same location are made with mismatched shareability or cacheability attributes.

A software protocol error can cause a loss of coherency and result in the corruption of data values. The protocol requires that the system does not deadlock for a software protocol error, and that transactions always progress through a system.

A software protocol error for an access in one 4KB memory region must not cause data corruption in a different 4KB memory region.

For locations held in Normal memory, the use of appropriate barriers and software cache maintenance can be used to return memory locations to a defined state.

When accessing a peripheral device, if Modifiable transactions are used as indicated by **AxCACHE[1]** = 1, then the correct operation of the peripheral cannot be guaranteed. The only requirement is that the peripheral continues to respond to transactions in a protocol-compliant manner. The sequence of events that might be needed to return a peripheral device, that has been accessed incorrectly, to a known working state is IMPLEMENTATION DEFINED.

D1.7.2 Hardware protocol error

A hardware protocol error is defined as any protocol error that is not a software protocol error. No support is required for hardware protocol errors.

If a hardware protocol error occurs, then recovery from the error is not guaranteed. The system might crash, lock up, or suffer some other non-recoverable failure.

Chapter D2
Signal Descriptions

This chapter introduces the additional ACE interface signals. It contains the following sections:

Later chapters define the signal parameters and usage.

D2.1 Changes to existing AXI channels

The following subsections introduce the additional signals that are defined on the AXI channels:

- *Read address channel (AR) signals*
- *Write address channel (AW) signals*
- *Read data channel (R) signals* on page D2-171

Note

There are no additional signals on the write data channel (W) or the Write response channel (B).

See Chapter A2 *Signal Descriptions* for the remaining signals on the AXI channels.

D2.1.1 Read address channel (AR) signals

Table D2-1 shows the additional signals on the read address channel. See *Read and write address channel signaling* on page D3-176.

Table D2-1 Read address channel signals

Signal	Source	Description
ARSNOOP[3:0]	Master	Indicates the transaction type for Shareable read transactions
ARDOMAIN[1:0]	Master	Indicates the shareability domain of a read transaction
ARBAR[1:0]	Master	Indicates whether a transaction is a read barrier

D2.1.2 Write address channel (AW) signals

Table D2-2 shows the additional signals on the write address channel. See *Read and write address channel signaling* on page D3-176.

Table D2-2 Write address channel signals

Signal	Source	Description
AWSNOOP[2:0]	Master	Indicates the transaction type for Shareable write transactions
AWDOMAIN[1:0]	Master	Indicates the shareability domain of a write transaction
AWBAR[1:0]	Master	Indicates whether a transaction is a write barrier
AWUNIQUE[a]	Master	Indicates that the data in this transaction is permitted to be held in a Unique cache state

a. The **AWUNIQUE** signal is only required by a component that supports the WriteEvict transaction.

D2.1.3 Read data channel (R) signals

Table D2-3 shows the additional signals on the read data channel. See *Read data channel signaling* on page D3-186:

Table D2-3 Read data channel signals

Signal	Source	Description
RRESP[3:2]	Interconnect	Read response, indicates the status of a read transfer

D2.2 Additional channels defined by ACE

The following subsections introduce the ACE snoop channels:

- *Snoop address channel (AC) signals*
- *Snoop response channel (CR) signals*
- *Snoop data channel (CD) signals* on page D2-173

D2.2.1 Snoop address channel (AC) signals

Table D2-4 shows the signals on the snoop address channel. See *Snoop address channel signaling* on page D3-192.

Table D2-4 Snoop address channel signals

Signal	Source	Description
ACVALID	Interconnect	Indicates that the snoop address channel signals are valid
ACREADY	Master	Indicates that a transfer on the snoop address channel can be accepted
ACADDR[ac-1:0][a]	Interconnect	The address of the first transfer in a snoop transaction
ACSNOOP[3:0]	Interconnect	Snoop transaction type
ACPROT[2:0][b]	Interconnect	Protection attributes of a snoop transaction

a. ac is the width of the snoop address bus.
b. The ACE specification only assigns meaning to **ACPROT[1]**.

D2.2.2 Snoop response channel (CR) signals

Table D2-5 shows the signals on the snoop response channel. See *Snoop response channel signaling* on page D3-195.

Table D2-5 Snoop response channel signals

Signal	Source	Description
CRVALID	Master	Indicates that the snoop response channel signals are valid
CRREADY	Interconnect	Indicates that a transfer on the snoop response channel can be accepted
CRRESP[4:0]	Master	Snoop response, indicates the status of a snoop transfer

D2.2.3 Snoop data channel (CD) signals

Table D2-6 shows the signals on the snoop data channel. See *Snoop data channel signaling* on page D3-199.

Table D2-6 Snoop data channel signals

Signal	Source	Description
CDVALID	Master	Indicates that the snoop data channel signals are valid
CDREADY	Interconnect	This signal indicates that the snoop data can be accepted in the current cycle
CDDATA[cd-1:0][a]	Master	Snoop data
CDLAST	Master	Indicates whether this is the last data transfer in a snoop transaction

a. cd is the width of the snoop data bus.

D2.3 Additional response signals and signaling requirements defined by ACE

The following subsections introduce the additional response signals, and an additional signaling requirement, introduced by ACE:

- *Read acknowledge signal*
- *Write acknowledge signal*
- *Reset requirements*

D2.3.1 Read acknowledge signal

Table D2-7 shows the additional read acknowledge signal. This signal indicates that a master has completed a read transaction. See *Read acknowledge signaling* on page D3-189.

Table D2-7 Read acknowledge signal

Signal	Source	Description
RACK	Master	Read acknowledge signal

D2.3.2 Write acknowledge signal

Table D2-8 shows the additional write acknowledge signal. This signal indicates that a master has completed a write transaction. See *Write Acknowledge signaling* on page D3-191.

Table D2-8 Write acknowledge signal

Signal	Source	Description
WACK	Master	Write acknowledge signal

D2.3.3 Reset requirements

The ACE protocol uses the AXI single active LOW reset signal **ARESETn**. See *Reset* on page A3-40 for the **ARESETn** requirements.

During reset, the following interface requirements apply:

- A master interface must drive **RACK**, **WACK**, **CRVALID**, and **CDVALID** LOW.
- An interconnect must drive **ACVALID** LOW.

The earliest point after reset that the interconnect is permitted to begin driving **ACVALID** HIGH is at a rising **ACLK** edge, after **ARESETn** is HIGH.

Chapter D3
Channel Signaling

This chapter describes the basic channel signaling requirements on an ACE interface. It contains the following sections:

D3.1 Read and write address channel signaling

The following sections define the additional signals on the read and write address channels.

D3.1.1 Shareability domain types

The ACE protocol uses a concept that is called shareability domains. A shareability domain is a set of masters that enables a master to determine which other masters to include when issuing coherency or barrier transactions. See *Domains* on page D1-165.

Each address channel has its own shareability domain signal. Table D3-1 shows the signal for each address channel:

Table D3-1 Shareability domain signals

Transaction Channel	Signal	Source	Description
Read address channel	**ARDOMAIN[1:0]**	Master	Indicates shareability domain of a read transaction
Write address channel	**AWDOMAIN[1:0]**	Master	Indicates shareability domain of a write transaction

The ACE protocol specifies four levels of shareability domain:

Non-shareable The domain contains a single master.

Inner Shareable The Inner domain can include additional masters.

Outer Shareable The Outer domain contains all masters in the Inner domain and can include additional masters.

System The System domain includes all masters in the system.

Use of the Inner Shareable domain is deprecated in ACE5 and ACE5-Lite. See *Shareability domain support* on page F5-452 for more details.

Note

Although multiple Non-shareable, Inner Shareable and Outer Shareable domains can exist in a system, there must be a single consistent definition of the masters that are contained in each domain. Figure D1-3 on page D1-165 shows an example set of shareability domains.

In this specification, **AxDOMAIN** indicates **ARDOMAIN** or **AWDOMAIN**.

Table D3-2 shows the **AxDOMAIN[1:0]** signal encoding.

Table D3-2 Shareability domain encoding

AxDOMAIN[1:0]	Domain
`0b00`	Non-shareable
`0b01`	Inner Shareable
`0b10`	Outer Shareable
`0b11`	System

Restrictions apply to the shareability domain for transactions with different memory types:

- A *Device* transaction, indicated by **AxCACHE[1]** equal to zero, must only use domain level System.
- A *Cacheable* transaction, indicated by **AxCACHE[3:2]** not equal to zero, must not use domain level System.

Table D3-3 shows all **AxCACHE** and **AxDOMAIN** combinations. See *AXI4 changes to memory attribute signaling* on page A4-64 for details of the **AxCACHE** encodings.

Table D3-3 AxCACHE and AxDOMAIN signal combinations

AxCACHE		AxDOMAIN		Legal	Caches accessed[a]
Value	**Attribute**	**Value**	**Attribute**		
0b000x	Device	0b00	Non-shareable	No	-
		0b01	Inner Shareable	No	-
		0b10	Outer Shareable	No	-
		0b11	System	Yes	No caches
0b001x	Non-cacheable	0b00	Non-shareable	Permitted	No caches
		0b01	Inner Shareable	Permitted	Inner Shareable peer caches
		0b10	Outer Shareable	Permitted	Outer Shareable peer caches
		0b11	System	Yes	No caches
0b011x 0b101x 0b111x	WriteThrough WriteBack	0b00	Non-shareable	Yes	Downstream caches
		0b01	Inner Shareable	Yes	Inner Shareable peer caches and downstream caches
		0b10	Outer Shareable	Yes	Outer Shareable peer caches and downstream caches
		0b11	System	No	-

a. Shows which caches must be accessed to complete the transaction.

Note

- Table D3-3 does not include any access that is made to a local cache within the initiating master.
- The three combinations of **AxCACHE** and **AxDOMAIN** that are indicated as *Permitted* are legal within the protocol, but not expected. These combinations can be used when a memory location can be cached at a domain level that requires snooping, but the transaction is deliberately not cached downstream, for example, in a system level cache.
- For transactions where **AxCACHE** indicates Non-cacheable and **AxDOMAIN** indicates Inner Shareable or Outer Shareable it is not required that the data is fetched from the final destination.
- When Table D3-3 shows that the caches that are accessed are Outer Shareable peer caches, this includes all caches that are Inner Shareable peer caches.
- A memory location that is indicated as being in the System domain cannot be held in any cache.

D3.1.2 Read and write barrier transactions

Each address channel has its own barrier transaction signal. Table D3-4 shows the signal for each address channel:

Table D3-4 Barrier transaction signals

Transaction channel	Signal	Source	Description
Read address channel	**ARBAR[1:0]**	Master	Indicates whether a transaction is a read barrier.
Write address channel	**AWBAR[1:0]**	Master	Indicates whether a transaction is a write barrier.

See *Barrier transaction signaling* on page D8-285.

In this specification, **AxBAR** indicates **ARBAR** or **AWBAR**. Table D3-5 shows the **AxBAR[1:0]** signal encoding:

Table D3-5 Barrier transaction signal encoding

AxBAR[1:0]	Barrier type
`0b00`	Normal access, respecting barriers
`0b01`	Memory barrier
`0b10`	Normal access, ignoring barriers
`0b11`	Synchronization barrier

D3.1.3 Read and write Shareable transaction types

Each address channel has its own transaction type signal. Table D3-6 shows the signal for each address channel:

Table D3-6 Shareable transaction type signals

Transaction channel	Signal	Source	Description
Read address channel	**ARSNOOP[3:0]**	Master	Indicates the transaction type for Shareable read transactions
Write address channel	**AWSNOOP[2:0]**	Master	Indicates the transaction type for Shareable write transactions

Transactions on the read and write address channel are categorized into the following groups:

Non-snooping These transactions must not snoop other masters.

Coherent These transactions are to memory locations that can be held in the cache of other masters and require snooping.

Memory update These transactions update main memory and must not snoop other masters.

Cache maintenance These transactions are to memory locations that can be held in the cache of other masters and require snooping. These transactions might also require to be passed to downstream caches.

Barrier These transactions establish the ordering between other transactions. See Chapter D8 *Barrier Transactions*.

DVM These transactions pass operations between components that participate in a distributed virtual memory scheme. See Chapter D13 *Distributed Virtual Memory Transactions*.

Table D3-7 shows the permitted combinations of **ARBAR[0]**, **ARDOMAIN[1:0]**, and **ARSNOOP[3:0]** for each group of read transactions.

All unused **ARSNOOP[3:0]** encodings are Reserved.

Table D3-7 Permitted read address control signal combinations

Transaction group	ARBAR[0]	ARDOMAIN	ARSNOOP	Transaction type
Non-snooping	0b0	0b00 0b11	0b0000	ReadNoSnoop
Coherent	0b0	0b01 0b10	0b0000	ReadOnce
			0b0001	ReadShared
			0b0010	ReadClean
			0b0011	ReadNotSharedDirty
			0b0111	ReadUnique
			0b1011	CleanUnique
			0b1100	MakeUnique
Cache maintenance	0b0	0b00 0b01 0b10	0b1000	CleanShared
			0b1001	CleanInvalid
			0b1101	MakeInvalid
Barrier	0b1	0b00 0b01 0b10 0b11	0b0000	Barrier
DVM	0b0	0b01 0b10	0b1110	DVM Complete
			0b1111	DVM Message

Note

A component without a cache only needs to indicate the shareability of a read transaction using **ARDOMAIN** and can tie **ARSNOOP** to zero. As Table D3-7 shows, if the transaction is Non-shareable it is treated as a ReadNoSnoop transaction and if the transaction is Shareable it is treated as a ReadOnce transaction.

Table D3-8 shows the permitted combinations of **AWBAR[0]**, **AWDOMAIN[1:0]**, and **AWSNOOP[2:0]** for each group of write transactions.

All unused **AWSNOOP[2:0]** encodings are Reserved.

Table D3-8 Permitted write address control signal combinations

Transaction group	AWBAR[0]	AWDOMAIN	AWSNOOP	Transaction type
Non-snooping	0b0	0b00 0b11	0b000	WriteNoSnoop
Coherent	0b0	0b01 0b10	0b000	WriteUnique
			0b001	WriteLineUnique
Memory update	0b0	0b00 0b01 0b10	0b010	WriteClean
			0b011	WriteBack
		0b01 0b10	0b100	Evict
		0b00 0b01 0b10	0b101	WriteEvict[a]
Barrier	0b1	0b00 0b01 0b10 0b11	0b000	Barrier

a. A component that supports the WriteEvict transaction must provide the **AWUNIQUE** signal.

Note

A component without a cache only needs to indicate the shareability of a write transaction using **AWDOMAIN** and can tie **AWSNOOP** to zero. As Table D3-8 shows, if the transaction is Non-shareable it is treated as a WriteNoSnoop transaction and if the transaction is Shareable it is treated as a WriteUnique transaction.

D3.1.4 AWUNIQUE signal

The **AWUNIQUE** signal is a write address channel signal that can be used with various write transactions to improve the operation of lower levels of the cache hierarchy, such as an L3 or system-level cache. Table D3-9 shows the **AWUNIQUE** signaling requirements for different write transactions:

Table D3-9 AWUNIQUE signaling requirements for different write transactions

Transaction type	AWUNIQUE requirement	Notes
WriteNoSnoop	Has no meaning. Can be asserted or deasserted.	-
WriteUnique	Can be asserted if the master is not keeping a copy of the cache line. Must be deasserted if the master issuing the transaction is keeping a copy of the cache line.	-
WriteLineUnique	Can be asserted if the master is not keeping a copy of the cache line. Must be deasserted if the master issuing the transaction is keeping a copy of the cache line.	-
WriteClean	Must be deasserted.	The cache line cannot be held in a Unique state as the master issuing the transaction is keeping a copy.
WriteBack	Can be asserted if the cache line was held in a Unique state. Must be deasserted if the cache line was in a Shared state.	It is permitted to deassert the signal alongside a WriteBack transaction even if the cache line was held in a Unique state.
WriteEvict	Must be asserted.	A WriteEvict transaction is only permitted for a cache line in a UniqueClean state and therefore the cache line must have been in a Unique state.
Evict	Has no meaning. Can be asserted or deasserted.	-
Barrier	Has no meaning. Can be asserted or deasserted.	-

While a WriteBack or WriteEvict transaction is in progress that has the **AWUNIQUE** signal asserted, the master must not give a snoop response that would allow another copy of the cache line to be created, or an agent to consider that it has another Unique copy of the cache line.

It is a requirement that a master that supports the WriteEvict transaction, supports the **AWUNIQUE** signal.

A master that implements the **AWUNIQUE** signal can be connected to an interconnect that does not implement the signal. There is no loss in functionality.

A master that does not support the **AWUNIQUE** signal can be connected to an interconnect that does support the signal. In this case, the input to the interconnect must be tied low. This is protocol-compliant as all transactions are permitted to drive **AWUNIQUE** LOW, with the exception of a WriteEvict transaction.

D3.1.5 Cache line size restrictions

The cache line size that each ACE master can support is determined at design time.

Restrictions apply to the minimum and maximum cache line sizes that can be supported.

The minimum cache line size is 16 bytes.

The maximum cache line size is the smaller of:

- 2048 bytes
- The product of the maximum burst length of 16 and the width of the data bus in bytes

D3.1.6 Transaction constraints

The following sections define the constraints for:

- *Cache line size transactions*
- *ReadOnce and WriteUnique transactions* on page D3-184
- *WriteBack and WriteClean transactions* on page D3-184
- *Barrier transactions* on page D3-185

Cache line size transactions

The following transactions must be of cache line size:

- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique
- CleanUnique
- MakeUnique
- CleanShared
- CleanInvalid
- MakeInvalid
- WriteLineUnique
- WriteEvict
- Evict

Cache line size transactions must comply with the constraints for Regular transactions. See *Regular transactions* on page A3-53. Table D3-10 shows the full list of constraints that apply to cache line size transactions.

Table D3-10 Cache line size transaction constraints

Attribute	Condition	Constraint
AxLEN	-	The burst length must be 1, 2, 4, 8 or 16 transfers. See *Burst length* on page A3-48.
AxSIZE	-	If the burst length is greater than 1, the number of bytes in a transfer must be equal to the data bus width. See *Burst size* on page A3-49.
AxBURST	INCR	The address must be aligned to the cache line size, which is equal to **(AxLEN** x **AxSIZE)**, the total burst length in bytes. See *Burst type* on page A3-49.
	WRAP	The address must be aligned to **AxSIZE**, which is equal to the data bus width.
	FIXED	Not supported.

Table D3-10 Cache line size transaction constraints (continued)

Attribute	Condition	Constraint
AxDOMAIN	All transactions except: CleanShared CleanInvalid MakeInvalid or WriteEvict	The domain must be Inner Shareable or Outer Shareable.
	CleanShared CleanInvalid MakeInvalid and WriteEvict	The domain must be Non-shareable, Inner Shareable or Outer Shareable.
AxBAR	-	Must be a normal access.
AxCACHE	-	Must be Modifiable.
AxLOCK	-	Must be: `0b0` If the transaction is ReadNotSharedDirty, ReadUnique, MakeUnique, CleanShared, CleanInvalid, MakeInvalid, WriteLineUnique, WriteEvict or Evict. `0b0` or `0b1` If the transaction is ReadClean, ReadShared, or CleanUnique.
AxPROT	-	No constraint, can take any value.
AxQOS	-	No constraint, can take any value.

WriteLineUnique and WriteEvict transactions are required to have every write data strobe asserted, that is, sparse write data strobes are not permitted.

The following transactions must use **AxLEN** to indicate the correct cache line size, even though these transactions do not transfer data:

- CleanUnique
- MakeUnique
- CleanShared
- CleanInvalid
- MakeInvalid
- Evict

ReadOnce and WriteUnique transactions

The ReadOnce and WriteUnique transactions are not constrained to cache line size. This permits legacy components to operate in a coherent environment by adding an appropriate domain to Modifiable transactions.

Table D3-11 shows the constraints that apply to ReadOnce and WriteUnique transactions.

Table D3-11 ReadOnce and WriteUnique transaction constraints

Attribute	Constraint
AxDOMAIN	Must be Inner Shareable or Outer Shareable
AxBURST	Must be INCR or WRAP
AxCACHE	Must be Modifiable
AxLOCK	Must be normal access
AxPROT	No constraint, can take any value
AxQOS	No constraint, can take any value

WriteUnique transactions are not required to have every write data strobe asserted, that is, sparse write data strobes are permitted.

Note

The FIXED burst type is not supported for ReadOnce and WriteUnique transactions. Any conversion from AXI to ACE-Lite must provide a translation for a FIXED burst.

WriteBack and WriteClean transactions

The WriteBack and WriteClean transactions are not constrained to cache line size. A partial update of a cache line is permitted. However, WriteBack and WriteClean transactions are constrained to updates within a single cache line.

Table D3-12 shows the constraints that apply to WriteBack and WriteClean transactions.

Table D3-12 WriteBack and WriteClean transaction constraints

Attribute	Condition	Constraint
AWLEN	-	Normal restrictions apply. See *Address structure* on page A3-48.
AWSIZE	-	Normal restrictions apply. See *Address structure* on page A3-48.
AWBURST	WRAP	The address must be aligned to **AxSIZE**, which is equal to the data bus width. The burst length must be 2, 4, 8 or 16. **AWSIZE** x **AWLEN** must not exceed the cache line size in bytes.
	INCR	The burst length must be 16 or less. The transaction must not cross a cache line boundary. The location of the last byte in the burst is determined by (**AWSIZE** × **AWLEN**) added to the **AWSIZE** aligned start address. The location of this last byte must fall within the same cache line as the first byte in the burst.
	FIXED	Not supported.
AWDOMAIN	-	Must not be domain type System.
AWBAR	-	Must be normal access.

Table D3-12 WriteBack and WriteClean transaction constraints (continued)

Attribute	Condition	Constraint
AWCACHE	-	Must be Modifiable.
AWLOCK	-	Must be normal access.
AWPROT	-	No constraint, can take any value.
AWQOS	-	No constraint, can take any value.

The WriteBack and WriteClean transactions are permitted to use sparse write data strobes.

Components that support a snoop filter must correctly indicate the shareability domain for all WriteBack, WriteClean, and Evict transactions. This enables a snoop filter to track the allocation of Inner Shareable and Outer Shareable transactions.

A snoop filter must not track the allocation of Non-shareable transactions because notification of the eviction of the associated cache line is not required.

Components that do not support a snoop filter can use any of the following domains for WriteBack or WriteClean transactions:

- Non-shareable
- Inner Shareable
- Outer Shareable

Barrier transactions

For a barrier transaction, as indicated by **AxBAR[0]** equal to 1, constraints apply to the read address and write address signals. Table D3-13 shows the constraints that apply:

Table D3-13 Barrier transaction constraints

Attribute	Constraint
AxADDR	Must be all zeros
AxBURST	Must be burst type INCR
AxLEN	Must be all zeros
AxSIZE	Must be equal to the data bus width
AxCACHE	Must be Normal, Non-cacheable
AxPROT	No constraint, can take any value
AxLOCK	Must be normal access
AxSNOOP	Must be all zeros

Note

A barrier transaction can have any shareability domain. The choice of domain is used to determine the precise behavior of the barrier with respect to other masters in the system. See Chapter D8 *Barrier Transactions*.

D3.2 Read data channel signaling

The following sections define the additional response and acknowledge signaling and constraints on the read data channel. See *Read and write response structure* on page A3-59 for information on the baseline read response signaling.

D3.2.1 Read response signaling

Table D3-14 shows the additional read response signals:

Table D3-14 Additional RRESP read response bits

Signal	Source	Name	Meaning	
RRESP[2]	Interconnect	PassDirty	HIGH	The cache line is Dirty with respect to main memory and the initiating master must ensure that the cache line is written back to main memory, at some time. The initiating master must either perform the write, or pass the responsibility to perform the write to another master.
			LOW	It is not the responsibility of the initiating master to ensure that the cache line is written back to main memory.
RRESP[3]	Interconnect	IsShared	HIGH	Another copy of the associated data might be held in another cache and the cache line must be held in a Shared state.
			LOW	It is the only cached copy of the associated data and the cache line can be held in a Unique state.

The IsShared and PassDirty responses have the following restrictions:

- The IsShared and PassDirty responses must be constant for all data transfers within a burst.
- The IsShared response must be LOW for the transactions that require all other cached copies to be removed. The transactions that require all other cached copies to be removed are:
  - ReadUnique
  - CleanUnique
  - MakeUnique
  - CleanInvalid
  - MakeInvalid
- The PassDirty response must be LOW for the transactions that do not permit the passing of Dirty data. The transactions that do not permit the passing of Dirty data are:
  - ReadOnce
  - ReadClean
  - CleanUnique
  - MakeUnique
  - CleanShared
  - CleanInvalid
  - MakeInvalid

- The IsShared and PassDirty response must be LOW for the following transactions for because they have no meaning for those responses:
  - ReadNoSnoop
  - Barrier transactions
  - DVM transactions

The value of **RRESP[3:2]** must be the same for all data transfers in a burst.

The following transactions have a single read data channel transfer:
- CleanUnique
- MakeUnique
- CleanShared
- CleanInvalid
- MakeInvalid
- Barrier
- DVM

These transactions complete in a single read data channel transfer and must have **RLAST** asserted. **RDATA** can have any value and must be ignored.

The EXOKAY response is only permitted for a ReadNoSnoop, ReadClean, ReadShared or CleanUnique transaction. See *Read and write response structure* on page A3-59.

Table D3-15 shows the permitted IsShared and PassDirty responses for each transaction:

Table D3-15 IsShared and PassDirty permitted responses

Transaction	IsShared	PassDirty
ReadNoSnoop	0	0
ReadOnce	0	0
	1	0
ReadClean	0	0
	1	0
ReadNotSharedDirty	0	0
	0	1
	1	0
ReadShared	0	0
	0	1
	1	0
	1	1
ReadUnique	0	0
	0	1
CleanUnique	0	0
MakeUnique	0	0
CleanShared	0	0
	1	0

Table D3-15 IsShared and PassDirty permitted responses (continued)

Transaction	IsShared	PassDirty
CleanInvalid	0	0
MakeInvalid	0	0
Read Barrier	0	0
DVM Message	0	0
DVM Complete	0	0

Note

Table D3-15 on page D3-187 only shows the permitted responses on **RRESP[3:2]**. For the permitted responses on **CRRESP,** see *Snoop response channel signaling* on page D3-195.

D3.3 Read acknowledge signaling

Table D3-16 shows the additional read acknowledge signal. This signal indicates that a master has completed a read transaction.

Table D3-16 Read acknowledge signaling.

Signal	Source	Description
RACK	Master	Read acknowledge signal

The **RACK** signal must be asserted for a single cycle and the interconnect must accept it in a single cycle.

The **RACK** signal must not be asserted before the cycle after the completion of the associated **RVALID**/**RREADY** handshake of the last read data channel transfer, as indicated by **RLAST**. The assertion of **RACK** must not be delayed to wait for the completion of another transaction. See *Handshake process* on page A3-41.

Read acknowledge must be sent for all transactions including coherent, barrier, and DVM transactions.

No ordering information is associated with read acknowledge, it is ordered the same as the last read data item and the associated read responses.

The interconnect must use read acknowledge to ensure that a transaction to a master's snoop port is not issued until any preceding transaction from that master to the same address has completed. See *Read and Write Acknowledge* on page D6-250.

D3.4 Write response channel signaling

Write transactions do not have additional response signaling. See *Read and write response structure* on page A3-59 for information on the baseline write response signaling.

The order that write responses for a single AXI ID are provided is the same as the order that the transactions are issued on the AW channel. Leading write data does not change the order in which the write responses are provided.

Note

The EXOKAY response is only permitted for a WriteNoSnoop transaction.

D3.5 Write Acknowledge signaling

Table D3-17 shows the additional write acknowledge signal. This signal indicates that a master has completed a write transaction.

Table D3-17 Write acknowledge signaling

Signal	Source	Description
WACK	Master	Write acknowledge signal

The **WACK** signal is asserted by a master for a single cycle and the interconnect must accept the **WACK** signal in a single cycle.

The **WACK** signal must not be asserted before the cycle after the completion of the associated **BVALID**/**BREADY** handshake. The assertion of **WACK** must not be delayed to wait for the completion of another transaction. See *Handshake process* on page A3-41.

Write acknowledge, **WACK** is asserted for all write transactions, including barrier transactions.

No ordering information is associated with write acknowledge, it is ordered the same as the associated write responses.

The interconnect must use write acknowledge to ensure that a transaction to a master's snoop port is not issued until any preceding transaction from that master to the same address has completed. See *Read and Write Acknowledge* on page D6-250.

D3.6 Snoop address channel signaling

The following sections define the snoop address channel and signals.

D3.6.1 About the snoop address channel

The snoop address channel (AC channel) is necessary for a master that:

- Holds cached copies of shared data
- Supports DVM transactions

The snoop address channel is an input channel to a cached master. The snoop address channel passes the snoop transactions of other components to a cached master so that the master can determine what actions it must take. The master can respond to the snoop transactions in different ways and its response determines what action the interconnect must take to complete the snoop process.

Supplementary information

Control information on the snoop address channel is a subset of the control information on the normal address channel. It provides sufficient information for the coherency operations, but does not provide unnecessary information. The snoop address channel does not provide information on the:

- Burst type
- Burst length
- Transaction size
- Modifiable or Shareable nature of the transaction
- Transaction ID

Fundamentally, the snoop address channel provides the same transactions that are issued on the normal read and write address channels. However, there are a number of exceptions.

The following transactions are not presented on the snoop address channel:

- Non-snooping type transactions. These transactions are:
  - ReadNoSnoop
  - WriteNoSnoop
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict
- WriteUnique. Other cached masters see a transaction, such as CleanInvalid. This ensures that all other copies of the cache line are cleaned to main memory and removed before the write occurs.
- WriteLineUnique. Other cached masters see a transaction, such as MakeInvalid. This ensures that all other copies of the cache line are removed before the write occurs.
- MakeUnique. Typically, this transaction is converted to a MakeInvalid transaction.
- CleanUnique. Typically, this transaction is converted to a CleanInvalid transaction.

Some snoop operations can be fulfilled without snooping every cached master in the system. Therefore, the snoop address channel for a cached master might not provide every snoop transaction.

Snoop read transactions can be adapted by the interconnect so that when the required data is obtained, further snoops to other masters are not requested.

Transactions that are not required to be a full cache line length are converted to be a full cache line length. These transactions are:

- ReadOnce
- WriteUnique

D3.6.2 Snoop address channel signaling

Table D3-18 shows the signals on the snoop address channel.

Table D3-18 Snoop address channel signals

Signal	Source	Description
ACVALID	Interconnect	Indicates the snoop address channel signals are valid
ACREADY	Master	Indicates a transfer on the snoop address channel can be accepted
ACADDR[ac-1:0][a]	Interconnect	The address of the first transfer in a snoop transaction
ACSNOOP[3:0]	Interconnect	Snoop transaction type. See Table D3-19
ACPROT[2:0]	Interconnect	Protection attributes of a snoop transaction

a. ac is the width of the snoop address bus.

The standard AXI **VALID**/**READY** handshake rules apply. See *Handshake process* on page A3-41.

When the **ACVALID** signal is asserted the snoop address and control signals on **ACADDR**, **ACPROT**, and **ACSNOOP** must not change, until **ACREADY** is asserted by the master. When **ACVALID** is asserted, it must remain asserted until **ACREADY** is asserted.

It is permitted to assert **ACREADY** before or in the same cycle as **ACVALID**. If **ACREADY** is asserted before **ACVALID**, then **ACREADY** can be deasserted without **ACVALID** being asserted.

ACADDR must be aligned to the data transfer size, which is determined by the width of the snoop data bus in bytes.

ACPROT[1] indicates the Secure or Non-secure nature of the snoop transaction.

For coherency transactions, **ACPROT[1]** can be considered as defining two address spaces, a Secure address space and a Non-secure address space, and can be treated as an additional address bit. Any aliasing between the Secure and Non-secure address spaces must be handled correctly.

Hardware coherency does not manage coherency between Secure and Non-secure address spaces.

ACSNOOP indicates the snoop transaction type. Not all transaction types, observed on the read address channel or write address channel can be observed on the snoop address channel. Table D3-19 shows the **ACSNOOP** encodings for the transactions that can be observed on the snoop address channel. All unused encodings are Reserved.

Table D3-19 ACSNOOP encodings

ACSNOOP[3:0]	Transaction
`0b0000`	ReadOnce
`0b0001`	ReadShared
`0b0010`	ReadClean
`0b0011`	ReadNotSharedDirty
`0b0111`	ReadUnique
`0b1000`	CleanShared
`0b1001`	CleanInvalid

Table D3-19 ACSNOOP encodings (continued)

ACSNOOP[3:0]	Transaction
0b1101	MakeInvalid
0b1110	DVM Complete
0b1111	DVM Message

A snoop transaction of burst length greater than one must be of burst type WRAP. A snoop transaction of burst length one must be of burst type INCR.

A snoop transaction must be a full cache line in length.

Note

If the original transaction that caused the snoop process was not a full cache line in length, then the interconnect must convert it to be a full cache line in length.

A snoop transaction must be the same width as the snoop data channel.

D3.7 Snoop response channel signaling

Table D3-20 shows the signals on the snoop response channel.

Table D3-20 Snoop response channel signals

Signal	Source	Meaning	Description
CRVALID	Master	Snoop response valid.	Indicates that the snoop response channel signals are valid.
CRREADY	Interconnect	Snoop response ready.	Indicates that a transfer on the snoop response channel can be accepted.
CRRESP[4:0]	Master	Snoop response.	Read response, indicates the status of a snoop transfer.

The standard AXI **VALID**/**READY** handshake rules apply. See *Handshake process* on page A3-41.

When the **CRVALID** signal is asserted, the snoop response must not change until the interconnect sets **CRREADY** HIGH. When **CRVALID** is asserted, it must remain asserted until **CRREADY** is asserted.

It is permitted to assert **CRREADY** before or in the same cycle as **CRVALID**. If **CRREADY** is asserted before **CRVALID**, then **CRREADY** can be deasserted without **CRVALID** being asserted.

A snoop response is required on the snoop response channel for each snoop address that is presented to a cached master on the snoop address channel.

All snoop transactions are ordered. A response on the snoop response channel must be in the same order as the addresses on the snoop address channel.

Table D3-20 shows the allocation of each bit of **CRRESP**[4:0].

Table D3-21 Snoop response bit allocations

Signal	Name	Meaning	
CRRESP[0]	DataTransfer	HIGH	Indicates that a full cache line of data will be provided on the snoop data channel for this transaction.
		LOW	Indicates that no data will be provided on the snoop data channel for this transaction.
CRRESP[1]	Error	HIGH	When HIGH, the Error bit indicates that the snooped cache line is in error. Typically, this is caused by a corrupt cache line that has been detected through the use of an *Error Correction Code* (ECC) system.
		LOW	Indicates that no Error condition has been detected.
CRRESP[2]	PassDirty	HIGH	When HIGH, it indicates that before the snoop process, the cache line was held in a Dirty state and the responsibility for writing the cache line back to main memory is being passed to the initiating master or the interconnect. For all transactions, except MakeInvalid, if the cache line was held in a Dirty state before the snoop process and a copy is not being retained by the cache, then the PassDirty bit must be set HIGH.
		LOW	Indicates the responsibility for writing the cache line back to main memory is not being passed.

Table D3-21 Snoop response bit allocations (continued)

Signal	Name	Meaning	
CRRESP[3]	IsShared	HIGH	Indicates that the snooped cache retains a copy of the cache line after the snoop process has completed.
		LOW	Copy of cache line was not retained.
CRRESP[4]	WasUnique	HIGH	Indicates that the cache line was held in a Unique state before the snoop process. The WasUnique bit must only be HIGH if it is known that no other cache can have a copy of the cache line.
		LOW	No information is provided on whether or not the cache line was held in a Unique state before the snoop process.

The meaning of the snoop response bits and the limitations of use are as follows:

DataTransfer bit

The snoop transaction type and the state of the cache line in the snooped cache determine whether the DataTransfer bit is set HIGH and a data transfer occurs.

For the following transactions, a data transfer occurs if the snoop process has resulted in a cache hit:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique

If the cache line is Clean, it is not mandatory that data transfer occurs. However, this specification recommends that data transfer still occurs.

For the following transactions, data transfer must occur if the snoop process has resulted in a cache hit and the cache line is Dirty:

- CleanInvalid
- CleanShared

A MakeInvalid transaction never requires a data transfer.

The DataTransfer bit can be set to 1 and data can be returned on the snoop data channel when it is not required. For example, a CleanInvalid transaction can return data when it holds the cache line in a Clean state, and a MakeInvalid transaction can return data. However, this specification does not recommend this use of the snoop data channel.

Note

The protocol permits the return of data on the snoop data channel when it is not required to enable a simplified snoop port implementation to handle a MakeInvalid transaction in the same manner as a ReadUnique or CleanInvalid, and a CleanInvalid transaction to be handled in the same manner as a ReadUnique.

Error bit

When HIGH, it indicates that the snooped cache line is in error. Typically, this is caused by a corrupt cache line that has been detected through the use of an *Error Correction Code*, (ECC) system.

If an error is detected, the snooped cache can take appropriate action, such as discarding a Clean cache line. Alternatively, the snooped cache can flag the error by setting the Error bit HIGH and take no further action.

PassDirty bit

For the following transactions, the responsibility for writing the Dirty cache line back to main memory can be passed to the master requesting the data:

- ReadNotSharedDirty
- ReadShared
- ReadUnique

In other cases, such as ReadClean, the Dirty cache line must be written back to main memory by the interconnect.

IsShared bit

The restrictions on the use of IsShared are:

- For the following transactions, the cache line in the snooped cache must be invalidated and the IsShared response must be LOW:
  - ReadUnique
  - CleanInvalid
  - MakeInvalid
- For the following transactions, the snooped cache can determine if it retains a copy of the cache line after the snoop process has completed, and the snooped cache must use the IsShared bit to signal the outcome:
  - ReadOnce
  - ReadClean
  - ReadNotSharedDirty
  - ReadShared
  - CleanShared

Typically, a snooped cache retains a local copy of the cache line after the snoop process has completed. However, there are cases when a snooped cache does not retain a local copy, such as when passing the cache line as unique to another cache in response to a ReadNotSharedDirty snoop transaction.

Note

For a ReadOnce snoop transaction, the IsShared bit must be set to 1 if the snooped master is retaining a copy of the cache line, even if it is keeping the line in a Unique state.

WasUnique bit

A WasUnique response permits the snoop process to be terminated because no other cache can hold a copy of the cache line.

The protocol permits a cache to not generate a WasUnique response. In this case, the WasUnique bit must be permanently LOW.

Note

Permanently setting the WasUnique LOW can result in the cache line being provided to the original requester as Shared, when it could have been provided as Unique. It can also result in additional caches being needlessly snooped.

Table D3-22 shows the **CRRESP** response meanings and transactions for which they are valid.Table D3-22 does not show the Error bit **CRRESP[1]**, of the response field, because the value of this bit does not affect the meaning of the other snoop response bits. In this table:

- The meaning of the bits in the **CRRESP** response fields, when asserted, are:

  WU, WasUnique The cache line was in Unique state before this snoop.

  IS, IsShared The cache is keeping a copy of this cache line after this snoop.

  PD, PassDirty The cache line was Dirty before this snoop. This response transfers responsibility for updating main memory, as well as the data.

  DT, DataTransfer The response to the snoop transaction includes a transfer on the snoop data channel.

- The snoop transactions are abbreviated as follows:

  RO ReadOnce
  RC ReadClean
  RN ReadNotSharedDirty
  RS ReadShared
  RU ReadUnique
  CI CleanInvalid
  MI MakeInvalid
  CS CleanShared

- Whether a response is permitted, for each transaction, is indicated as follows:

  E Expected response
  P Permitted response
  No Response not permitted

Table D3-22 Response meanings and transactions for which they are valid

CRRESP[3:2,0] [4]				Snoop transaction								
IS	**PD**	**DT**	**WU**	**RO**	**RC**	**RN**	**RS**	**RU**	**CI**	**MI**	**CS**	**Response meaning**
0	0	0	0	E	E	E	E	E	E	E	E	Line was invalid or has been invalidated.
0	0	0	1	P	P	P	P	P	E	E	E	Line was unique but has been invalidated.
0	0	1	x	P	P	P	P	E	P	P	P	Passing clean data before invalidating.
x	1	0	x	No	No	No	No	No	No	No	No	Cannot assert PassDirty with DataTransfer low.
0	1	1	x	P	E	E	P	E	E	P	P	Passing dirty data before invalidating.
1	0	0	x	P	P	P	P	No	No	No	E	Line is valid and clean but not being passed.
1	0	1	x	E	E	E	E	No	No	No	P	Passing clean data and keeping copy.
1	1	1	x	P	E	E	E	No	No	No	E	Passing dirty data and keeping copy.

The following responses are illegal:

- IsShared, **CRRESP[3]** = 1 for:
  - ReadUnique
  - CleanInvalid
  - MakeInvalid
- PassDirty, **CRRESP[2]** = 1, and DataTransfer, **CRRESP[0]** = 0, for any transaction.

D3.8 Snoop data channel signaling

Table D3-23 shows the signals on the snoop data channel.

Table D3-23 Snoop data channel signals

Signal	Source	Description
CDVALID	Master	Indicates that the snoop data channel signals are valid .
CDREADY	Interconnect	This signal indicates the snoop data can be accepted in the current cycle.
CDDATA[cd-1:0][a]	Master	Snoop data.
CDLAST	Master	Indicates whether this is the last data transfer in a snoop transaction.

a. cd is the width of the snoop data bus.

The standard AXI **VALID**/**READY** handshake rules apply. See *Handshake process* on page A3-41.

If the **CDVALID** signal is asserted, the data value on **CDDATA** and the last transfer indication on **CDLAST**, must not change until **CDREADY** is asserted to indicate that the information has been accepted by the interconnect. When **CDVALID** is asserted, it must remain asserted until **CDREADY** is asserted.

The assertion of **CDREADY** is permitted before, or in the same cycle as, **CDVALID**. If **CDREADY** is asserted before **CDVALID,** then **CDREADY** can be deasserted without **CDVALID** being asserted.

The width of the snoop data bus, **CDDATA** is not required to be the same width as the read data and write data buses.

Note

Where the expected cache hit rate is low and transfer latency is not important, a snoop data bus narrower than the read and write data buses can be implemented.

The snoop data bus can be 32, 64, 128, 256, 512, or 1024 bits wide. However, the following restrictions apply:

- All cache line size transactions must be a full data bus width.
- The burst length must be 1, 2, 4, 8, or 16.

These restrictions determine the minimum and maximum data bus widths that can be supported for a given cache line size. See *Cache line size restrictions* on page D3-182.

When **CDVALID** is asserted, all byte lanes of **CDDATA** must be valid, as the snoop data bus does not support byte strobes.

Snoop data is not required for every snoop transaction, it is only provided for a snoop transaction that has a snoop response with the DataTranfer bit asserted. See *Snoop response channel signaling* on page D3-195. When snoop data is required, it must be provided in the same order as the associated snoop addresses were presented on the snoop address channel.

All snoop transactions of burst length greater than one are defined to be of burst type WRAP. The order in which data transfers within a snoop burst are provided is the same as for a standard wrapping burst. See *Burst type* on page A3-49.

The **CDLAST** signal must be asserted during the final data transfer associated with a snoop transaction.

The snoop data channel is optional. However, any cached master that does not support a snoop data channel must still support all snoop transaction types on the snoop address channel.

The cached master must not be required to return dirty data to complete a snoop transaction, and must never use a snoop response with DataTransfer asserted. To achieve this, the cached master must either:

- Not hold dirty data
- Perform a WriteBack or WriteClean before responding to any snoop process that must obtain a Dirty cache line

Note

This option is not compatible with the WriteUnique and WriteLineUnique transactions. See *Write transactions* on page D4-222.

D3.9 Snoop channel dependencies

There are dependencies between the signals on different snoop channels.

In Figure D3-1:

- Single-headed arrows point to signals that can be asserted before or after the signal at the start of the arrow.
- Double-headed arrows point to signals that must be asserted only after assertion of the signal at the start of the arrow.

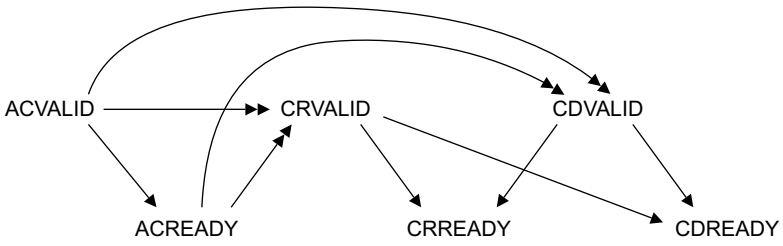


Figure D3-1 Snoop channel dependencies

Figure D3-1 shows the snoop address, snoop response, and snoop data channel dependencies and shows that:

- The interconnect must not wait for the master to assert **ACREADY** before asserting **ACVALID**.
- The master can wait for **ACVALID** to be asserted before asserting **ACREADY**.
- The master must wait for both **ACVALID** and **ACREADY** to be asserted before asserting **CRVALID**.
- The master must wait for both **ACVALID** and **ACREADY** to be asserted before asserting **CDVALID**.
- The master must not wait for the interconnect to assert **CRREADY** or **CDREADY** before asserting **CRVALID**.
- If data transfer is required to complete the snoop operation, the master must not wait for the interconnect to assert **CRREADY** or **CDREADY** before asserting **CDVALID**.
- The interconnect can wait for **CRVALID** to be asserted before asserting **CRREADY** or **CDREADY**.
- If data transfer is required, the interconnect can wait for **CDVALID** to be asserted before asserting **CRREADY** or **CDREADY**.

Chapter D4
Coherency Transactions on the Read Address and Write Address Channels

This chapter describes the transactions that can be issued by an initiating master on the read address and write address channels. The expected channel activity for each transaction group is described, and a brief overview is given for each transaction. Each transaction has a description of the associated cache line state changes. It contains the following sections:

D4.1 About an initiating master

This section describes the behavior of an initiating master. Typically, an initiating master issues a transaction to progress an internal action such as a load or store operation.

The internal action requires:

- For a load, the master must get the data from either:
  - A valid copy of the appropriate cache line
  - A transaction that returns valid read data
- For a store, the master needs permission to store the cache line from either:
  - A copy of the appropriate cache line in a Unique state
  - A transaction type that gives the master permission to store the cache line

D4.1.1 Transaction groups

The following sections describe the expected channel activity for the transaction groups:

- *Read transactions*
- *Clean transactions*
- *Make transactions* on page D4-205
- *Write transactions* on page D4-205
- *Evict transactions* on page D4-205

Read transactions

The read transaction group is:

- ReadNoSnoop
- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique

A Read transaction progresses as follows:

1. The address is issued on the read address (AR) channel.
2. The data and response is returned on the read data (R) channel. The number of data beats required is determined by **ARLEN**.
3. Completion of a Read transaction is signaled by the master asserting **RACK**.

Clean transactions

The clean transaction group is:

- CleanUnique
- CleanShared
- CleanInvalid

A Clean transaction progresses as follows:

1. The address is issued on the AR channel.
2. A single transfer on the R channel returns the response. No data is returned for a Clean transaction.
3. Completion of a Clean transaction is signaled by the master asserting **RACK**.

Make transactions

The make transaction group is:

- MakeUnique
- MakeInvalid

For the initiating master, a Make transaction progresses as follows:

1. The address is issued on the AR channel.
2. A single transfer on the R channel returns the response. No data is returned for a Make transaction.
3. Completion of a Make transaction is signaled by the master asserting **RACK**.

Write transactions

The write transaction group is:

- WriteNoSnoop
- WriteUnique
- WriteLineUnique
- WriteBack
- WriteClean
- WriteEvict

For the initiating master, a Write transaction progresses as follows:

1. The address is issued on the AW channel.
2. The data is transferred on the W channel.
3. The response is returned on the B channel.
4. Completion of a Write transaction is signaled by the master asserting **WACK**.

Evict transactions

The evict transaction group is, Evict.

For the initiating master, an Evict transaction progresses as follows:

1. The address is issued on the AW channel.
2. The response is returned on the B channel. No data is transferred for an Evict transaction.
3. Completion of an Evict transaction is signaled by the master asserting **WACK**.

Read barrier transactions

For the master initiating the transaction, a Read Barrier transaction progresses as follows:

1. The transaction is issued on the AR channel.
2. A single transfer on the R channel returns the response. No data is returned for a Read Barrier transaction.
3. Completion of a Read Barrier transaction is signaled by the master asserting **RACK**.

See Chapter D8 *Barrier Transactions*.

Write barrier transactions

For the master initiating the transaction, a Write Barrier transaction progresses as follows:

1. The transaction is issued on the AW channel.
2. The response is returned on the B channel. No data is transferred for a Write Barrier transaction.
3. Completion of a Write Barrier transaction is signaled by the master asserting **WACK**.

See Chapter D8 *Barrier Transactions*.

DVM transactions

For the master initiating the transaction, a DVM transaction progresses as follows:

1. The transaction is issued on the AR channel.
2. A single transfer on the R channel returns the response. No data is returned for a DVM transaction.
3. Completion of a DVM transaction is signaled by the master asserting **RACK**.

See Chapter D13 *Distributed Virtual Memory Transactions*.

D4.2 About snoop filtering

Snoop filtering tracks the cache lines that are allocated in a master's cache. To support an external snoop filter, a cached master must be able to broadcast cache lines that are allocated and cache lines that are evicted.

Support for an external snoop filter is optional within the ACE protocol. A master component must state in its data sheet if it provides support. See Chapter D10 *Optional External Snoop Filtering* for the mechanism the ACE protocol supports for the construction of an external snoop filter.

For a master component that does not support an external snoop filter, the cache line states that are permitted after a transaction has completed are less strict.

D4.3 State changes on different transactions

The state changes that can be associated with a transaction are determined by:

- The transaction type
- The read response for transactions that are issued on the AR channel
- Whether the master supports an external snoop filter
- Whether the master performs speculative reads

The rules that apply to a master are:

- If a transaction read response has PassDirty asserted, then the cache line must move to a Dirty state. The PassDirty response can be asserted for:
  - ReadNotSharedDirty
  - ReadShared
  - ReadUnique
- If a transaction read response has IsShared asserted, then the cache line must move to either a Shared state or the Invalid state. The IsShared response can be asserted for:
  - ReadOnce
  - ReadClean
  - ReadNotSharedDirty
  - ReadShared
  - CleanShared
- A cache line that is in a Unique state is permitted to move to the equivalent Shared state, but this is not expected behavior.
- If an external snoop filter is not supported, a cache line that is in a Clean state can move to the Invalid state.

D4.3.1 State changes associated with a load

No cache line state change is required for the internal action of a load.

D4.3.2 State changes associated with a coherent store

Before carrying out the internal operation of a store to a cache line in Shareable memory, the master must ensure that it has permission to store. A master has permission to store if the cache line is in the UniqueClean or UniqueDirty state.

If the master does not have permission to store then it must either:

- Issue a transaction on the AR channel that obtains permission to store, and then perform the store to the cache line. After the store to a cache line, the master must be in the UniqueDirty state. The transactions that obtain permission to store are:
  - ReadUnique
  - CleanUnique
  - MakeUnique
- Issue a transaction on the AW channel that obtains permission to store and also updates main memory. The transactions that obtain permission to store data and also update main memory are:
  - WriteUnique
  - WriteLineUnique

D4.3.3 State changes associated with a main memory update

An update to main memory can be performed when the cache line is in a Dirty state.

When a master is given permission to update main memory, the earliest the associated write transaction can occur is the cycle after the **RVALID**/**RREADY** handshake in which **RLAST** is asserted for the transaction that gave permission to update main memory.

An update to main memory is performed using a WriteBack or WriteClean transaction.

After an update to main memory, the cache line must be in a Clean or Invalid state.

If an external snoop filter is supported, then the following restrictions apply:

- After a WriteBack transaction, the cache line must be in the Invalid state.
- After a WriteClean transaction, the cache line must be in a Clean state.

D4.3.4 State changes associated with cache maintenance operations

The cache maintenance transactions are:

- CleanShared
- CleanInvalid
- MakeInvalid

Before issuing a cache maintenance transaction, the master must ensure that:

- For CleanShared, the cache line must be in a Clean or Invalid state.
- For CleanInvalid and MakeInvalid, the cache line must be in the Invalid state.

Note

A cache maintenance transaction does not change the cache line state.

D4.4 State change descriptions

The cache line state changes associated with a transaction are defined in the following sections:

- *Read transactions* on page D4-211
- *Clean transactions* on page D4-217
- *Make transactions* on page D4-220
- *Write transactions* on page D4-222
- *Evict transactions* on page D4-227

For each transaction, the starting state for the transaction and the three possible end state groups are given. The three possible end state groups are:

- The expected end states, which are also the end states that this specification recommends.
- The full list of legal end states for a cached master that supports an external snoop filter. This set of end states takes into account that:
  - A cache line in UniqueClean state can always be held in SharedClean state.
  - A cache line in UniqueDirty state can always be held in SharedDirty state.
- The full list of legal end states for a cached master that does not support an external snoop filter. This full list of legal end states includes the legal end states for external snoop filter support, and takes into account that a cache line in UniqueClean state, or SharedClean state, can be in the Invalid state.

Some transactions have two tables provided. The first table shows the expected starting states when the transaction is issued. The second table shows the other permitted starting states for the transaction that is not normally issued. For example, a ReadShared transaction with a Valid starting state. Typically, the transaction and starting state combinations in the second table are associated with a speculative read where the master issues a transaction before it has determined the state of the cache line in its local cache.

Any state that is not shown as a starting state in the tables is not a legal starting state.

The starting state is defined as the cache line state just before the transaction response is received by the initiating master. If the initiating master receives a snoop transaction to the same cache line between issuing a transaction and receiving the associated response, then the cache line state changes required by the snoop transaction must be applied first. See Chapter D5 *Snoop Transactions*.

The following abbreviations are used for the cache line states:

UC UniqueClean
UD UniqueDirty
SC SharedClean
SD SharedDirty
I Invalid

D4.5 Read transactions

This section defines the state changes associated with the Read transaction group that are issued on the AR channel. The Read transactions are:

- *ReadNoSnoop*
- *ReadOnce* on page D4-212
- *ReadClean* on page D4-212
- *ReadNotSharedDirty* on page D4-213
- *ReadShared* on page D4-214
- *ReadUnique* on page D4-215

D4.5.1 ReadNoSnoop

ReadNoSnoop is a read transaction that is used in a region of memory that is not Shareable with other masters. The transaction response requirements are:

- The IsShared response must be deasserted.
- The PassDirty response must be deasserted.

Table D4-1 shows the expected cache line state changes for the ReadNoSnoop transaction:

Table D4-1 Expected ReadNoSnoop cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadNoSnoop	I	00	I, UC	I, UC, SC	I, UC, SC

Note

A ReadNoSnoop transaction does not indicate when the cache line is allocated after the transaction has completed.

Table D4-2 shows the other permitted cache line state changes for the ReadNoSnoop transaction:

Table D4-2 Other permitted ReadNoSnoop cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadNoSnoop	UC	00	I, UC	I, UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	I, UC	I, UC, SC	I, UC, SC
	SD	00	UD	UD, SD	UD, SD

D4.5.2 ReadOnce

ReadOnce is a read transaction that is used in a region of memory that is Shareable with other masters. This transaction is used when a snapshot of the data is required. The location is not cached locally for future use.

The transaction response requirements are:

- The IsShared response indicates that the cache line is shared or unique.
- The PassDirty response must be deasserted.

Table D4-3 shows the expected cache line state changes for the ReadOnce transaction:

Table D4-3 Expected ReadOnce cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadOnce	I	00	I	I	I
		10	I	I	I

Table D4-4 shows the other permitted cache line state changes for the ReadOnce transaction:

Table D4-4 Other permitted ReadOnce cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadOnce	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	UC	UC, SC	I, UC, SC
		10	SC	SC	I, SC
	SD	00	UD	UD, SD	UD, SD
		10	SD	SD	SD

D4.5.3 ReadClean

ReadClean is a read transaction that is used in a region of memory that is Shareable with other masters. A ReadClean transaction is guaranteed not to pass responsibility for updating main memory to the initiating master.

Typically, a ReadClean transaction is used by a master that wants to obtain a Clean copy of a cache line, for example a master with a write-through cache.

The transaction response requirements are:

- The IsShared response indicates that the cache line is shared or unique.
- The PassDirty response must be deasserted.

Table D4-5 shows the expected cache line state changes for the ReadClean transaction:

Table D4-5 Expected ReadClean cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadClean	I	00	UC	UC, SC	I, UC, SC
		10	SC	SC	I, SC

Table D4-6 shows other permitted cache line state changes for the ReadClean transaction:

Table D4-6 Other permitted ReadClean cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadClean	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	UC	UC, SC	I, UC, SC
		10	SC	SC	I, SC
	SD	00	UD	UD, SD	UD, SD
		10	SD	SD	SD

D4.5.4 ReadNotSharedDirty

ReadNotSharedDirty is a read transaction that is used in a region of memory that is Shareable with other masters. A ReadNotSharedDirty transaction can complete with any combination of the IsShared and PassDirty responses except for IsShared and PassDirty asserted.

Typically, the transaction is used by a cached master that is carrying out a load operation and can accept the cache line in any state except the SharedDirty state.

The transaction response requirements are:

- The IsShared response indicates that the cache line is Shared or Unique.
- The PassDirty response indicates that the cache line is Clean or Dirty.
- If the IsShared response indicates that the cache line is Shared, then the PassDirty response must indicate that the cache line is Clean.

Table D4-7 shows the expected cache line state changes for the ReadNotSharedDirty transaction:

Table D4-7 Expected ReadNotSharedDirty cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadNotSharedDirty	I	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD
		10	SC	SC	I, SC

Table D4-8 shows other permitted cache line state changes for the ReadNotSharedDirty transaction:

Table D4-8 Other permitted ReadNotSharedDirty cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadNotSharedDirty	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD
		10	SC	SC	I, SC
	SD	00	UD	UD, SD	UD, SD
		10	SD	SD	SD

Note

If a cache line starts in the SharedClean state, and the transaction response has PassDirty asserted, the cache line must move to a Dirty state.

D4.5.5 ReadShared

ReadShared is a read transaction that is used in a region of memory that is Shareable with other masters. A ReadShared transaction can complete with any combination of the IsShared and PassDirty responses.

Typically, the ReadShared transaction is used by a cached master that is carrying out a load operation and can accept the cache line in any state.

The transaction response requirements are:

- The IsShared response indicates that the cache line is Shared or Unique.
- The PassDirty response indicates that the cache line is Clean or Dirty.

Table D4-9 shows the expected cache line state changes for the ReadShared transaction:

Table D4-9 Expected ReadShared cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadShared	I	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD
		10	SC	SC	I, SC
		11	SD	SD	SD

Table D4-10 shows other permitted state changes for the ReadShared transaction:

Table D4-10 Other permitted ReadShared cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadShared	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD
		10	SC	SC	I, SC
		11	SD	SD	SD
	SD	00	UD	UD, SD	UD, SD
		10	SD	SD	SD

Note

If a cache line starts in the SharedClean state, and the transaction response has PassDirty asserted, the cache line must move to a Dirty state.

D4.5.6 ReadUnique

A ReadUnique transaction is used in a region of memory that is Shareable with other masters. The transaction gets a copy of the data and also ensures that the cache line can be held in a Unique state. This permits the master to carry out a store operation to the cache line.

Typically, a ReadUnique transaction is used when the initiating master is carrying out a partial cache line store and does not have a copy of the cache line.

The transaction response requirements are:

- The IsShared response must be deasserted to indicate that the cache line is Unique.
- The PassDirty response must indicate when the cache line is Clean or Dirty.

Note

The cache line state changes associated with the ReadUnique transaction that Table D4-11 and Table D4-12 on page D4-216 show, do not include the cache line state changes associated with any subsequent store operation by the master when the cache line is in a Unique state.

Table D4-11 shows the expected cache line state changes for the ReadUnique transaction:

Table D4-11 Expected ReadUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadUnique	I	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD

Table D4-12 shows other permitted cache line state changes for the ReadUnique transaction:

Table D4-12 Other permitted ReadUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
ReadUnique	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD
	SC	00	UC	UC, SC	I, UC, SC
		01	UD	UD, SD	UD, SD
	SD	00	UD	UD, SD	UD, SD

D4.6 Clean transactions

This section defines the state changes associated with the Clean transaction group that are issued on the AR channel. The Clean transactions are:

- *CleanUnique*
- *CleanShared* on page D4-218
- *CleanInvalid* on page D4-219

D4.6.1 CleanUnique

A CleanUnique transaction is used in a region of memory that is Shareable with other masters. The CleanUnique transaction ensures that:

- The cache line can be held in a Unique state. This permits the master to carry out a store operation to the cache line, but the transaction does not obtain a copy of the data for the master.
- Data held in another cache in a Dirty state is written to main memory and all other copies of the cache line are removed.

Typically, a CleanUnique transaction is used before a partial cache line store operation to Shareable memory when the master already has a copy of the data.

The transaction response requirements are:

- The IsShared response must be deasserted to indicate that the cache line is unique.
- The PassDirty response must be deasserted.

Note

The cache line state changes associated with the CleanUnique transaction that Table D4-13 and Table D4-14 on page D4-218 show, do not include the cache line state changes associated with any subsequent store operation by the master when the cache line is in a Unique state.

Table D4-13 shows the expected cache line state changes for the CleanUnique transaction:

Table D4-13 Expected CleanUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
CleanUnique	SC	00	UC	UC, SC	I, UC, SC
	SD	00	UD	UD, SD	UD, SD

Table D4-14 shows other permitted cache line state changes for the CleanUnique transaction:

Table D4-14 Other permitted CleanUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
CleanUnique	I	00	I	I[a]	I
	UC	00	UC	UC, SC	I, UC, SC
	UD	00	UD	UD, SD	UD, SD

a. See *Snoop filter cache line allocation awareness*.

On completing a CleanUnique transaction, the initiating master has permission to store to the cache line. If the cache line was in the Invalid state before the store operation, then the store must be a full cache line size for the cache line to be allocated in the cache. After the full cache line store, the cache line is in the UniqueDirty state. The store must occur atomically with the completion of the CleanUnique transaction. Therefore, any snoop that occurs after the CleanUnique transaction must be delayed until the store is complete.

CleanUnique transactions can be used for Exclusive accesses, see Chapter D9 *Exclusive Accesses from ACE Masters*.

Snoop filter cache line allocation awareness

A snoop filter regards a cache line as allocated after the completion of a CleanUnique transaction. Therefore, the snoop filter has the correct information on the allocation of a cache line in the following circumstances:

- The cache line was allocated before the CleanUnique transaction and remains allocated after the CleanUnique transaction completes.
- If the cache line was not allocated before the CleanUnique transaction, or the cache line was invalidated during the CleanUnique transaction, then when the CleanUnique transaction completes the master:
  - Performs a full cache line store and the cache line is allocated.
  - Performs a WriteBack transaction of either a full or partial cache line store, and indicates to the snoop filter that the cache line is no longer allocated.
  - Reissues another transaction, for example a ReadUnique transaction, before performing a full or partial cache line store and the cache line becomes allocated.
  - Does not perform a store operation. In this situation, the master must issue an Evict transaction to indicate to the snoop filter that the cache line is no longer allocated.

D4.6.2 CleanShared

A CleanShared transaction is a broadcast cache clean operation. It can be used in Shareable and Non-shareable memory regions.

A CleanShared transaction is used to ensure that all cached copies of a main memory location are Clean.

Note

If the master carrying out the cache maintenance operation holds the cache line in a Dirty state, then the master must carry out a WriteBack or WriteClean transaction so that the cache line is in a Clean state before it issues a CleanShared transaction. While a CleanShared is in progress, the master is permitted to write to the line, so it might become Dirty before the CleanShared completes.

The transaction response requirements are:

- The IsShared response indicates that the cache line is shared or unique.
- The PassDirty response must be deasserted.

Table D4-15 shows the expected cache line state changes for the CleanShared transaction:

Table D4-15 Expected CleanShared cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
CleanShared	I	00	I	I	I
		10	I	I	I
	UC	00	UC	UC, SC, UD[a], SD[a]	I, UC, SC, UD[a], SD[a]
	SC	00	UC	UC, SC	I, UC, SC
		10	SC	SC	I, SC

a. A line in the UC state might become Dirty as a result of a local write, rather than the response to a CleanShared transaction.

D4.6.3 CleanInvalid

A CleanInvalid transaction is a broadcast cache clean and invalidate operation. It can be used in Shareable and Non-shareable memory regions.

A CleanInvalid transaction is used to ensure that main memory is updated and there are no cached copies of a main memory location.

Note

If the master carrying out the cache maintenance operation holds the cache line in a Dirty state, then the master must carry out a WriteBack or WriteClean transaction. Then the master must invalidate the cache line, so that the cache line is in the Invalid state before it issues a CleanInvalid transaction.

The transaction response requirements are:

- The IsShared response must be deasserted.
- The PassDirty response must be deasserted.

Table D4-16 shows the expected cache line state changes for the CleanInvalid transaction:

Table D4-16 Expected CleanInvalid cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
CleanInvalid	I	00	I	I	I

D4.7 Make transactions

This section defines the state changes associated with the Make transaction group that are issued on the AR channel. The Make transactions are:

- *MakeUnique*
- *MakeInvalid* on page D4-221

D4.7.1 MakeUnique

A MakeUnique transaction is used in a region of memory that is Shareable with other masters. The MakeUnique transaction ensures that:

- The cache line can be held in a Unique state. This permits the master to carry out a store operation to the cache line, but the transaction does not obtain a copy of the data for the master.
- All other copies of the cache line are removed.

Note

A MakeUnique transaction must be used only by an initiating master that is carrying out a full cache line store operation.

The transaction response requirements are:

- The IsShared response must be deasserted indicating that the cache line is unique.
- The PassDirty response must be deasserted.

The expected cache line state changes for a MakeUnique transaction are different from all other transactions because a MakeUnique transaction must be coupled to a full cache line store operation.

Table D4-17 shows the expected cache line state changes for the MakeUnique transaction with a full cache line store operation:

Table D4-17 Expected MakeUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
MakeUnique with full cache line store	I	00	UD	UD, SD	UD, SD
	SC	00	UD	UD, SD	UD, SD
	SD	00	UD	UD, SD	UD, SD

Table D4-18 shows the other permitted cache line state changes for the MakeUnique transaction with a full cache line store operation:

Table D4-18 Other permitted MakeUnique cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
MakeUnique with full cache line store	UC	00	UD	UD, SD	UD, SD
	UD	00	UD	UD, SD	UD, SD

D4.7.2 MakeInvalid

A MakeInvalid transaction is a broadcast cache invalidate operation. It can be used in Shareable and Non-shareable memory regions.

A MakeInvalid transaction is used to ensure that there are no cached copies of a main memory location.

Note

If the master carrying out the cache maintenance operation holds the cache line in a Valid state, then the master must invalidate the cache line, so that the cache line is in the Invalid state before it issues a MakeInvalid transaction.

The transaction response requirements are:

- The IsShared response must be deasserted.
- The PassDirty response must be deasserted.

Table D4-19 shows the expected cache line state changes for the MakeInvalid transaction:

Table D4-19 Expected MakeInvalid cache line state changes

Transaction	Start state	RRESP[3:2] IsShared/ PassDirty	Expected end state	Legal end state	
				With Snoop Filter	No Snoop Filter
MakeInvalid	I	00	I	I	I

D4.8 Write transactions

This section defines the state changes associated with the Write transaction group that are issued on the AW channel. The Write transactions are:

- *WriteNoSnoop*
- *WriteUnique* on page D4-223
- *WriteLineUnique* on page D4-223
- *WriteBack* on page D4-224
- *WriteClean* on page D4-224
- *WriteEvict* on page D4-225
- *Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226
- *Handling overlapping write transactions* on page D4-228

D4.8.1 WriteNoSnoop

A WriteNoSnoop transaction is used in a region of memory that is not Shareable with other masters. A WriteNoSnoop transaction can result from:

- A program action, such as a store operation.
- An update of main memory for a cache line that is in a Non-shareable region of memory.

Table D4-20 shows the expected cache line state changes for the WriteNoSnoop transaction:

Table D4-20 Expected WriteNoSnoop cache line state changes

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteNoSnoop	I	I	I, UC, SC	I, UC, SC
	UC	UC	I, UC, SC	I, UC, SC
	UD	UC	I, UC, SC	I, UC, SC

Note

A cache line must only move from the Invalid state to a Valid state if a full cache line store has been performed.

Table D4-21 shows the other permitted cache line state changes for the WriteNoSnoop transaction:

Table D4-21 Other permitted WriteNoSnoop cache line state changes

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteNoSnoop	SC	UC	I, UC, SC	I, UC, SC
	SD	UC	I, UC, SC	I, UC, SC

D4.8.2 WriteUnique

A WriteUnique transaction is used in a region of memory that is Shareable with other masters. A single write occurs that is required to propagate to main memory or a downstream cache.

There are restrictions on the use of WriteUnique transactions by cached masters that can hold Dirty cache lines. See *Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226.

Table D4-22 shows the expected cache line state changes for the WriteUnique transaction:

Table D4-22 Expected WriteUnique cache line state changes

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteUnique	I	I	I	I
	UC	SC	SC	I, SC
	SC	SC	SC	I, SC

In the case of master holding a line in a Clean state while performing a WriteUnique transaction, the cache line must be updated to the new value when the WriteUnique transaction response is received.

D4.8.3 WriteLineUnique

A WriteLineUnique transaction is used in a region of memory that is Shareable with other masters. A single write occurs, that is required to propagate to main memory or a downstream cache.

Note

A WriteLineUnique transaction must be a full cache line store and all bytes within the cache line must be updated.

There are restrictions on the use of WriteLineUnique transactions by cached masters that can hold Dirty cache lines. See *Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226.

Table D4-23 shows the expected cache line state changes for the WriteLineUnique transaction.

Table D4-23 Expected WriteLineUnique cache line state changes

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteLineUnique	I	I	I	I
	UC	SC	SC	I, SC
	SC	SC	SC	I, SC

In the case of master holding a line in a Clean state while performing a WriteLineUnique transaction, the cache line must be updated to the new value when the WriteLineUnique transaction response is received.

D4.8.4 WriteBack

A WriteBack transaction is a write that can be used in Shareable and Non-shareable regions of memory. A WriteBack transaction is a write of a Dirty cache line to update main memory or a downstream cache.

Note

The difference between a WriteBack and a WriteClean transaction is whether the cache line remains allocated in the cache for a Shareable region of memory. After a WriteBack transaction, the cache line is no longer allocated. After a WriteClean transaction, the cache line remains allocated.

The permitted state changes that Table D4-24 and Table D4-25 show, do not take into account a preceding store operation that makes a cache line Dirty. If a store operation and WriteBack transaction occur as an atomic process, then the legal cache line state changes can be determined by combining the legal state changes for a store operation. See *State changes associated with a coherent store* on page D4-208, followed by the legal state changes for a WriteBack transaction.

Table D4-24 shows the expected cache line state changes for the WriteBack transaction in a Shareable memory region.

Table D4-24 Expected WriteBack cache line state changes in a Shareable memory region

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteBack	UD	I	I	I, UC, SC
	SD	I	I	I, SC

Table D4-25 shows the expected cache line state changes for the WriteBack transaction in a Non-shareable memory region.

Table D4-25 Expected WriteBack cache line state changes in a Non-shareable memory region

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteBack	UD	I	I, UC, SC	I, UC, SC
	SD	I	I, UC, SC	I, UC, SC

D4.8.5 WriteClean

A WriteClean transaction is a write operation that can be used in Shareable and Non-shareable regions of memory. A WriteClean transaction is a write of a Dirty cache line to update main memory or a downstream cache.

Note

The difference between a WriteClean and a WriteBack transaction is the state of the cache line that remains allocated in the cache for a Shareable region of memory. After a WriteClean transaction, the cache line remains allocated. After a WriteBack transaction, the cache line is no longer allocated.

The permitted state changes that Table D4-26 on page D4-225 and Table D4-27 on page D4-225 show, do not take into account any preceding store operation that makes a cache line Dirty. If a store operation and WriteBack transaction occur as an atomic process, then the legal cache line state changes can be determined by combining the legal state changes for a store operation. See *State changes associated with a coherent store* on page D4-208, followed by the legal state changes for a WriteClean transaction.

Table D4-26 shows the expected cache line state changes for the WriteClean transaction in a Shareable memory region.

Table D4-26 Expected WriteClean cache line state changes in a Shareable memory region

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteClean	UD	UC	UC, SC	I, UC, SC
	SD	SC	SC	I, SC

Table D4-27 shows the expected cache line state changes for the WriteClean transaction in a Non-shareable memory region.

Table D4-27 Expected WriteClean cache line state changes in a Non-shareable memory region

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteClean	UD	UC	I, UC, SC	I, UC, SC
	SD	UC	I, UC, SC	I, UC, SC

D4.8.6 WriteEvict

A WriteEvict transaction can be used when evicting a Clean cache line. This transaction is used to write the line to a lower level of the cache hierarchy, such as an L3 or system level cache. A WriteEvict transaction is not required to update main memory.

A WriteEvict transaction must only be used in the following circumstances:

- When the cache line is held in a UniqueClean state.
- When the cache line has not been speculatively fetched from a different shareability domain.

Note

It is important that a cache line that could have been speculatively fetched, so that it was located outside of its shareability domain, could become out-of-date as the cache line is not required to be updated by subsequent stores to the cache line. If a cache line could be a stale copy, then it must not be written back into its shareability domain by the use of a WriteEvict transaction.

A WriteEvict transaction can be discarded.

A component can use the WriteEvict_Transaction property to declare if it supports WriteEvict transactions. Any master must permit the WriteEvict transaction to be disabled to ensure that the master operates correctly with any previous version of the ACE interface. If the WriteEvict_Transaction property is not declared, it is considered to be False.

Table D4-28 shows the expected cache line state changes for the WriteEvict transaction.

Table D4-28 Expected WriteEvict cache line state changes

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
WriteEvict	UC	I	I	I

D4.8.7 Restrictions on WriteUnique and WriteLineUnique usage

Typically, WriteUnique and WriteLineUnique transactions are used by a non-cached component that is writing to a Shareable region of memory. However, WriteUnique and WriteLineUnique transactions can be used by a cached component that meets the requirements.

A cached component must be able to complete any incoming snoop transaction while a WriteUnique or WriteLineUnique transaction is in progress. A cached component must:

- Complete any outstanding WriteBack, WriteClean, WriteEvict, or Evict transactions before issuing a WriteUnique or WriteLineUnique transaction.

  Note

  No additional WriteBack, WriteClean, WriteEvict, or Evict transactions can be issued until all outstanding WriteUnique or WriteLineUnique transactions are completed.

- Complete any incoming snoop transactions without the use of WriteBack, WriteClean, WriteEvict, or Evict transactions while a WriteUnique or WriteLineUnique transaction is in progress.

  Note

  WriteNoSnoop transactions can also be blocked behind WriteUnique and WriteLineUnique transactions. Therefore, the design of the master must ensure that an incoming snoop transaction can complete when a WriteNoSnoop transaction is blocked by an outstanding WriteUnique or WriteLineUnique transaction.

This is necessary, because earlier transactions that also might require earlier snoop transactions to complete, can prevent WriteUnique and WriteLineUnique transactions from progressing.

These requirements restrict the use of WriteUnique and WriteLineUnique transactions to components that can either:

- Complete all snoop transactions without requiring any data to be supplied, for example write-through caches that do not keep Dirty cache lines for Shareable data.
- Complete snoop transactions by using the snoop data channel, **CDDATA**.

D4.9 Evict transactions

This section defines the state changes associated with the Evict transaction group that are issued on the AW channel.

D4.9.1 Evict

An Evict transaction indicates that a cache line has been evicted from a master's local cache. There is no data transfer associated with an Evict transaction. An Evict transaction must be used only in a Shareable memory region.

Note

An Evict transaction is only used by a master that supports a snoop filter. When used, it is permitted, but not expected, for a master to evict a cache line without issuing an Evict transaction.

Table D4-29 shows the expected cache line state changes for the Evict transaction.

Table D4-29 Expected Evict cache line state changes in a Shareable memory region

Transaction	Start state	Expected end state	Legal end state	
			With Snoop Filter	No Snoop Filter
Evict	UC	I	I	Not used
	SC	I	I	Not used

D4.10 Handling overlapping write transactions

This section describes the expected behavior when two masters attempt stores to the same cache line in a Shareable region of memory at approximately the same time. When this happens, it is the responsibility of the interconnect to sequence the order that the transactions occur.

The master that gets sequenced first proceeds with the transaction as normal. However, the master that is sequenced second sees the transactions that are associated with the first master's store on its snoop port while attempting to carry out a store.

The following sections describe the expected behavior from the standpoint of the master that is sequenced second. For brevity, the master that is sequenced first is referred to as Master1 and the master that is sequenced second is referred to as Master2.

D4.10.1 Overlapping ReadUnique

If Master2 has issued a ReadUnique transaction because it required a copy of the data, the following occurs:

1. Master2 issues a ReadUnique transaction.
2. Master2 then sees one of the following transactions on its snoop port from Master1 attempting a write to the same line:
   - ReadUnique
   - CleanInvalid
   - MakeInvalid

   At this point, Master2 must invalidate any local copy that it has of the cache line. If Master2 does not have a local copy of the cache line, then no action is required.
3. When the ReadUnique completes, it returns with the updated copy of the cache line that includes the store that is performed by Master1.
4. Master2 can perform its store.

D4.10.2 Overlapping MakeUnique

If Master2 has issued a MakeUnique transaction because it was performing a full cache line write, the following occurs:

1. Master2 issues a MakeUnique transaction.
2. Master2 sees one of the following transactions on its snoop port from Master1 attempting a write to the same line:
   - ReadUnique
   - CleanInvalid
   - MakeInvalid

   At this point, Master2 must invalidate any local copy that it has of the cache line. If Master2 does not have a local copy of the cache line, then no action is required.
3. When the MakeUnique completes, Master2 can perform its full cache line store.

D4.10.3 Overlapping CleanUnique

If Master2 has issued a CleanUnique transaction because it was performing a partial line store but it already had a cached copy of the line, the following occurs:

1. Master2 issues a CleanUnique transaction.
2. Master2 sees one of the following transactions on its snoop port from Master1 attempting a write to the same line:
   - ReadUnique
   - CleanInvalid
   - MakeInvalid

   At this point, Master2 must respond to the snoop appropriately and then invalidate its local copy of the cache line.
3. When the CleanUnique completes, Master2 cannot perform its local store because it has lost its local copy of the cache line.
4. Master2 can issue a new ReadUnique transaction to obtain a copy of the line.
5. Master2 can perform its store.

A master can remove the need to issue a new ReadUnique the transaction, as described in the CleanUnique case, by initially issuing a ReadUnique transaction instead of a CleanUnique transaction. However, this sometimes results in a fetch from main memory occurring when it is not required.

Alternatively, a master can remove the need to issue a new ReadUnique transaction by performing a partial line WriteBack to main memory, that only updates the required bytes, when its CleanUnique transaction completes and the master has permission to store to the line. This does mean that the master does not retain a copy of the line.

It is acceptable for a master to use a CleanUnique transaction when carrying out a full cache line store. In this case, the master does not have to retry the transaction with a ReadUnique. It can simply perform the full cache line store when the CleanUnique is complete.

Chapter D5
Snoop Transactions

This chapter describes the snoop transactions that are seen on the snoop address channel. Both the required and protocol-recommended snoop transaction behaviors are described. It contains the following sections:

D5.1 Mapping coherency operations to snoop operations

This section describes the snoop transactions that are seen on the snoop address channel by a cached master that is being snooped by an initiating master.

When an initiating master issues a transaction, the interconnect is responsible for carrying out any snoop transactions that are required to complete the original transaction.

Not all transactions that are issued by an initiating master are permitted on the snoop address channel. Table D5-1 shows the protocol-recommended mappings between transactions that are issued by the initiating master and the snoop transactions that are seen on the snoop address channel by a cached master.

Table D5-1 Recommended transaction mappings

Transaction from initiating master	Transaction to snooped master
ReadNoSnoop	Not snooped
ReadOnce	ReadOnce
ReadClean	ReadClean
ReadNotSharedDirty	ReadNotSharedDirty
ReadShared	ReadShared
ReadUnique	ReadUnique
CleanUnique	CleanInvalid
MakeUnique	MakeInvalid
CleanShared	CleanShared
CleanInvalid	CleanInvalid
MakeInvalid	MakeInvalid
WriteNoSnoop	Not snooped
WriteUnique	CleanInvalid
WriteLineUnique	MakeInvalid
WriteBack	Not snooped
WriteClean	Not snooped
WriteEvict	Not snooped
Evict	Not snooped

Note

The interconnect can use other mappings that force the same cache line state changes in a snooped master. See *Alternative snoop transactions* on page D5-233.

D5.1.1 Permitted snoop transactions

Although the protocol does not require a fixed set of transaction mappings, the protocol does require that only the following defined subset of transactions is seen on the snoop address channel of a cached master:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique
- CleanInvalid
- MakeInvalid
- CleanShared

D5.1.2 Transactions not permitted as snoop transactions

The following transactions must not be seen on the snoop address channel of a cached master:

- ReadNoSnoop
- CleanUnique
- MakeUnique
- WriteNoSnoop
- WriteUnique
- WriteLineUnique
- WriteBack
- WriteClean
- WriteEvict
- Evict

D5.1.3 Alternative snoop transactions

Table D5-2 shows each permitted snoop transaction on the snoop address channel, the required cache line state change for the transaction, and the alternative snoop transaction that can be used. For completeness, the snoop transaction option column includes the original snoop transaction.

Table D5-2 Snoop transaction options on the address snoop channel

Snoop transaction	Required cache line state change	Snoop transaction option
ReadOnce	None	ReadOnce ReadClean, ReadNotSharedDirty, ReadShared ReadUnique, CleanInvalid CleanShared
ReadClean	Shared or Invalid	ReadClean, ReadNotSharedDirty, ReadShared ReadUnique, CleanInvalid
ReadNotSharedDirty	Shared or Invalid	ReadClean, ReadNotSharedDirty, ReadShared ReadUnique, CleanInvalid
ReadShared	Shared or Invalid	ReadClean, ReadNotSharedDirty, ReadShared ReadUnique, CleanInvalid
ReadUnique	Invalid	ReadUnique, CleanInvalid

Table D5-2 Snoop transaction options on the address snoop channel (continued)

Snoop transaction	Required cache line state change	Snoop transaction option
MakeInvalid	Invalid	ReadUnique, CleanInvalid, MakeInvalid
CleanInvalid	Invalid	ReadUnique, CleanInvalid
CleanShared	Clean or Invalid	ReadUnique, CleanInvalid CleanShared

Mapping to different snoop transactions can simplify the design of a snooped master. For example, a snooped master can handle all snoop transactions in the same way as a ReadUnique transaction. This is permitted because a ReadUnique transaction, as Table D5-2 on page D5-233 shows, is an alternative snoop transaction for all other snoop transactions.

D5.2 General requirements for snoop transactions

For each snoop transaction, the protocol specifies required and recommended behaviors.

Table D5-3 shows the required behavior for each snoop transaction:

Table D5-3 Required snoop transaction behavior

Snoop transaction	Must transfer data if Dirty	End state must be Shared or Invalid	End state must be Invalid	End state must be Clean or Invalid
ReadOnce	Yes	-	-	-
ReadClean	Yes	Yes	-	-
ReadNotSharedDirty	Yes	Yes	-	-
ReadShared	Yes	Yes	-	-
ReadUnique	Yes	-	Yes	-
CleanInvalid	Yes	-	Yes	-
MakeInvalid	-	-	Yes	-
CleanShared	Yes	-	-	Yes

Note

If a cache line is in the Dirty state and the associated cache does not assert the PassDirty snoop response, **CRRESP**[2], the cache line can remain in the Dirty state. If a cache line is in the Dirty state and the associated cache does assert the PassDirty snoop response, then the cache line must move to a Clean or Invalid state.

The cache line end states in Table D5-3 are classified as follows:

Shared or Invalid

The snooped cache must broadcast a transaction before it can perform a store to the cache line. That is, the snooped cache must consider that another master can hold a copy of the cache line.

Invalid

The snooped cache does not hold a copy of the line. This permits another agent to perform a store to the cache line.

Clean or Invalid

The snooped cache is not holding the cache line in a Dirty state. The snooped cache cannot perform a memory update, using a WriteBack or WriteClean transaction, until a later store to the cache line has occurred.

The following state changes must not occur due to a snoop transaction. A cache line must not move from:

- The Invalid state to any Valid state
- A Clean state to a Dirty state
- A Shared state to a Unique state
- The UniqueDirty state to the UniqueClean state

Note

A cache line must not move from the UniqueDirty state to the UniqueClean state. Such a transition would indicate that the interconnect has taken responsibility for writing back the cache line to main memory. Therefore, the cached master must not issue a WriteBack or WriteClean transaction without requesting permission to store to the cache line using an appropriate transaction.

The cache line end state that is permitted as a result of a snoop transaction is dependent on:

- The state of the cache line before the snoop
- The snoop transaction that is issued

Table D5-4 shows the permitted end states for valid combinations of the initial state and the issued snoop transaction. Combinations of initial state and end state that are not permitted as a result of a snoop transaction are excluded from Table D5-4.

A WriteBack, WriteClean, WriteEvict, or Evict transaction can occur while a snoop transaction is in progress. Table D5-4 does not include the state transition that can occur as a result of these write or evict transactions occurring. To understand such a scenario, the state transition for the WriteBack, WriteClean, WriteEvict, or Evict transaction must be applied, followed by the snoop transaction state transition that Table D5-4 shows.

The following abbreviations are used for the cache line states:

UC UniqueClean
UD UniqueDirty
SC SharedClean
SD SharedDirty
I Invalid

Table D5-4 Permitted end states for combinations of initial state and snoop transaction

Cache line state		Permitted for snoop transaction			
Initial	**End**	**ReadOnce**	**ReadClean ReadNotSharedDirty ReadShared**	**ReadUnique CleanInvalid MakeInvalid**	**Clean Shared**
I	I	Yes	Yes	Yes	Yes
UC	I	Yes	Yes	Yes	Yes
	UC	Yes	-	-	Yes
	SC	Yes	Yes	-	Yes
UD	I	Yes	Yes	Yes	Yes
	UD	Yes	-	-	-
	SC	Yes	Yes	-	Yes
	SD	Yes	Yes	-	-
SC	I	Yes	Yes	Yes	Yes
	SC	Yes	Yes	-	Yes
SD	I	Yes	Yes	Yes	Yes
	SC	Yes	Yes	-	Yes
	SD	Yes	Yes	-	-

The requirements for the IsShared and PassDirty snoop response bits are as follows:

- If the end state of the cache line is any Valid state, the IsShared snoop response bit must be asserted.
- If the cache line moves from a Dirty state to a Clean state, the PassDirty snoop response bit must be asserted.
- If the line moves from a Dirty state to the Invalid state as a result of any snoop transaction, except MakeInvalid, then the PassDirty snoop response bit must be asserted.
- If the cache line moves from a Dirty state to the Invalid state as a result of a MakeInvalid snoop transaction, then the PassDirty snoop response bit can be asserted or deasserted.

The permitted state changes in Table D5-4 on page D5-236 are combined with these requirements in Table D5-5 to show the permitted state changes and associated snoop response bits.

Table D5-5 Associated snoop responses for combinations of initial state and snoop transaction

Cache line state		Permitted for snoop transaction				Snoop Response	
Initial	**End**	**ReadOnce**	**ReadClean ReadNotSharedDirty ReadShared**	**ReadUnique CleanInvalid MakeInvalid**	**CleanShared**	**PassDirty**	**IsShared**
I	I	Yes	Yes	Yes	Yes	0	0
UC	I	Yes	Yes	Yes	Yes	0	0
	UC	Yes	-	-	Yes	0	1
	SC	Yes	Yes	-	Yes	0	1
UD	I	Yes	Yes	Yes	Yes	1[a]	0
	UD	Yes	-	-	-	0	1
	SC	Yes	Yes	-	Yes	1	1
	SD	Yes	Yes	-	-	0	1
SC	I	Yes	Yes	Yes	Yes	0	0
	SC	Yes	Yes	-	Yes	0	1
SD	I	Yes	Yes	Yes	Yes	1[a]	0
	SC	Yes	Yes	-	Yes	1	1
	SD	Yes	Yes	-	-	0	1

a. For a MakeInvalid snoop transaction, these PassDirty responses are also permitted to be 0.

D5.2.1 Channel activity

The required channel activity is the same for all snoop transactions:

- The address is received on the AC channel.
- The response is returned on the CR channel.
- The data is provided, if required, on the CD channel.

The DataTransfer snoop response bit **CRRESP**[0], indicates that a data transfer is required.

The snoop response on CR and the snoop data on CD must only be provided after the **ACVALID**/**ACREADY** handshake occurs.

D5.2.2 Snoop data transfers

A cached master can provide the data value of a cache line. The DataTransfer snoop response bit indicates that the data value of the cache line is to be transferred.

If a cached master receives a snoop transaction other than MakeInvalid for a cache line that is in a Dirty state, then the cached master must ensure that the data value is available so that the original transaction can complete. The cached master can ensure that the data value is available by:

- Returning the data when it completes the snoop transaction.
- Carrying out a memory update, using a WriteBack or WriteClean transaction, before responding to the snoop transaction.

Note

When a cached master holds a cache line in a Dirty state, the cache line might be the only up-to-date copy of that address location. Therefore, the data must be made available to any snoop transaction other than a MakeInvalid snoop transaction.

Typically, data is transferred for the following read snoop transactions:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique

Table D5-6 shows protocol-recommended data transfer behavior for snoop transactions, assuming that better system performance and lower power operation are achieved by providing data in response to these snoop transactions. This behavior is not mandatory, and alternative schemes can be implemented.

Table D5-6 Recommended data transfer behavior for snoop transactions

Snoop transaction	Data transfer if cache line is Clean	Data transfer if cache line is Dirty
ReadOnce	Yes	Yes
ReadClean	Yes	Yes
ReadNotSharedDirty	Yes	Yes
ReadShared	Yes	Yes
ReadUnique	Yes	Yes
CleanInvalid	No	Yes
MakeInvalid	No	No
CleanShared	No	Yes

D5.2.3 Memory update in progress

The protocol ensures that two components cannot update the same area of main memory at the same time.

If a snooped master receives a snoop transaction when it is updating main memory using either a WriteBack or WriteClean transaction, then it is the responsibility of the snooped master to ensure that no other master can update the same area of main memory at the same time. The snooped master achieves this by one of the following:

- Giving a snoop response with PassDirty deasserted and IsShared asserted, which does not pass permission to store to the line and does not pass responsibility for updating memory.
- Delaying the snoop response until the snooped master has completed the update to main memory.

When a snooped master is passing the permission to store to a cache line by sending a suitable snoop response, all write transactions to update main memory must have completed before the cycle in which the snoop response is given on the CR channel.

Note

If a snoop response is given while a memory update is in progress, the initial cache state must be considered prior to the memory update completing. This limits certain combinations of snoop response that can be given. For example, the response to a CleanShared snoop must have PassDirty asserted if the line is in a Dirty state. However, if it has a memory update in progress from that line, the master is not permitted to assert PassDirty. Therefore, the master must wait for the update to complete before responding to a CleanShared snoop.

D5.2.4 WasUnique snoop response

The WasUnique snoop response, **CRRESP**[4] indicates that the snooped cache line was held in a Unique state before the snoop transaction.

The WasUnique snoop response must be asserted only if the cache line was held in a Unique state. No other cache can have a copy of the cache line. A WasUnique response indicates that the interconnect does not have to carry out further snoop transactions to other cached masters because no other cache can hold a copy of the data.

A cached master does not have to generate the WasUnique response. The protocol permits **CRRESP**[4] to be fixed as deasserted. However, always deasserting WasUnique in this way can result in the cache line being provided to the initiating master as Shared, when it could have been provided as Unique. This might result in additional caches being snooped unnecessarily.

D5.2.5 Non-blocking requirements for a snooped master

The protocol defines rules for snooped masters and the interconnect to ensure transactions always progress through a system. The rules stipulate which transactions must always progress and which transactions can wait for others to complete.

The rules for the interconnect are defined in Chapter D6 *Interconnect Requirements*. See *Non-blocking requirements* on page D6-260.

The rules that apply to a cached master are:

- A master must complete any snoop transaction, to any address, before any of the following transactions, issued by the master, can be guaranteed to complete:
  - Any transaction, to any address, issued on the AR channel.
  - A WriteUnique or WriteLineUnique transaction, to any address, issued on the AW channel.
    See also *Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226.
- A master is permitted to wait for the following transactions to complete, to any address, before completing a snoop transaction:
  - WriteNoSnoop
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict
- If the response to a snoop transaction could result in the interconnect generating a write to main memory, or another master being given permission to write the cache line, then the master being snooped must complete any WriteBack, WriteClean, or WriteEvict transaction that is in progress for the cache line before it provides a response to the snoop transaction.

- A master must not wait for a WriteUnique or WriteLineUnique transaction to complete before completing a snoop transaction. If a snoop transaction is received by a master and a WriteUnique or WriteLineUnique transaction is in progress then the snoop transaction must be completed without the use of the AW and W channels.

  Note

  This requirement means that if a master has a WriteUnique or WriteLineUnique transaction in progress for any cache line that is in a Dirty state, and it receives a snoop transaction other than a MakeInvalid transaction, then it must return the data on the CD channel.

Figure D5-1 shows the non-blocking requirements.

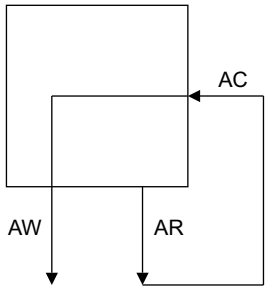


Figure D5-1 Required transaction channel ordering

In summary, the requirements are:

- Any transaction on the AR channel can be stalled waiting for a transaction on the AC channel.
- Any snoop transaction on the AC channel can be stalled waiting for a write transaction on the AW channel, except for a WriteUnique or WriteLineUnique transaction.

D5.3 Snoop transactions

This section describes each of the snoop transactions and provides information on the recommended behavior where options exist.

The following abbreviations are used for the cache line states:

UC UniqueClean
UD UniqueDirty
SC SharedClean
SD SharedDirty
I Invalid

D5.3.1 ReadOnce

Table D5-7 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for the ReadOnce snoop transaction.

Table D5-7 ReadOnce permitted cache line state changes

Cache line initial state	Cache line end state	Snoop Response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
	UC	0	1
	SC	0	1
UD	I	1	0
	UD	0	1
	SC	1	1
	SD	0	1
SC	I	0	0
	SC	0	1
SD	I	1	0
	SC	1	1
	SD	0	1

A ReadOnce snoop transaction is received by a snooped master when the initiating master indicates that it is not going to keep a cached copy of the cache line it is accessing. The ReadOnce snoop transaction enables the snooped master to:

- Keep the cache line in a Unique state.
- Carry out a later store to the cache line without issuing a transaction to obtain permission to store.

If the snooped master has a copy of the cache line, then this specification recommends that data is transferred. If the snooped master has the cache line in a Dirty state, then data must be transferred.

This specification recommends that the cache line is passed as Clean. Although it is permitted to pass the cache line as Dirty, this requires the interconnect to write the cache line back to main memory and the cache line to move to either the SharedClean or Invalid state.

Note

The IsShared snoop response must be asserted if the snooped master is retaining a copy of the cache line, even if the retained copy is in a Unique state.

D5.3.2 ReadClean, ReadShared, and ReadNotSharedDirty

The ReadClean, ReadShared, and ReadNotSharedDirty snoop transactions have the same requirements, but differ in the behavior that this specification recommends.

Table D5-8 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for these snoop transactions.

Table D5-8 ReadClean, ReadShared, and ReadNotSharedDirty permitted cache line state changes

Initial state	End state	Snoop Response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
	SC	0	1
UD	I	1	0
	SC	1	1
	SD	0	1
SC	I	0	0
	SC	0	1
SD	I	1	0
	SC	1	1
	SD	0	1

If the cached line being snooped is part of an exclusive sequence, then the cache line must remain valid in the snooped master.

If data is available, this specification recommends that the data is transferred.

ReadClean

For a ReadClean snoop transaction, if the responsibility for writing the cache line back to main memory is being passed to the interconnect, as indicated by the PassDirty snoop response being asserted, the cache line is written back to main memory immediately. This specification recommends that the cache line remains Dirty in the snooped cache.

ReadShared

For a ReadShared snoop transaction, if the responsibility for writing the cache line back to main memory is being passed to the initiating master, then it is accepted by the master. The decision to pass responsibility for writing the cache line that is Dirty back to main memory depends on which master accesses the cache line next:

- If the snooped master is likely to be the next master to store to the cache line, then this specification recommends that the cache line remains Dirty in the snooped cache but is passed as Clean to the initiating master.

- If the initiating master is likely to be the next master to store to the cache line, then this specification recommends that the cache line is passed to the initiating master as Dirty. In this case:
  - If it is likely that the initiating master carries out a store before the snooped master next loads from the cache line, then this specification recommends that the snooped master does not retain a cached copy.
  - If it is likely that the snooped cache loads from the cache line before the initiating master performs a store, then this specification recommends that the snooped master does retain a copy of the cache line.
- If it is not known whether the initiating master or the snooped master is the next to store to the cache line, then this specification recommends that the cache line is held as Dirty in the cache that is least likely to evict the cache line. Typically, this would be the initiating master, because this is the master that has most recently accessed the cache line.
- If the snooped master does not support all five cache states, then fewer options are available.

If information on the access patterns for a cache line is not available, then this specification recommends that the cache line is passed as Dirty to the initiating master and moves to the SharedClean state in the snooped cache.

ReadNotSharedDirty

If responsibility for updating main memory is passed to the initiating master, it is only accepted if the cache line moves to the Invalid state in the snooped cache. The decision to pass responsibility for writing the cache line back to main memory depends on which master accesses the cache line next:

- If the snooped master is likely to be the next master to store to the cache line, then this specification recommends that the cache line remains Dirty in the snooped cache but is passed as Clean to the initiating master.
- If the initiating master is likely to be the next master to store to the cache line, then this specification recommends that the cache line is passed to the initiating master as Dirty and the cache line is removed from the snooped cache.

If it is not known which master accesses the cache line next, then no recommendations are provided by this specification.

D5.3.3 ReadUnique

Table D5-9 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for the ReadUnique snoop transaction.

Table D5-9 ReadUnique permitted cache line state changes

Initial state	End state	Snoop response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
UD	I	1	0
SC	I	0	0
SD	I	1	0

For a ReadUnique transaction, if the snooped cache holds a copy of the cache line in a Dirty state, then the data must be transferred.

This specification recommends that data is also transferred if the cache line is in a Clean state.

D5.3.4 CleanInvalid

Table D5-10 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for the CleanInvalid snoop transaction.

Table D5-10 CleanInvalid permitted cache line state changes

Initial state	End state	Snoop response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
UD	I	1	0
SC	I	0	0
SD	I	1	0

For a CleanInvalid transaction, if the snooped cache holds a copy of the cache line in a Dirty state, then the data must be transferred.

This specification recommends that data is not transferred if the cache line is in a Clean state.

D5.3.5 MakeInvalid

Table D5-11 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for the MakeInvalid snoop transaction.

Table D5-11 MakeInvalid permitted cache line state changes

Initial state	End state	Snoop response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
UD	I	0	0
		1	0
SC	I	0	0
SD	I	0	0
		1	0

For a MakeInvalid transaction, this specification recommends that the data is not transferred.

Note

If data is not transferred, as indicated by the DataTransfer snoop response, **CRRESP**[0] being deasserted, then the PassDirty snoop response bit must also be deasserted.

D5.3.6 CleanShared

Table D5-12 shows all the permitted cache line state changes and the associated PassDirty and IsShared snoop responses for the CleanShared snoop transaction.

Table D5-12 CleanShared permitted cache line state changes

Initial state	End state	Snoop response	
		PassDirty, CRRESP[2]	IsShared, CRRESP[3]
I	I	0	0
UC	I	0	0
	UC	0	1
	SC	0	1
UD	I	1	0
	SC	1	1
SC	I	0	0
	SC	0	1
SD	I	1	0
	SC	1	1

For a CleanShared transaction, if the snooped cache holds a copy of the cache line in a Dirty state, then the data must be transferred.

This specification recommends that the data is not transferred if the cache line is in a Clean state.

Note

The IsShared snoop response must be asserted if the snooped master is retaining a copy of the cache line, even if the retained copy is in a Unique state.

Chapter D6
Interconnect Requirements

This chapter describes the interconnect requirements for ACE. It contains the following sections:

This chapter does not describe the interconnect requirements for barriers or DVM operations. See Chapter D8 *Barrier Transactions* and Chapter D13 *Distributed Virtual Memory Transactions* for these requirements.

D6.1 About the interconnect requirements

It is the responsibility of the interconnect to:

- Receive transactions from an initiating master.
- Determine the order of transactions when multiple transactions are received at the same time.
- Issue snoop transactions, as required, for each transaction from an initiating master.
- Receive snoop responses and data, when data is provided, from a snooped master.
- Generate the response for the initiating master.
- Carry out any required access to main memory.

D6.2 Sequencing transactions

Many masters might issue transactions at the same time. The protocol permits each master to make multiple outstanding requests, and to receive multiple outstanding snoop transactions.

It is the responsibility of the interconnect to ensure that there is a defined order in which transactions to the same cache line can occur, and that the defined order is the same for all components. In the case of two masters issuing transactions to the same cache line at approximately the same time, then the interconnect determines which of the transactions is sequenced first. The arbitration method that is used by the interconnect is not defined by the protocol.

The interconnect indicates the order of transactions to the same cache line by sequencing transaction responses and snoop transactions to the masters. The ordering rules are:

- If a master issues a Coherent or Cache Maintenance transaction to a cache line and it receives a snoop transaction to the same cache line before it receives a response to the transaction it has issued, then the snoop transaction is defined as ordered first.
- If a master issues a Coherent or Cache Maintenance transaction to a cache line and it receives a response to the transaction before it receives a snoop transaction to the same cache line, then the transaction that is issued by the master is defined as ordered first.

Note

The relative ordering of transaction responses and snoop transactions only applies to transactions to the same cache line.

The interconnect must ensure the following:

- If the interconnect provides a master with a response to a Coherent or Cache Maintenance transaction, it must not send that master a snoop transaction to the same cache line before it has received the associated **RACK** or **WACK** response from that master.
- If the interconnect sends a snoop transaction to a master, it must not provide that master with a response to a Coherent or Cache Maintenance transaction to the same cache line before it has received the associated **CRRESP** response from that master.

Figure D6-1 shows that from the point that a master starts to receive a transaction response, it is guaranteed not to receive a snoop transaction to the same cache line until it has asserted the acknowledge signal, indicating that the transaction has completed.

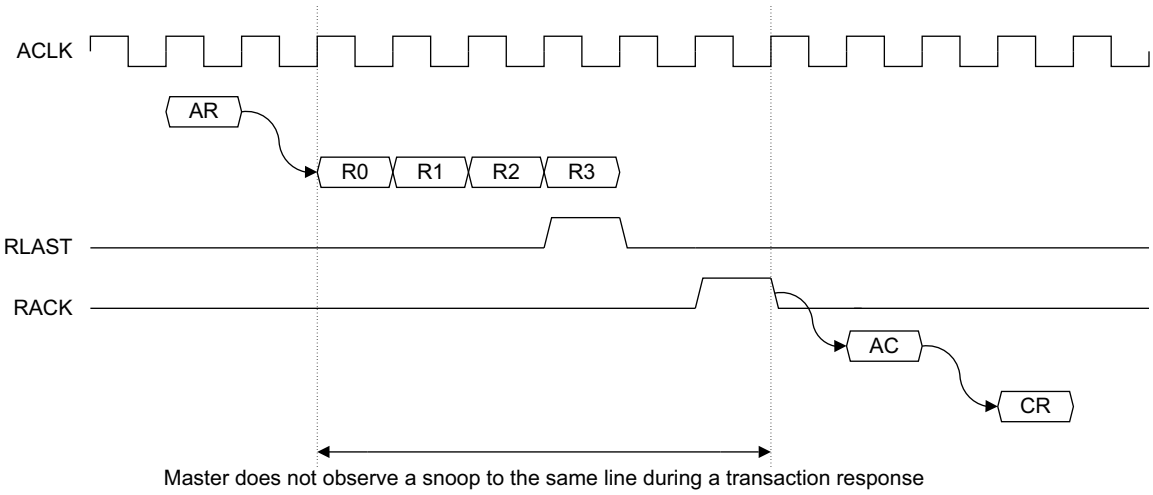


Figure D6-1 Transaction response before a snoop transaction

The diagram in Figure D6-2 shows that if a master receives a snoop transaction to a cache line to which it has issued a transaction, but has not yet received a transaction response, then it is guaranteed not to see a transaction response until it has provided a snoop response.

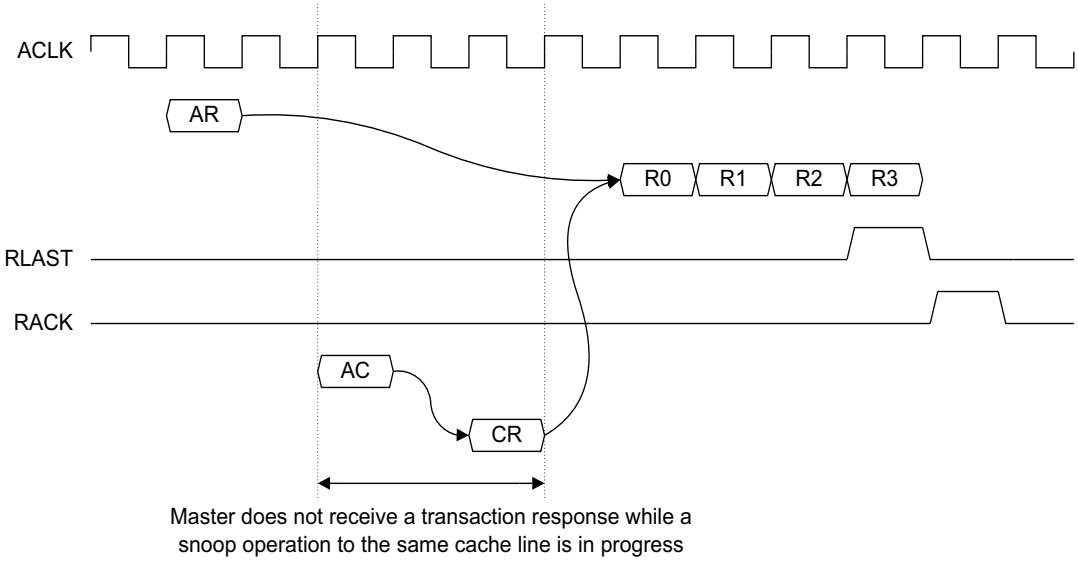


Figure D6-2 Snoop transaction before a transaction response

D6.2.1 Read and Write Acknowledge

The read and write acknowledge signals are required to ensure correct operation where there is a delay between an interconnect and the master completing a transaction. For example, this can occur when a register or clock domain boundary exists between the interconnect and the master.

The master provides the read acknowledge signal **RACK** and the write acknowledge signal **WACK** to guarantee that the interconnect can determine when a transaction has completed at the master.

The master sends the **RACK** and **WACK** signals for all transactions, not only Shareable transactions. This permits the signals to be generated using only the handshake signals on the read data channel or write response channel respectively.

The master must only send a read acknowledge after the last read data transfer in which **RLAST** is asserted. See *Read acknowledge signaling* on page D3-189.

The master must only send a write acknowledge after the write response handshake has occurred. See *Write Acknowledge signaling* on page D3-191.

There is no mechanism to stall the **RACK** or **WACK** signal. The interconnect is required to accept the acknowledge in the same cycle as the master asserted the read or write acknowledge.

D6.2.2 Continuous read data return

To specify that a system provides continuous read data return, a Continuous_Cache_Line_Read_Data property is defined that can be True or False for an ACE interconnect.

True An ACE interconnect is declared as having property Continuous_Cache_Line_Read_Data.

False Interconnect does not support the Continuous_Cache_Line_Read_Data property. If not declared, the property is considered to be False.

A master is defined as requiring this property if it requires that when the first data beat of a cache line read is returned, then all subsequent data transfers for that cache line are returned without requiring progress on any snoop transaction.

Note

A master that requires the Continuous_Cache_Line_Read_Data property is not required to make forward progress on new snoop transactions between the return of the first and last read data beats for a cache line transaction it has issued. However, if the master has already started responding to a snoop transaction, and has returned at least one beat of snoop data, then it must return all the remaining beats of snoop data for a single snoop transaction that it is responding to.

An interconnect is defined as supporting this property if it is guaranteed that once the first data beat of a cache line read is returned, then all subsequent data transfers for that cache line are returned without requiring forward progress on any snoop transaction.

This property is only required for transaction types that are precisely a cache line size:

- All ReadClean, ReadShared, ReadUnique, and ReadNotSharedDirty transactions.
- ReadOnce transactions that are precisely a cache line size.
- ReadNoSnoop transactions that are Non-shareable, WriteThrough or WriteBack Cacheable, and are precisely a cache line size.

Note

This specification recommends this behavior for all new designs.

D6.3 Issuing snoop transactions

It is the responsibility of the interconnect to generate the snoop transactions that are required to progress a transaction from an initiating master.

The transaction from the initiating master determines which cached masters in the shareability domain must be snooped:

- The following transactions do not cause a snoop of any cached masters:
  - ReadNoSnoop
  - WriteNoSnoop
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict
- The following transactions must cause a snoop of the cached masters that can hold a copy of the cache line:
  - ReadOnce
  - ReadClean
  - ReadNotSharedDirty
  - ReadShared

  Snooping of the cached masters must continue until any one of the following occurs:
  - A copy of the line is obtained.
  - A snoop response is received with WasUnique, **CRRESP**[4], asserted.
  - All caches have been snooped.
- The CleanShared transaction must cause a snoop of the cached masters that can hold a copy of the cache line until any one of the following occurs:
  - A Dirty copy of the line is obtained, as indicated by snoop response PassDirty, **CRRESP**[2], being asserted.
  - A snoop response is received with WasUnique, **CRRESP**[4], asserted.
  - All caches have been snooped.
- The following transactions must cause a snoop of the cached masters that can hold a copy of the cache line:
  - ReadUnique
  - CleanUnique
  - MakeUnique
  - CleanInvalid
  - MakeInvalid
  - WriteUnique
  - WriteLineUnique

  Snooping of the cached masters must continue until either of the following occurs:
  - A snoop response is received with WasUnique, **CRRESP**[4], asserted.
  - All caches have been snooped.

Note

The interconnect must not issue a snoop transaction to the initiating master.

Table D6-1 on page D6-253 shows for each transaction that is issued by the initiating master:

- The snooped cache line state change that must be ensured by the interconnect.
- The snoop transaction that this specification recommends the interconnect to use.
- The optional snoop transactions that the interconnect can use.

In Table D6-1, the snoop transaction that this specification recommends is also included as an optional snoop transaction.

Table D6-1 Interconnect snoop requirements

Transaction from Initiating Master	State change for the snooped cache	Recommended Snoop transaction	Optional Snoop transaction
ReadNoSnoop	None	-	-
ReadOnce	None	ReadOnce	ReadOnce, ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique, CleanInvalid, CleanShared
ReadClean	Shared or Invalid	ReadClean	ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique, CleanInvalid
ReadNotSharedDirty	Shared or Invalid	ReadNotSharedDirty	ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique, CleanInvalid
ReadShared	Shared or Invalid	ReadShared	ReadClean, ReadNotSharedDirty, ReadShared, ReadUnique, CleanInvalid
ReadUnique	Invalid	ReadUnique	ReadUnique, CleanInvalid[a]
CleanUnique	Invalid	CleanInvalid	ReadUnique, CleanInvalid[a]
MakeUnique	Invalid	MakeInvalid	ReadUnique, CleanInvalid MakeInvalid
CleanShared	Clean or Invalid	CleanShared	ReadUnique, CleanInvalid CleanShared
CleanInvalid	Invalid	CleanInvalid	ReadUnique, CleanInvalid[a]
MakeInvalid	Invalid	MakeInvalid	ReadUnique, CleanInvalid MakeInvalid
WriteNoSnoop	None	-	-
WriteUnique	Invalid	CleanInvalid	ReadUnique, CleanInvalid[a]
WriteLineUnique	Invalid	MakeInvalid	ReadUnique, CleanInvalid MakeInvalid
WriteBack	None	-	-
WriteClean	None	-	-
WriteEvict	None	-	-
Evict	None	-	-

a. Other optional snoop transactions can be used if the cached masters in the same shareability domain are not all snooped at the same time.

The interconnect is not required to snoop all caches at the same time, caches can be snooped sequentially.

When the interconnect is snooping multiple cached masters, it is not required to snoop all the cached masters in an identical manner.

If the interconnect is carrying out the snoop transactions sequentially, issuing some snoop transactions after other snoop transactions for the same cache line have completed, then after a snoop response is received with PassDirty asserted, it is permitted to use the MakeInvalid snoop transaction for the remaining cached masters that are still to be snooped. The transactions that can benefit from this use of the MakeInvalid snoop transaction are:

- WriteUnique
- ReadUnique
- CleanUnique
- CleanInvalid

D6.4 Transaction responses from the interconnect

The interconnect must provide a response for all transactions from an initiating master.

Table D6-2 shows the permitted response from the interconnect for a transaction that is issued on the AR channel.

Table D6-2 Permitted interconnect response for a transaction on the AR channel

Transaction from initiating master	Permitted response from the interconnect	
	IsShared, RRESP[3]	PassDirty, RRESP[2]
ReadNoSnoop	0	0
ReadOnce	0	0
	1	0
ReadClean	0	0
	1	0
ReadNotSharedDirty	0	0
	0	1
	1	0
ReadShared	0	0
	0	1
	1	0
	1	1
ReadUnique	0	0
	0	1
CleanUnique	0	0
MakeUnique	0	0
CleanShared	0	0
	1	0
CleanInvalid	0	0
MakeInvalid	0	0

The interconnect must determine the IsShared response for the following transactions from the initiating master:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- CleanShared

After snooping all the required cached masters, the IsShared response to the initiating master for these transactions is determined as follows:

- If WasUnique was asserted for any snoop response received by the interconnect then:
  - If IsShared was asserted for that snoop response then, IsShared must be asserted in the transaction response to the initiating master.
  - If IsShared was deasserted for that snoop response then, this specification recommends that IsShared is deasserted in the transaction response to the initiating master. However, it is permitted to assert IsShared in the transaction response to the initiating master.
- If WasUnique was not asserted for any snoop response received by the interconnect then:
  - If any snoop responses had IsShared asserted then, IsShared must be asserted in the transaction response to the initiating master.
  - If all snoop responses received by the interconnect had IsShared and DataTransfer deasserted then, IsShared must be deasserted in the transaction response to the initiating master.
  - If all snoop responses received by the interconnect had IsShared deasserted and any snoop response had DataTransfer asserted then, this specification recommends that IsShared is deasserted in the transaction response to the initiating master. However, it is permitted to assert IsShared in the transaction response to the initiating master.

The interconnect must determine the PassDirty response to the initiating master for the following transactions:

- ReadNotSharedDirty
- ReadShared
- ReadUnique

After snooping all the required cached masters, the PassDirty response to the initiating master for these transactions is determined as follows:

- If PassDirty was asserted for any snoop response that is received by the interconnect, and the interconnect has not generated a write transaction to update main memory, then PassDirty must be asserted in the transaction response to the initiating master.

Note

Only transactions that are initiated on the AR channel have additional response bits returned with the transaction response to the initiating master on the R channel.

Write transactions do not have additional response bits. The response from the transaction that passes through the interconnect can be returned directly to the initiating master. The write transactions are:

- WriteNoSnoop
- WriteUnique
- WriteLineUnique
- WriteBack
- WriteClean
- WriteEvict

An Evict transaction does not propagate downstream and the interconnect is required to generate an OKAY, **BRESP**[1:0] = `0b00` write response.

D6.5 Interactions with main memory

This section describes the circumstances in which the interconnect:

- Must read or update main memory directly.
- Can pass permission to update main memory to a master.

It contains to following sections:

D6.5.1 Interconnect read from main memory or peripheral device

The interconnect must always read from main memory, or the appropriate peripheral device, for a ReadNoSnoop transaction.

If the interconnect has not obtained the required data from a snoop transaction, the interconnect must read from main memory to complete the following transactions:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared
- ReadUnique

An interconnect is permitted to read from main memory before all snoop transactions have completed. However, the following rules apply:

- Data obtained from main memory must not be used if any cache in the shareability domain of the master is holding a Dirty copy of the cache line. Therefore, if the cache line is provided by a snoop transaction, then data that is obtained from main memory must not be used.

  Note

  The snoop response does not indicate if a cache is holding a Dirty copy of the cache line. It only indicates that the responsibility for updating main memory is being passed.

- Data read from main memory must not be used if it is possible that the data read is different to the data that would be read after all associated snoop transactions have completed. For example, if a WriteBack or WriteClean transaction to the cache line did not complete before the read from main memory was issued, then the data that is obtained from main memory must not be used. A new read from main memory must be issued to obtain the correct data.

D6.5.2 Main memory update that is generated by the interconnect

The following transactions are passed through the interconnect to the appropriate main memory or peripheral device:

- WriteNoSnoop
- WriteBack
- WriteClean

For the WriteUnique and WriteLineUnique transactions, the interconnect must carry out the required snoop transactions as described in *Issuing snoop transactions* on page D6-252. If a snoop response is received with PassDirty asserted, then the final value in memory must be the same as if the memory is updated with the original Dirty cache line first and then the write data that is part of the WriteUnique or WriteLineUnique transactions.

Examples of how this can be achieved are:

- The order in which data is written is:
  1. The Dirty cache line that is obtained from the snoop is written to main memory.
  2. The write data that is part of the WriteUnique or WriteLineUnique transaction is written to main memory.
- The write data that is part of the WriteUnique or WriteLineUnique transaction must be merged with the data that is obtained from the Dirty cache line. The valid bytes of the WriteUnique or WriteLineUnique transaction must overwrite the associated bytes of the Dirty cache line. A single write to main memory is then performed of the merged data.

When a snoop response has the PassDirty response asserted, and the interconnect does not assert the PassDirty transaction response for the initiating master, the interconnect must generate a write transaction to update main memory. This occurs when:

- The transaction from the initiating master does not permit the assertion of the PassDirty response bit. This is true for the following transactions:
  - ReadOnce
  - ReadClean
  - CleanUnique
  - CleanShared
  - CleanInvalid
  - ReadNotSharedDirty, if the IsShared response is asserted
- The interconnect has provided a read response to an initiating master before it has received all the snoop responses and a later snoop response has PassDirty asserted.

The interconnect is permitted to carry out a write transaction to update main memory when it receives a snoop response with the PassDirty response asserted. In this case, it must not assert the PassDirty transaction response for the initiating master.

If it does not receive a snoop response with the PassDirty response asserted, then the interconnect must not carry out a write to update main memory.

This specification recommends that the interconnect does not carry out a write transaction to update main memory, unless required by the combination of the initiating master transaction type and the received snoop response.

D6.5.3 Permission to update main memory

The interconnect must ensure that all updates to main memory, both from cached masters and the interconnect itself, are performed in the correct order. The interconnect must only give a cached master permission to update main memory when it is guaranteed that any earlier updates to main memory are ordered.

Permission to update main memory is given to a master by either:

- Giving a transaction response to the master with the PassDirty response asserted.
- Giving a transaction response to the master with the IsShared response deasserted. This gives the master permission to store to the cache line and therefore permission to carry out a write to update main memory.

When a master is given permission to update main memory, the first point at which the master can start the associated write transaction is the cycle after the **RVALID**/**RREADY** handshake in which **RLAST** is asserted for the transaction that gave permission to update main memory.

D6.6 Other requirements

This section describes other requirements that apply to the interconnect. It contains the following sections:

- *Non-blocking requirements*
- *Permitted transaction modifications* on page D6-261
- *Speculative reads* on page D6-261

D6.6.1 Non-blocking requirements

To ensure transactions always progress through a system, the protocol defines rules for snooped masters and the interconnect. The rules stipulate which transactions must always progress and which transactions can wait for others to complete.

The rules for snooped masters are defined in Chapter D5 *Snoop Transactions*. See *Non-blocking requirements for a snooped master* on page D5-239.

To ensure transactions always progress through a system, the following rules apply for an interconnect:

- The following transactions must progress to any address without requiring any pending snoop transactions to progress:
  - WriteNoSnoop
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict

  Note

  None of these transactions require an associated snoop transaction.

- An interconnect is permitted to wait for a snoop transaction to complete before it progresses the following transactions:
  - Any transaction to any address issued on the AR channel
  - WriteUnique or WriteLineUnique transactions to any address issued on the AW channel

    Note

    See also *Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226.

The diagram in Figure D6-3 shows the non-blocking requirements.

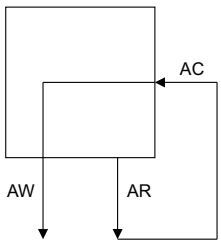


Figure D6-3 Required non-blocking transaction channel ordering

In summary, the requirements are:

- Any transaction on the AR channel can be stalled waiting for a transaction on the AC channel.
- Any snoop transaction on the AC channel can be stalled waiting for a write transaction on the AW channel, except for a WriteUnique or WriteLineUnique transaction.

D6.6.2 Permitted transaction modifications

An interconnect is permitted to modify transactions as defined by the Modifiable attribute, **AxCACHE**[1], in *Modifiable transactions* on page A4-65:

- A transaction can be broken into multiple transactions.
- Multiple transactions can be merged into a single transaction.
- A read transaction can fetch more data than required.
- A write transaction can access a larger address range than required, making use of Write strobes to ensure that only the required memory locations are updated.
- In each generated transaction, the following signals can be modified:
  - The transfer address, **AxADDR**
  - The burst size, **AxSIZE**
  - The burst length, **AxLEN**
  - The burst type, **AxBURST**

Note

A modification to a transaction by the interconnect is not seen by any master in the system.

D6.6.3 Speculative reads

A master in the ACE protocol is permitted to carry out a read of a cache line that it already holds in its cache. This is referred to as a Speculative Read.

A master issuing a speculative read must ensure that:

- The transaction uses the correct shareability and cacheability attributes for the address location.
- It uses its cached version of the data and not the data that is returned by the speculative read.

  Note

  It is required that a master uses its cached version because this could be in the Dirty state and therefore no other valid copies of the cache line exist.

An interconnect must consider that a master might be carrying out a speculative read, as it is not explicit in the transaction. The interconnect must ensure that it does not use data that is obtained by a speculative read to service another transaction.

D6.7 Interoperability considerations

A system wide coherency protocol has to work correctly with components that might have:

- Different structures for caching and storing data
- Different cache line sizes
- Different physical address space sizes

D6.7.1 Cache Line size conversions

Maximum performance and efficiency is usually achieved when all components use the same cache line size. For systems where this is not possible, it is the responsibility of the interconnect to convert between the different cache line sizes.

Note

The supported cache line sizes and maximum physical address space size are defined at design time.

Narrow to wide conversion

When the master initiating a transaction has a narrow cache line, the following conversion is required:

- A read transaction can be converted to a wider cache line size. The transactions that can be converted are:
  - ReadOnce
  - ReadClean
  - ReadNotSharedDirty
  - ReadShared
  - ReadUnique

  The converted transaction fetches the data that is required to complete the original transaction together with data that is not required. The excess data must be written back to main memory if it is Dirty, but can be discarded if it is Clean.
- A clean transaction can be converted to a wider cache line size. The transactions that can be converted are:
  - CleanUnique
  - CleanShared
  - CleanInvalid

  Dirty data obtained as a result of the clean transaction must be written back to main memory.
- The MakeUnique or MakeInvalid transactions require special consideration. A cache line that is wider than that requested by a master with a narrow cache line cannot be invalidated. When converting a MakeUnique or MakeInvalid transaction to a wider cache line size, it must be converted to a CleanInvalid transaction. This ensures that all dirty data is written back to main memory before the wider line is invalidated.

  Note

  Similar consideration is required for the WriteUnique or WriteLineUnique transaction.

Wide to narrow conversion

When the master initiating a transaction has a wide cache line, the transaction can be broken into multiple narrow transactions.

Each of these narrow transactions can be responded to by different cached masters during the snoop process and some of the narrow transactions might require access to main memory.

It is the responsibility of the interconnect to:

- Assemble the transaction response sent to the originating master.
- Ensure that the multiple narrow transactions are sequenced correctly, that is, as a contiguous block with respect to other snoop transactions.

If any part of the wide cache line is shared, then the whole cache line must be considered as shared. If any part of the wide cache line is Dirty, then the whole cache line must be considered to be Dirty.

The passing of dirty data from a snooped master is optional for the following transactions:

- ReadOnce
- ReadClean
- ReadNotSharedDirty
- ReadShared

It is the responsibility of the interconnect to ensure that no parts of the cache line can be Dirty in more than one cache.

D6.7.2 Additional Cache Line conversion considerations

The following transactions, issued by a master, are not required to be a full cache line size:

- ReadNoSnoop
- ReadOnce
- WriteNoSnoop
- WriteUnique
- WriteBack
- WriteClean

All other transactions are required to be a full cache line size and must use the full width of the data bus.

As a snoop transaction is required to be a full cache line size, it is the responsibility of the interconnect to carry out the required size translation. Size translation is required for:

- A ReadOnce transaction that is converted into a ReadOnce snoop.
- A WriteUnique transaction that is converted into a CleanInvalid snoop.

D6.7.3 Address space size

The protocol supports communication between components that have different physical address space sizes.

Components with different physical address space sizes must communicate as follows:

- The component with the smaller physical address space must be positioned within an aligned window in the larger physical address space. Typically, the window is located at the bottom of the larger physical address space. However, it is acceptable for the component with the smaller physical address space to be positioned in an offset window within the larger physical address space.
- An outgoing transaction must have the required additional higher-order bits added to the transaction address.
- An incoming transaction must be examined so that:
  - A transaction that is within the address window has the higher-order address bits removed and is passed through.
  - A transaction that does not have the required higher-order address bits is suppressed.

Note

It is the responsibility of the interconnect to provide the required functionality.

Chapter D7
Cache Maintenance

This chapter describes *cache maintenance operations* (CMOs) that assist with software cache management.

It contains the following sections:

D7.1 Cache Maintenance Operations

There are two general types of CMO:

Cache cleaning

Ensures that a store to a cache line is made visible to non-coherent agents by updating main memory with the value that is held in a Dirty cache line.

Cache invalidation

Ensures that a subsequent load from a location does not use a cached copy. Any accesses to that location will be to main memory. After a line is invalidated from all caches, it can be updated by coherent or non-coherent agents.

D7.2 CMO transactions

The specification supports the following transactions for cache maintenance:

CleanShared When completed, all cached copies of the addressed line in the specified domain are Clean and any associated writes are observable.

CleanInvalid When completed, all cached copies of the addressed line in the specified domain are invalidated, having been written to memory if they were Dirty. Any associated writes are observable.

MakeInvalid When completed, all cached copies of the addressed line in the specified domain are invalidated. Dirty data might have been discarded.

D7.3 Actions on receiving a CMO

When a component receives a CMO, it must do the following:

1. If the component is a cache and the CMO is cacheable, it must look up the line.
2. If the component is a coherent interconnect and the CMO is Inner or Outer Shareable, a CMO snoop must be sent to any cache that might have the line:
   - Allocated, for an Invalidate CMO
   - Dirty, for a Clean CMO
3. Write back any dirty data that is found in the cache or peer caches. This specification recommends that Write-Through No-Allocate is used for writes to memory which will be followed by a CMO to the same line. This ensures that the line will be looked up in any downstream cache but will not be allocated.
4. Wait for all snoops and associated writes to receive a response.
5. If the CMO does not need to be sent downstream, the component can issue a response to the CMO.
6. If the CMO does need to be sent downstream, the CMO must be sent and the response that is returned must be propagated when it is received from downstream.

D7.4 Cache maintenance transaction attributes

The following rules apply to CMO transactions:

- CMOs must be cache line sized and Regular. See *Regular transactions* on page A3-53 for more details.
- The domain can be Non-shareable, Inner Shareable or Outer Shareable.
  - Inner Shareable is supported for legacy reasons and is not recommended for new designs.
  - System Shareable is not permitted, which means that CMO transactions must be Normal rather than Device.

Table D7-1 is a summary of which caches must action CMOs, based on the **AxCACHE** and **AxDOMAIN** attributes:

Table D7-1 Caches that must action CMOs

AxCACHE	AxDOMAIN	CMO applies to
Device	System Shareable	N/A (not legal for CMOs)
Non-cacheable	Non-shareable	No caches
	Inner Shareable	Inner-domain peer caches
	Outer Shareable	Inner and outer-domain peer caches
Cacheable	Non-shareable	In-line caches
	Inner Shareable	Inner-domain peer caches, in-line caches
	Outer Shareable	Inner and outer-domain peer caches, in-line caches

To maintain coherency, the following recommendations apply to CMOs and non-CMOs:

- If a location is cacheable for non-CMO transactions, it should be cacheable for CMO transactions.
- If a location is in the Inner or Outer Shareable domain for non-CMO transactions, it should be in the same domain for CMO transactions.
- If a location is in the Non-shareable domain for non-CMO transactions, it can be in the Non-shareable, Inner Shareable, or Outer Shareable domain for CMO transactions.
- A master should not issue a read request that permits it to allocate a line, while there is an outstanding CMO to that line.
- Allocation hints, such as **AxCACHE[3:2]**, are not required to match between CMO and non-CMO transactions to the same cache line.

D7.5 Cache maintenance propagation

The propagation of CMOs downstream of components, depends on the system topology. A CMO must be propagated downstream if the CMO is cacheable and there is a downstream cache which might have allocated the line and there is an observer downstream of that cache.

The mechanism for controlling whether a component propagates CMOs is IMPLEMENTATION DEFINED, options include:

- Using the interface control signals that are described in Chapter D12 *Interface Control*.
- Changing interface properties, where these are configurable. For example, a cache slave interface can be configured to accept CMOs, but the master interface might be configured to not issue CMOs.

D7.6 CMO signaling

CMOs can be transported on either the read or write channels.

Two properties are used to determine which channels are used:

- CMO_On_Read
- CMO_On_Write

Transporting CMOs on read channels is included in this specification to support legacy components. This specification recommends that new designs transmit CMOs on the write channels.

D7.6.1 Cache maintenance on read channels

The CMO_On_Read property is used to indicate whether an interface supports CMOs on the read channels:

True CMOs are supported on the AR and R channels. This is consistent with this specification up to and including Issue F.

If CMO_On_Read is not declared, it is considered True.

False CMOs are not supported on the AR and R channels. In this case, they are either signaled on the write channels or not used by this interface.

The CMO_On_Read property can be set True for the following interfaces:

- ACE5
- ACE5-Lite
- ACE5-LiteDVM

A CMO transaction on the read channels consists of a request on the AR channel and a single beat response on the R channel. The response indicates that the CMO is observed, and all cache lines have been cleaned and invalidated if necessary.

Table D7-2 shows the **ARSNOOP** encodings that are used to signal CMO requests on the AR channel.

Table D7-2 ARSNOOP encodings

ARSNOOP	Operation
0b1000	CleanShared
0b1001	CleanInvalid
0b1101	MakeInvalid

D7.6.2 Cache maintenance on write channels

The CMO_On_Write property is used to indicate whether an interface supports CMOs on the write channels:

True CMOs are supported on the AW and B channels.

False CMOs are not supported on the AW and B channels. In this case, they are either signaled on the read channels or not used by this interface.

If CMO_On_Write is not declared, it is considered False.

The CMO_On_Write property can be set True for the following interfaces:

- ACE5-Lite
- ACE5-LiteDVM

The following CMOs can be sent on write channels:

- CleanInvalid
- CleanShared

MakeInvalid is not supported on the write channels.

Table D7-3 shows **AWSNOOP** encoding permissible when the CMO_On_Write property is True.

Table D7-3 AWSNOOP encoding

AWSNOOP	Operation	Property
0b0110	CMO	CMO_On_Write

Table D7-4 shows the signal that is added to the AW channel when CMO_On_Write is True.

Table D7-4 Signal added to AW channel when CMO_On_Write is True

Signal	Description
AWCMO[1:0]	Write address channel CMO indicator. When **AWSNOOP** is CMO, this signal indicates the type of CMO which is being signaled: **0b00** CleanInvalid **0b01** CleanShared When **AWSNOOP** is not CMO, this signal must be 0b00.

Other encodings of **AWCMO** are used to signal CMOs for persistence. See Table D7-7 on page D7-274 for a full list of encodings.

A CMO transaction on the write channels consists of a request on the AW channel and a response on the B channel. There are no transfers on the W channel in a CMO transaction.

The write response to the CMOs CleanInvalid and CleanShared have a single response beat on the B channel. This indicates that all caches are Clean and/or invalid within the specified domain and any associated writes are observable.

D7.7 Cache maintenance for Persistence

Additional cache maintenance operations are introduced that are used to provide a cache clean to the Point of Persistence or Point of Deep Persistence. These operations are used to ensure that a store operation, which might be held in a Dirty cache line, is moved downstream to persistent memory.

The Persist_CMO property is used to indicate whether a component supports cache maintenance for Persistence:

True Persistent CMOs are supported.

False Persistent CMOs are not supported.

If Persist_CMO is not declared, it is considered False.

Persistent CMOs can be transmitted on either read or write channels, according to the CMO_On_Read and CMO_On_Write properties.

If CMO_On_Read and CMO_On_Write are both False, Persist_CMO must be False.

The Persist_CMO property can be set True for the following interfaces:

- ACE5
- ACE5-Lite
- ACE5-LiteDVM

D7.7.1 Point of Persistence and Deep Persistence

In systems with non-volatile memory, each memory location has a point in the hierarchy at which data can be relied upon to be persistent when power is removed. This is known as the *Point of Persistence* (PoP).

Some systems require multiple levels of guarantee regarding the persistence of data. For example, some data might need the guarantee that it is preserved on power failure and also backup battery failure. To support such a requirement, this specification also defines the *Point of Deep Persistence* (PoDP).

Systems might have different points for the PoP and PoDP, or they might be the same.

D7.7.2 Persistent CMO (PCMO) transactions

The specification supports the following PCMO transactions:

CleanSharedPersist

When this completes, all cached copies of the addressed line in the specified domain are Clean and any associated writes are observable and have reached the Point of Persistence.

CleanSharedDeepPersist

When this completes, all cached copies of the addressed line in the specified domain are Clean and any associated writes are observable and have reached the Point of Deep Persistence.

When a component receives a PCMO, it is processed in the same way as a CleanShared transaction. If a snoop is required, a CleanShared snoop transaction is used.

D7.7.3 PCMO propagation

The propagation of PCMOs downstream of components, depends on the system topology. A PCMO must be propagated downstream in the following circumstances:

1. If the PCMO is cacheable and there is a downstream cache which might have allocated the cache line and there is an observer downstream of that cache.
2. If there is a PoP downstream of the component.
3. If the PCMO is a CleanSharedDeepPersist and there is a PoDP downstream of the component.

If (1) applies, but not (2) or (3), then a CleanSharedPersist or CleanSharedDeepPersist can be changed to a CleanShared before being sent downstream.

If the PCMO is changed to a CleanShared, the Persist response must be sent by the component doing the transformation.

D7.7.4 PCMOs on read channels

If using read channels to transport cache maintenance operations, only the CleanSharedPersist transaction is supported.

Table D7-5 shows how the request is signaled on the AR channel using the following encoding of **ARSNOOP**.

Table D7-5 Encoding of ARSNOOP for PCMOs on read channels

ARSNOOP	Operation
0b1010	CleanSharedPersist

CleanSharedDeepPersist is not supported on read channels.

A CleanSharedPersist transaction on the read channels has a single response beat on the R channel. This indicates that the request is observed, and all cache lines have been cleaned to the PoP.

D7.7.5 PCMOs on write channels

When using write channels to transport cache maintenance operations, CleanSharedPersist and CleanSharedDeepPersist are both supported.

PCMO request on the AW channel

Table D7-6 shows how PCMO requests are signaled using a combination of the **AWSNOOP** and **AWCMO** signals.

Table D7-6 Encoding of AWSNOOP on write channels

AWSNOOP	Operation
0b0110	CMO

Table D7-7 shows that when **AWSNOOP** indicates CMO, the **AWCMO** signal indicates the type of operation being requested.

Table D7-7 AWCMO operation types

Signal	Description
AWCMO[1:0]	0b00: CleanInvalid 0b01: CleanShared 0b10: CleanSharedPersist 0b11: CleanSharedDeepPersist

When Persist_CMO is False, **AWCMO** must not indicate CleanSharedPersist or CleanSharedDeepPersist.

PCMO response on the B channel

Table D7-8 shows the signals that are added to the write response channel when CMO_On_Write and Persist_CMO are both True.

Table D7-8 Signals added to write response channel with CMO_On_Write and Persist_CMO

Signal	Description
BCOMP	Response flag that indicates that a write is observable. If **BCOMP** is present on an interface, it must be asserted for one beat of the write response for all write transactions.
BPERSIST	Response flag that indicates that write data is updated in persistent memory. This must be asserted for one beat of a response to a CleanSharePersist or CleanSharedDeepPersist. It must be deasserted for all other responses.

CleanSharedPersist and CleanSharedDeepPersist transactions on the AW channel have two responses, known as a *Completion response* and *Persist response*.

Having separate responses enables system tracking resources to be freed up early, in the case that committing data to the PoP/PoDP takes a long time. The Completion and Persist responses can occur in any order and can be separated by responses from other transactions.

The Completion response indicates that all caches are Clean, and any associated writes are observable. It has the following rules:

- **BCOMP** is asserted and **BPERSIST** is deasserted.
- **BID** is driven with the same value as **AWID**.
- If loopback signaling is supported, **BLOOP** is driven from **AWLOOP**.
- If **AWIDUNQ** was asserted, the ID can be reused when this response is received.
- **BRESP** can be OKAY, SLVERR or DECERR.
- The Completion response must follow normal response ordering rules.

The Persist response indicates that the data has reached the PoP or PoDP. It has the following rules:

- **BCOMP** is deasserted and **BPERSIST** is asserted.
- **BID** is driven from **AWID**.
- **BLOOP** can take any value, it is not required to be driven from **AWLOOP**.
- **BRESP** can be OKAY, SLVERR or DECERR.
- The Persist response has no ordering requirements, it can overtake or be overtaken by other response beats.

A slave can optionally combine the two responses into a single beat. The following rules apply:

- **BCOMP** and **BPERSIST** are both asserted.
- **BID** is driven from **AWID**.
- If loopback signaling is supported, **BLOOP** is driven from **AWLOOP**.
- **BRESP** can be OKAY, SLVERR or DECERR.
- The combined response must follow normal response ordering rules.
- If **AWIDUNQ** was asserted, the ID can be reused when this response is received.

A master can count the number of responses returned with **BPERSIST** asserted, allowing it to determine when it has no outstanding persistent operations.

Example PCMO using write channels

Figure D7-1 shows an example of an ACE-Lite master issuing a CleanSharedPersist transaction. The last-level cache is at the PoS, so can send an OKAY response once the request has been serialized with transactions to that line from other masters. In this example, the Non-volatile Memory sends a combined Completion and Persist response, so the cache must deassert **BCOMP** when it propagates the response upstream.

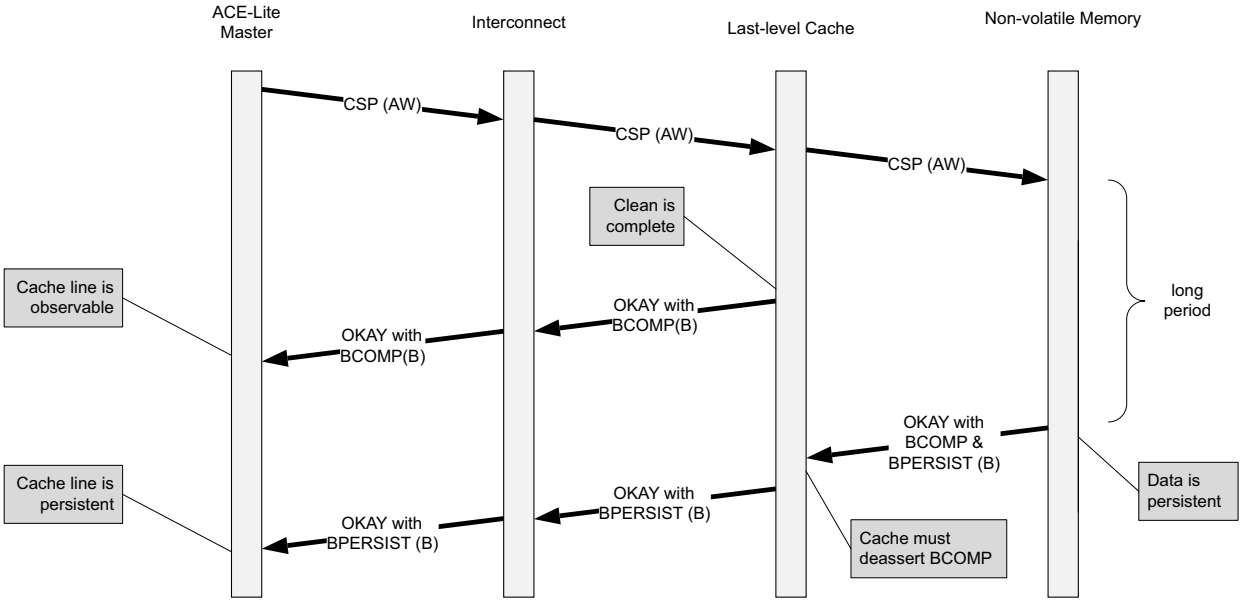


Figure D7-1 Example CMO transactions

D7.8 Write with cache maintenance

Cache maintenance operations are often used in conjunction with a write to memory. For example:

- A write from an I/O agent which must be made visible to observers which are downstream of caches.
- A write to persistent memory that must ensure that all copies of the line are also cleaned to the point of persistence.
- A CMO that causes a write back of dirty data, which must be followed by the CMO.

Write with cache maintenance adds new opcodes that combine a write with a CMO in order to improve the efficiency of this type of scenario. It is expected that some masters will natively generate a write with CMO. In other cases, a cache or interconnect will combine a CMO with a write before propagating them downstream.

D7.8.1 Write with CMO configuration

The Write_Plus_CMO property is used to indicate whether a component supports combined write and CMOs on the write channels.

True Write with CMO operations are supported on the write request channel.

False Write with CMO operations are not supported on the write request channel.

If Write_Plus_CMO is not declared, it is considered False.

The Write_Plus_CMO property can be set True for the following interface types:

- ACE5-Lite
- ACE5-LiteDVM

If the Write_Plus_CMO property is True, the CMO_On_Write property must also be True.

D7.8.2 Write with CMO operations

A combined write and CMO is supported using previously reserved encodings of **AWSNOOP**. For these new encodings, the **AWCMO** signal indicates which CMO type is combined with the write. The following CMOs are supported:

- CleanInvalid (CI)
- CleanShared (CS)
- CleanSharedPersist (CSP)
- CleanSharedDeepPersist (CSDP)

Table D7-9 shows valid combinations of **AWSNOOP**, **AWDOMAIN**, and **AWCMO** for the new operations:

Table D7-9 Valid combinations of AWSNOOP, AWDOMAIN, and AWCMO

AWSNOOP	Operation	AWDOMAIN	AWCMO Options	Size restrictions
0b1010	WritePtlCMO	Non-shareable, Inner Shareable, Outer Shareable	CI, CS, CSP, CSDP	Less than or equal to one cache line.
0b1011	WriteFullCMO	Non-shareable, Inner Shareable, Outer Shareable	CI, CS, CSP, CSDP	Cache line sized. Regular transaction.

This specification recommends that new designs use Outer Shareable in preference to Inner Shareable.

Many combinations of write, domain, and CMO are possible. Some example operations with use-case and action are shown in the following table:

Table D7-10 Combinations of write, domain, and CM

Operation	Primary use-case	Action
Outer Shareable WritePtlCMO with CS	An I/O agent writing less than a cache line to a Shareable region, where the data must be visible to observers downstream of a cache.	All in-line and peer caches must look up the line and write back any dirty data. Data from a Dirty cache line can be merged with the partial write to form a WriteFullCMO with CS to go downstream.
Outer Shareable WriteFullCMO with CSP	An I/O agent writing a cache line to a Shareable region, where the data must reach the Point of Persistence.	The coherent interconnect issues a MakeInvalid snoop to Outer Shareable peer caches. In-line caches look up the line and either update or invalidate any copies. The write and CSP must be propagated if there is a Point of Persistence downstream.
Non-shareable WriteFullCMO with CI	Issued by a cache when a CleanInvalid CMO has hit a Dirty line and caused a write to memory.	All in-line cache entries must be cleaned and invalidated. The write and CI must be propagated if there are observers downstream.

D7.8.3 Attributes for write with CMO

A write with CMO has the following attribute constraints:

- **AWDOMAIN** is Non-shareable, Inner Shareable, or Outer Shareable.
- **AWCACHE[1]** is asserted, the transaction must be Modifiable.
- **AWLOCK** is deasserted, Normal access.

A WriteFullCMO must be cache line sized and Regular, see *Regular transactions* on page A3-53.

A WritePtlCMO must be cache line sized or smaller and not cross a cache line boundary. The associated CMO applies to the whole of the addressed cache line. **AWBURST** must not be FIXED.

The cache maintenance part of the write with CMO is always treated as cacheable and Shareable, irrespective of **AWCACHE** and **AWDOMAIN**.

D7.8.4 Propagation of write with CMO

Propagation of a write with CMO follows the same rules as the propagation of a CMO. It is possible to split a write with CMO into separate write and CMO transactions for propagation downstream. In that case, either:

- The write is issued first, followed by the CMO on the write request channel with the same ID as the write.
- The write is issued first. When the write response is received, the CMO can be issued on the write or read channel.

When splitting a write and CMO, if **AWDOMAIN** is Inner Shareable or Outer Shareable, then:

- WritePtlCMO becomes WriteUniquePtl
- WriteFullCMO becomes WriteUniqueFull

If **AWDOMAIN** is Non-shareable, then the write becomes WriteNoSnoop.

The CMO is sent as cacheable. If there is a downstream cache in the shareable domain, the CMO is sent as Shareable.

If there is no cache downstream that requires management by cache maintenance, the CMO part of the transaction can be discarded. If the discarded CMO is a CSP or CSDP, the **BCOMP** and **BPERSIST** signals must be set on the write response.

D7.8.5 Response to write with CMOs

Responses to writes with CMOs follow the same rules as CMOs on the write channel.

A write with CleanInvalid or CleanShared has a single response beat which indicates that the write and CMO are both observable.

A write with a CleanSharedPersist or CleanSharedDeepPersist, has one response that indicates that the write and is observable and one response that indicates that the write has reached the PoP / PoDP.

As with a standalone CSP/CSDP, a slave can optionally combine the two responses into a single beat.

A write with CMO is not permitted to use the MTE Match opcode, so a write response that is combined with Persist and Match responses is not necessary.

D7.8.6 Example flow

This figure shows an I/O Coherent master issuing an Outer Shareable CleanSharedPersist on the AW channel. The resulting snoop hits dirty data in the coherent cache. The cache then issues a Non-shareable WriteFullCMO with CleanSharedPersist. The slave sends a single, combined response on the B channel.

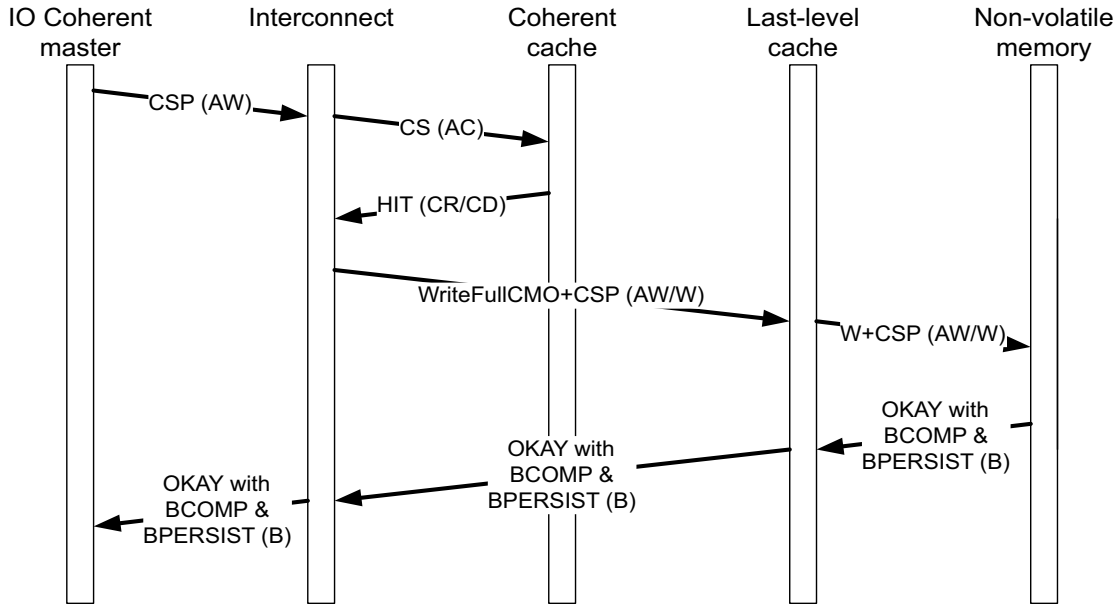


Figure D7-2 Example write with CMO

D7.9 ACE masters and CMOs

An ACE master issuing a broadcast cache maintenance operation has to co-ordinate the following:

- Appropriate action for local cache maintenance.
- Appropriate action for peer and downstream cache maintenance.

Issuing a broadcast cache maintenance transaction performs the required action on peer caches and causes the interconnect to generate a downstream cache maintenance transaction to other levels of cache.

The downstream cache maintenance transaction must be correctly ordered with respect to other transactions to the same cache line. The master carrying out the cache maintenance, must follow the sequence:

1. The master must complete any outstanding Shareable transactions, which permits the line to be allocated, to a cache line before it issues a cache maintenance transaction to the same cache line.
   a. For CleanShared and CleanInvalid operations:
      If the master holds the cache line in a Dirty state, it must issue a WriteBack or WriteClean transaction, that must complete, before issuing the cache maintenance transaction.
      If the cache line is initially Clean, but there are outstanding Cacheable transactions to the line, then the master must ensure that the line is not Dirty after the completion of all outstanding transactions to the cache line.
   b. For CleanInvalid and MakeInvalid transactions:
      The master must invalidate the cache line before issuing the cache maintenance transaction after all outstanding transactions and required WriteBack or WriteClean transactions are complete.
2. The master issues the appropriate cache maintenance transaction, after all required outstanding transactions and required WriteBack or WriteClean transactions are complete.

For CleanShared operations, the master is permitted to issue either an Evict or WriteEvict transaction at any point during the sequence. It is also permitted to perform a local write to the line, if it is in a Unique state.

The master must not issue any further Shareable transactions, that permits the line to be allocated, to the same cache line until the broadcast cache maintenance sequence is complete.

All masters that support an external snoop filter must ensure that the information that is provided enables the snoop filter to correctly track the allocation of cache lines. Typically, this is ensured by the correct use of WriteBack and WriteClean transactions and the appropriate snoop responses. See Chapter D10 *Optional External Snoop Filtering*.

For a given memory location, cache maintenance operations are permitted to use different shareability and cacheability attributes to those that the translation table attributes assign for any non-cache maintenance transaction to that location. This possible mismatch of attributes means that an interconnect cannot correctly determine the cacheability attributes to use for any interconnect-generated transactions that result from the cache maintenance operation, that is required if a snooped cache provides dirty data on the CD channel in response to a snoop transaction for the cache maintenance operation.

D7.9.1 Requirements for a snooped master

There are no additional requirements for a snooped master during a cache maintenance operation. All requirements are as specified in Chapter D5 *Snoop Transactions*.

D7.9.2 Processor cache maintenance instructions

The cache maintenance protocol requires that the cache maintenance operations use the **AxCACHE** and **AxDOMAIN** signals to identify the caches on which the cache maintenance operations must operate.

For a processor that has cache maintenance instructions that are required to operate on a different number of caches than are defined by the **AxCACHE** and **AxDOMAIN** values, the cacheability and shareability of the transaction must be adapted to meet the requirements of the processor.

For example, if a processor instruction performing a cache maintenance operation on a location with Device memory attributes is required to operate on all caches within the system, then the master must issue a cache maintenance transaction as Outer Shareable, since this is the most pervasive of the cache maintenance operations and operates on all the required caches.

D7.9.3 Unpredictable behavior with software cache maintenance

Cache maintenance can be used to reliably communicate shared memory data structures between a coherent group of masters and non-coherent agents. This process must follow a particular sequence to reliably make the data structures visible as required.

When using cache maintenance to make the writes of a non-coherent agent visible to a coherent group of masters, there are periods of time when writing and reading the data structures gives UNPREDICTABLE results and can cause a loss of coherency.

The observation of a line that is being updated by a non-coherent agent is UNPREDICTABLE during the period between the clean transaction that starts the sequence and the invalidate transaction that completes it. During this period, it is permissible to see multiple transitions of a cache line that is being updated by a non-coherent agent.

Figure D7-3 shows the required sequence of communication between a coherent domain and a non-coherent agent.

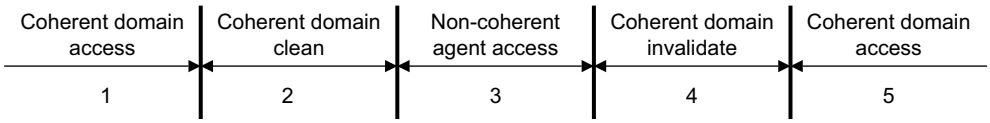


Figure D7-3 Required sequence of communication between coherent and non-coherent domains

The five stage sequence that Figure D7-3 shows is:

1. The coherent domain has access. The coherent domain has full read and write access to the appropriate memory locations during this stage. This stage finishes when all required writes from the coherent domain are complete within the coherent domain.
2. The coherent domain is cleaned. A cache clean operation is required for all the address locations that are undergoing software cache maintenance during this stage. The coherent domain clean forces all previous writes to be visible to the non-coherent agent. This stage finishes when all required writes are complete and therefore visible to the non-coherent agent.
3. The non-coherent agent has access. The non-coherent agent has both read and write access to the defined memory locations during this stage. This stage finishes when all required writes from the non-coherent agent are complete.
4. The coherent domain is invalidated. A cache invalidate operation is required for all the address locations that are undergoing software cache maintenance during this stage. This coherent domain invalidate stage removes all cached copies of the defined locations ensuring that any subsequent access from the coherent domain observes the writes from the non-coherent agent. This stage finishes when all the required invalidations are complete.
5. The coherent domain has full access to the defined memory locations.

Table D7-11 shows when accesses from the coherent domain or the non-coherent agent are permitted. The remaining accesses can have UNPREDICTABLE results, with possible loss of coherency.

Table D7-11 Permitted accesses from the Coherent domain and Non-coherent agent

Phase	Description	Coherent domain		External agent	
		Read	Write	Read	Write
1	Coherent domain access	Permitted	Permitted	-	-
2	Coherent domain clean	-	-	-	-
3	External agent access	-	-	Permitted	Permitted
4	Coherent domain invalidate	-	-	-	-
5	Coherent domain access	Permitted	Permitted	-	-

Chapter D8
Barrier Transactions

This chapter describes ACE barrier transactions. It contains the following sections:

D8.1 About barrier transactions

Barrier transactions provide guarantees about the ordering and observation of transactions in a system. Barrier transactions are not supported in ACE5 and ACE5-Lite variant interfaces. See *Barrier transaction support* on page F5-453 for further details.

ACE supports memory barriers and synchronization barriers:

- A memory barrier is issued by a master to guarantee that if another master in the appropriate domain can observe any transaction issued after the barrier it must be able to observe every transaction issued before the barrier.
- A synchronization barrier is issued by a master to guarantee that all transactions that are issued before the barrier are observable by every master in the appropriate domain when the barrier completes. System domain synchronization barriers have the additional requirement that all transactions that are issued before the barrier transaction must have reached the endpoint slaves they are destined for before the barrier completes.

A memory barrier is used for memory-based communication. For example, when writing an array of data to memory, a master component can issue a memory barrier before setting a memory flag to indicate that the array is available. Any other master component that can observe the flag must observe all transactions that write to the array.

Note

It is not necessary for all master components in the domain to observe the updated array at the same time. It is a requirement for each master in the domain that can observe the flag, to be guaranteed to observe the updated array.

A synchronization barrier is used with various forms of sideband signaling communication. For example, when writing an array of data to memory, a master component can use a synchronization barrier before generating an interrupt to indicate that the array is available. When the synchronization barrier completes, the updated array is guaranteed to be observable by all master components in the domain.

Barrier transactions can be read or write transactions, and are defined as follows:

Read barrier transactions

A master component issues a read barrier transaction on the read address channel and a response is returned on the read data channel. No data transfer occurs.

Write barrier transactions

A master component issues a write barrier on the write address channel and a response is returned on the write response channel. No data transfer occurs.

A master component must issue barrier transactions as a barrier pair, with a barrier transaction on both the read address channel and the write address channel. For each address channel, any transaction that is issued on the channel, before the barrier transaction, is defined to be before the barrier, even if it is issued after the corresponding barrier on the other address channel. A transaction is defined to be after the barrier if it is issued after both the read barrier response and write barrier response are received.

D8.2 Barrier transaction signaling

This section describes the read address channel and write address channel signaling associated with barrier transactions.

To permit interworking between barrier transactions and QoS, the following types of non-barrier transactions exist:

- Transactions that are affected by barrier transactions.
- Transactions that are not affected by barrier transactions.

This specification recommends that, by default, all transactions are affected by barrier transactions. The only transaction streams that can be signaled so that they are not affected by barrier transactions are transactions that do not require ordering with respect to other streams, such as those related to real-time data flows.

D8.2.1 AxBAR signaling

AxBAR is used to differentiate between barrier transactions and normal transactions. For normal transactions, **AxBAR** also indicates that the associated transaction must respect barriers or if the ordering requirements of any barrier transactions can be ignored. For barrier transactions, **AxBAR** also indicates that the transaction is a memory barrier or a synchronization barrier. See *Read and write barrier transactions* on page D3-178 for more information about **AxBAR** encoding.

Table D3-13 on page D3-185 shows the constraints that apply to barrier transactions.

D8.2.2 AxDOMAIN signaling

The **AxDOMAIN** signal determines the level of propagation of a barrier transaction, defining the domains within a system that the barrier transaction accesses. See *Shareability domain types* on page D3-176. Table D8-1 shows the different levels of barrier applicability for each domain type.

Table D8-1 Domain barrier applicability

AxDOMAIN	Domain	Barrier Applicability
00	Non-shareable	Acts as a barrier to other transactions in the current transaction stream. When two or more transaction streams are combined, no ordering is required with respect to the newly combined transaction stream.
01	Inner Shareable	Ordering must be established with respect to other transactions from all masters in the same Inner Shareable domain.
10	Outer Shareable	Ordering must be established with respect to other transactions from all masters in the same Outer Shareable domain.
11	System	Ordering must be established with respect to all other transactions. For a Synchronization barrier, a response must only be given when all transactions from the issuing master, which are ahead of the barrier, have reached their endpoint.

D8.2.3 Response signaling

All barrier transactions must complete with a response, as follows:

- Responses for barrier transactions issued on the read address channel are signaled on the read data channel.
- Responses for barrier transactions issued on the write address channel are signaled on the write response channel.

The response must have a matching AXI ID to the barrier transaction and OKAY is the only permitted response for a barrier transaction. Table D8-2 shows the constraints for barrier transaction response signaling.

Table D8-2 Barrier response transaction constraints

Attribute	Constraint
RID, **BID**	Must match barrier transaction ID.
RRESP	Must be all zeros.
RLAST	Must be HIGH.
RDATA	No constraint, can take any value. Must be ignored.
BRESP	Must be all zeros.

Note

User-defined signals, such as **ARUSER**, **AWUSER, RUSER**, **WUSER**, and **BUSER**, cannot be reliably transported alongside barrier transactions. It is therefore recommended that user-defined signals are all zeros for barrier transactions.

D8.3 Barrier responses and domain boundaries

The location of an interconnect in relation to the domain boundaries within a system influences the ability of the interconnect to issue responses to barrier transactions. In general, a system can contain domain boundaries and bi-section boundaries, where:

- A domain boundary is an interface downstream from all master components in the domain.
- A bi-section boundary is downstream of a subset of master components in the domain, but not all of them.

Figure D8-1 shows the domain boundaries in an example system.

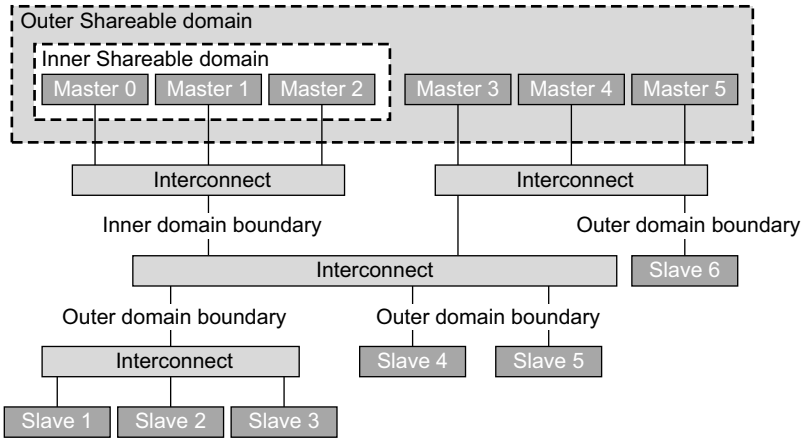


Figure D8-1 Domain boundaries

Note

The interface to Slave 6 is an outer domain boundary. Slave 6 cannot be accessed by Master 0, Master 1, and Master 2, and therefore it cannot be considered relative to these master components. It is downstream of all master components in the outer domain that can access it.

Figure D8-2 shows the bi-section boundary locations for the same system.

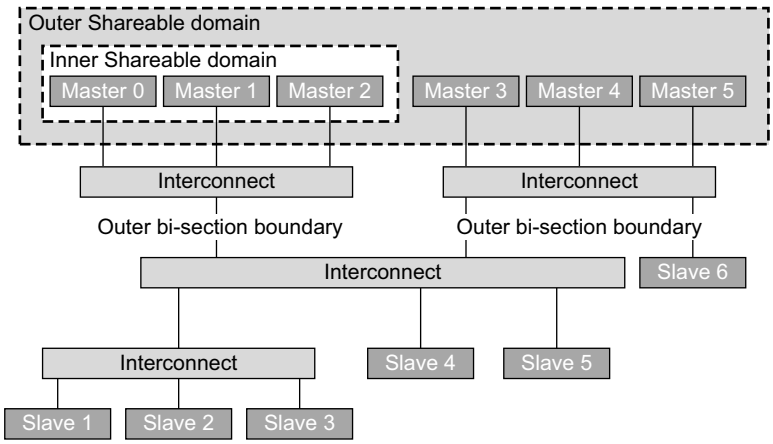


Figure D8-2 Bi-section boundaries

For an interconnect to issue a response to a barrier transaction, certain conditions apply. The main consideration influencing the ability of an interconnect to issue responses to barrier transactions is the location of the interconnect in relation to the domain boundaries within a system.

A system can contain the following types of boundary:

Domain boundary

A domain boundary is an interface downstream from all master components in the domain.

For an interface to be a domain boundary, all of the following must apply:

- The set of addresses that pass across the interface is identical for all masters in the domain.
- All accesses from any master in the domain to those addresses pass across the interface.

Note

If a master component can access an address using the interface, then it must not be possible for another master in the same domain to access the same address without using the interface.

Bi-section boundary

A bi-section boundary is an interface downstream of a subset of master components in the domain, but not all of them.

For an interface to be a bi-section boundary, all of the following must apply:

- The set of addresses that pass across the interface is identical for a subset of master components in the domain.
- All accesses from any master component in that subset to those addresses pass across the interface.
- No accesses from a master component that is in the domain but is not in the subset passes across the interface.
- Considering in turn each master component not in the subset, then all addresses that are accessed by both that master and the masters in the subset must be accessed by the masters in the subset across the same interface.

Note

- Informally, an interface is a bi-section boundary if all communication between a subset of masters and the other masters not in the subset pass across the same interface.
- In the definition of a bi-section boundary, the subset of masters is permitted to be all masters in the domain and this makes the bi-section boundary definition the same as the domain boundary definition.

See *Barrier responses and domain boundaries* on page D8-287 for more information.

An interconnect can provide a response to a barrier transaction in certain circumstances. The following rules apply:

- For memory barrier transactions, an interconnect can respond provided it is at the appropriate bi-section boundary or domain boundary, or beyond the domain boundary.
- For any synchronization barrier transaction that applies to a Non-shareable, Inner Shareable or Outer Shareable domain, an interconnect can respond provided the interconnect is at or beyond the appropriate domain boundary.
- For any synchronization barrier transaction that applies to a System domain, an interconnect can respond provided all transactions before the barrier have reached the endpoint slaves they are destined for.

When responding to a barrier transaction, an interconnect must ensure that all transactions that pass across the interface before the barrier are observable to every transaction after the barrier. Some techniques that can be used to achieve this are:

Blocking all transactions and sending barrier

The interconnect blocks all transactions received after the barrier transactions and issues a barrier transaction downstream. The block is removed after a response has been received on both the read data and write response channels for the downstream issued barrier transactions.

Blocking all transactions and waiting for completion

The interconnect blocks all transactions after the barrier transactions and waits for transactions before the barrier to provide a response. The block is removed when all transactions before the barrier have provided a response. To use this technique, it is required that all transactions must have attributes that ensure the response originates from a location that is observable by all masters in the required barrier domain.

Hazard-checking transactions

The interconnect blocks all transactions after the barrier transactions until transactions before the barrier to the same or overlapping addresses have provided a response.

D8.4 Barrier requirements

This section describes the formal requirements for barrier transactions.

D8.4.1 Master requirements

For a master component issuing a barrier transaction, the following rules apply:

- Both transactions in a barrier pair must have the same **AxID**, **AxBAR**, **AxDOMAIN**, and **AxPROT** values.
- If the **ARID** and **AWID** signals have different widths, the narrower version must be zero-extended to match the wider version.
- Barrier pairs must be issued in the same sequence on the read address and write address channels.
- A master interface is not required to issue barrier transactions on the read address and write address channels in the same cycle.
- A master interface is permitted to issue multiple outstanding barriers, meaning that additional barrier transactions can be issued before responses to earlier barrier transactions are received. However:
  - An ACE-Lite master interface can issue outstanding barrier transactions without restriction.
  - An ACE master interface must not issue more than 256 outstanding barrier transactions.

  Note

  Read and write response handshakes are separate events that can occur in any order. Therefore, a barrier is defined as an outstanding barrier from the cycle when the first of the read or write barrier becomes valid until the cycle when both the read and write response handshakes have occurred.

- Barrier transactions are required to use different ID values than those used for non-barrier transactions. It is permissible for barrier transactions and non-barrier transactions to use the same AXI ID value, provided one transaction has completed before the other is issued.

  Note

  Using different ID values ensures that any component tracking barrier responses does not have to track all responses to differentiate between a barrier response and a normal transaction response.

On each address channel, any transaction issued before the barrier on that channel is defined to be before the barrier, even if it is issued after the corresponding barrier on the other address channel.

Figure D8-3 shows pre-barrier and post-barrier transactions, with respect to barriers being issued on the address channels and responses being received:

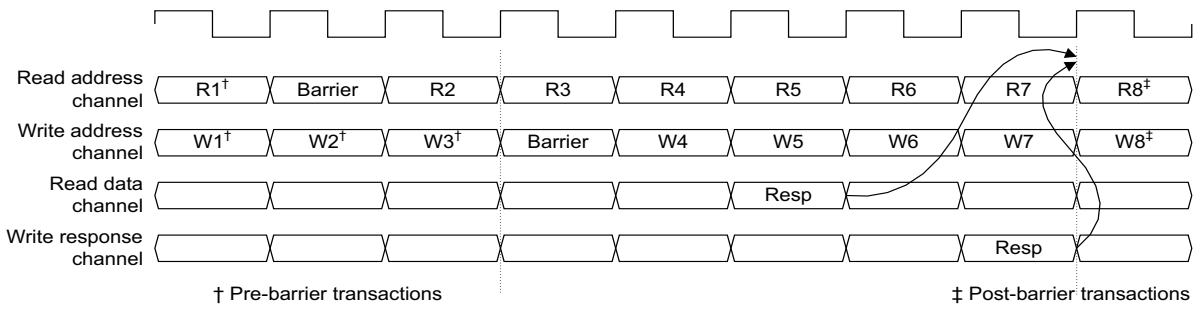


Figure D8-3 Barrier transaction timing

In Figure D8-3 on page D8-290:

- Transactions R1, W1, W2, and W3 are before the barrier.
- Transactions R8 and W8 are after the barrier.
- Transactions R2, R3, R4, R5, R6, R7, W4, W5, W6, and W7 have no relationship to the barrier.

The following rules apply to master components issuing barrier transactions, and relate to non-barrier transactions:

- A master must not issue any transaction, either read or write, that must be after the barrier until the master has received a response for the barrier on both the read data and write response channels.
- A master is permitted to issue transactions between issuing a barrier transaction on the address channel and receiving the read and write barrier responses. Such transactions have no ordering guarantees with respect to the barrier. On the address channel, these transactions are permitted to remain after the barrier transaction or they are permitted to overtake the barrier transaction.
- A master interface that has issued a read barrier on the read address channel must issue the corresponding write barrier on the write address channel, *in a timely manner*, if all other transactions on the write address channel are progressed. The master interface must not require either handshaking or a response to the read barrier or any read transaction after the read barrier before issuing the corresponding write barrier.
- A master interface that has issued a write barrier on the write address channel, must issue the corresponding read barrier on the read address channel, *in a timely manner*, if all other transactions on the read address channel are progressed. The master interface must not require either handshaking or a response to the write barrier or any write transaction after the write barrier before issuing the corresponding read barrier.
- A barrier must not be issued on the read address channel if subsequent read transactions are required for either:
  - Issuing the corresponding barrier on the write address channel
  - Issuing any write transactions that must be before the barrier

  For example, a read barrier must not be issued if, after issuing the read barrier, it is necessary to perform translation table walks to issue write transactions that must be before the corresponding write barrier.
- For an ACE-Lite interface, a barrier must not be issued on the write address channel if subsequent write transactions are required for either:
  - Issuing the corresponding barrier on the read address channel
  - Issuing any read transactions that must be before the barrier
- For an ACE interface, a barrier must not be issued on the write address channel if subsequent write transactions that must be ordered with respect to the barrier, must be issued for either:
  - Issuing the corresponding barrier on the read address channel
  - Issuing any read transactions that must be before the barrier
- An ACE interface is permitted to issue a write barrier, followed by any of the following transactions that are required for snoop transactions to that master to complete:
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict

The following rules apply to ACE master components and the interaction of barriers and local cache accesses:

- A master must not perform a store that must be ordered with respect to the barrier, to a Shareable location in its local cache until after the barrier response is received on both read data and write response channels. This rule applies even if there is no requirement for a transaction to be issued because the cache line is in a Unique state.
- A master must not perform a load that must be ordered with respect to the barrier, from a Shareable location in its local cache until after the barrier response is received on both read data and write response channels. This applies even if there is no requirement for a transaction to be issued because the cache line is in a Valid state.

- A master must be capable of issuing write transactions to complete snoop transactions, even if the read address channel is stalled.
- Issuing a barrier transaction must not prevent any of the following transactions, that are required for snoop transactions, being issued and completed:
  - WriteBack
  - WriteClean
  - WriteEvict
  - Evict

D8.4.2 Slave requirements

The following rules apply to a slave component that is handling barrier transactions:

- On receipt of a barrier transaction, an ACE-Lite slave interface is permitted to either:
  - Stall the read address channel until it receives the corresponding barrier transaction on the write address channel.
  - Stall the write address channel until it receives the corresponding barrier transaction on the read address channel.
- An ACE slave interface must be able to accept 256 barrier transactions on the write address channel without blocking the progress of subsequent transactions. It is required that the write address channel is available and that write transactions can progress.
- On receipt of a read barrier, an ACE slave interface is permitted, but not required, to stall the read address channel.

D8.4.3 Interconnect requirements

The following rules apply to interconnect processing barrier transactions:

- The interconnect topology must not permit transactions with overlapping destination addresses to have a common start point and end point but have different paths through the interconnect.
- When merging two streams of transactions, an interconnect must ensure that barrier pairs are issued on the read and write channels in the same sequence. Barriers pairs must not be interleaved.
- Any interconnect component that has multiple ACE slave ports must be capable of meeting the ACE slave interface requirements in *Slave requirements*, for all ports simultaneously.
- An interconnect must not permit a barrier transaction to overtake any transaction that respects barriers.
- A barrier must apply to any transaction that the component issuing the barrier observed before issuing the barrier.
- An interconnect that has not responded to a barrier can permit any non-barrier transaction to overtake that barrier.

Note

This specification recommends that ACE interconnect components stall a read barrier until:

- The corresponding write barrier is received.
- All transactions before the read barrier and write barrier have been snooped as required, and all write transactions that must be before the barrier have been issued.

D8.4.4 Barriers and Device transaction ordering

Barrier transactions ensure ordering between Device transactions to a single peripheral device, regardless of the addresses within the peripheral being accessed. This means that any hazard-checking that an interconnect performs when responding to a barrier must be extended to the entire address space of the peripheral, and must not consider only overlapping transactions. If an interconnect is unable to determine the address range of a particular peripheral, it must ensure ordering between all accesses that could be addressing that peripheral.

D8.4.5 Multi-copy atomicity requirements for Shareable locations

For Inner Shareable and Outer Shareable locations, multi-copy atomicity is required. This means that a snoop response to a write is issued only when all observers in the required shareability domain have observed the write. Also, on the cycle that a snoop response is received, the associated master must have already observed the write. The point of observation is defined as the handshake on the snoop response channel. The snoop data channel is not used in defining the point of observation. This means that there is no requirement for barriers on the snoop address channel.

Chapter D9
Exclusive Accesses from ACE Masters

This chapter describes Exclusive accesses from ACE masters. It contains the following sections:

D9.1 About Exclusive accesses from ACE masters

The principles of Exclusive accesses are that a master performing an Exclusive sequence does the following:

- Performs an Exclusive Load from a location
- Calculates a value to store to that location
- Performs an Exclusive Store to the location:
  - The Exclusive Store fails if another master has performed a store to the location since the Exclusive Load. In this case, the Exclusive Store does not occur and the master does not change the value that is held at the location.
  - The Exclusive Store can pass if no other master has performed a store to the location since the Exclusive Load. In this case, the store can occur and the master can change the value that is held at the location.

Note

An Exclusive Load by a processor is caused by the execution of an instruction such as `LDREX`. An Exclusive Store by a processor is caused by the execution of an instruction such as `STREX`.

In the ACE protocol, correct execution of an Exclusive sequence places requirements on both the master performing the Exclusive sequence and the interconnect.

For Non-shareable and System Shareable locations, the behavior is identical to the behavior specified in AXI.

For Inner Shareable and Outer Shareable locations, the following requirements apply:

- The master is responsible for ensuring that it only updates the location if no other master can have performed a store to the location since the master performed the Exclusive Load. The term *master exclusive monitor* describes the monitor that must exist within the master component to meet this requirement.
- The interconnect is responsible for ensuring that if two masters attempt an Exclusive Store transaction to the same location and it is possible that the second master will have its copy of the location invalidated before its Exclusive Store transaction completes, then the interconnect must fail the Exclusive Store transaction from the second master. The term *PoS exclusive monitor* describes the monitor that must exist within the interconnect, at the point of serialization, to meet this requirement.

The term *Exclusive Store* is used to describe the action of a master executing an appropriate program instruction. When an Exclusive Store passes, this indicates an update to the data value at the address location. When an Exclusive Store fails, this indicates that the store has not changed the data value at the address location, and the Exclusive sequence must be restarted.

The term *Exclusive Store transaction* is used to describe the transaction that is issued on the ACE interface of a master. Not every Exclusive Store requires an Exclusive Store transaction. An Exclusive Store transaction can pass or fail and this result is made known to the master using the transaction response. When an Exclusive Store transaction passes, this indicates that the transaction has been propagated to other masters, but it does not indicate whether the Exclusive Store passes or fails. When an Exclusive Store transaction fails, this indicates that the transaction has not been propagated to other masters and therefore the associated Exclusive Store cannot pass.

All masters that attempt an Exclusive access to the same location must be using the same shareability for the location. If the location for the Exclusive access is Shareable, then all masters must be able to participate in the coherency protocol.

When first obtaining a copy of the exclusive location, it is important that the line is not removed from another cache that is also performing an Exclusive sequence to the same cache line. For this reason, ReadClean or ReadShared must be used rather than ReadNotSharedDirty or ReadUnique.

A Load Exclusive, Store Exclusive sequence (`LDREX`, `STREX`) must use an Exclusive sequence. An atomic update, such as a swap operation or a read-modify-write atomic operation, is not required to use an Exclusive sequence. For such atomic updates, it is permitted to use a ReadUnique transaction that is not marked as Exclusive, if it can be guaranteed to successfully complete the atomic update with no other external action.

D9.2 Role of the master

The master must implement a *master exclusive monitor*, that is used to monitor the location that is used by an Exclusive sequence. This master exclusive monitor is used to determine if another master could have performed a store to the location during the Exclusive sequence by monitoring the snoop transaction that it receives.

When the master performs an Exclusive Load, the master exclusive monitor is set. The master exclusive monitor is reset when a snoop transaction is observed that indicates another master could perform a store to the location.

Note

In some implementations, the cache line state is sufficient to provide the functionality of the master exclusive monitor. However, it is important that a line that is invalidated and made valid again by a mechanism such as prefetching, is not considered as having remained valid since the Exclusive Load.

D9.2.1 Exclusive Load

The master starts an Exclusive sequence with an Exclusive Load. The start of the Exclusive sequence must set the master exclusive monitor.

If the master does not hold a copy of the cache line, then it must obtain a copy of the line using either a ReadClean or a ReadShared transaction.

An Exclusive Load transaction is a ReadClean or ReadShared transaction with the **ARLOCK** signal asserted. This indicates to the PoS exclusive monitor that the master is starting an Exclusive sequence.

It is recommended, but not required, that a master use an Exclusive Load transaction, with **ARLOCK** asserted, if it is issuing a transaction at the start of Exclusive sequence. If a master does not use an Exclusive Load transaction, it is permitted to use a ReadClean or ReadShared transaction with **ARLOCK** deasserted.

If the master holds a copy of the line in a Unique state, then issuing a transaction for the Exclusive Load is permitted but not recommended.

Note

This transaction is likely to cause an external memory access. It is also likely that informing the interconnect that an Exclusive sequence has started is unnecessary, since there is no requirement to issue an Exclusive Store transaction to complete the sequence if the cache line remains in the Unique state.

If the master holds a copy of the line in a Shared state, then issuing a transaction for the Exclusive Load is permitted, but not required.

Note

Issuing a transaction informs the interconnect that the master is performing an Exclusive sequence.

An Exclusive Load is expected to receive an EXOKAY response, which indicates that Exclusive accesses are supported at the address of the transaction. If Exclusive accesses are not supported, then the transaction will receive an OKAY response.

D9.2.2 Exclusive Load to Exclusive Store

After the execution of an Exclusive Load a master will typically calculate a new value to store to the location before it attempts the Exclusive Store.

It is not required that a master always completes an Exclusive sequence. For example, the value that is obtained by the Exclusive Load can indicate that a semaphore is held by another master and therefore the value cannot be changed until the semaphore is released by the other master. Therefore, the Exclusive sequence can be restarted with no attempt to complete the current Exclusive sequence.

During the time between the Exclusive Load and the Exclusive Store, the master exclusive monitor must monitor the location to determine whether another master might have performed a store to the location.

D9.2.3 Exclusive Store

A master must not permit an Exclusive Store transaction to be in progress at the same time as any transaction that registers that it is performing an Exclusive sequence. The master must wait for any such transaction to complete before issuing an Exclusive Store transaction. The transactions that register that a master is performing an Exclusive sequence are Exclusive Load transactions to any location, and Exclusive Store transactions to any location. These transactions are:

- ReadClean with **ARLOCK** asserted
- ReadShared with **ARLOCK** asserted
- CleanUnique with **ARLOCK** asserted

When a master executes an Exclusive Store, the following behavior is required:

- If the master exclusive monitor has been reset, the Exclusive Store must fail and the master must not issue an Exclusive Store transaction. The master must restart the Exclusive sequence.

  Note

  In this case, not issuing an Exclusive Store transaction avoids unnecessarily invalidating other copies of the line by preventing the issue of a transaction that will eventually fail.

- If the line is held in a Unique state and the master exclusive monitor is set, then the Exclusive Store has passed and the master can execute the Exclusive Store without issuing a transaction.
- If the line is held in a Shared state and the master exclusive monitor is set, then the master must issue a transaction to perform the Exclusive Store. This check of the master exclusive monitor must only occur after any other transactions that register a master is performing an Exclusive sequence have completed. A CleanUnique transaction with **ARLOCK** asserted must be used. The master exclusive monitor must continue to operate during this transaction. The transaction will respond with an OKAY or EXOKAY response.

If the transaction receives an EXOKAY response, then it indicates that the transaction has passed and been propagated to invalidate all other copies of the line. After an Exclusive transaction completes with an EXOKAY response, the master must again check the master exclusive monitor:

- If the master exclusive monitor is set, then the Exclusive Store has passed and the store is performed.
- If the master exclusive monitor has been reset, it indicates that a non-Exclusive Store has occurred to the cache between the point that the Exclusive Store transaction was issued and the point that it completed. The Exclusive Store must fail and the Exclusive sequence must be restarted.
- If the master has not been able to track the exclusive nature of the cache line, because the line has been evicted, then the Exclusive Store must fail and the Exclusive sequence must be restarted.

If the transaction receives an OKAY response, then it indicates that another master has been permitted to progress a transaction that is associated with an Exclusive Store. The transaction that is associated with the Exclusive Store from this master has failed and has not propagated to other masters in the system. When an Exclusive transaction completes with an OKAY response the following options exist:

- The master can fail the Exclusive Store and restart the Exclusive sequence without checking the state of the line when the access completed.
- The master can check the master exclusive monitor:
  - If the master exclusive monitor has been reset, then the master must fail the Exclusive Store and restart the Exclusive sequence.
  - If the master exclusive monitor is set, then the master can repeat the Exclusive Store transaction.

D9.3 Role of the interconnect

The interconnect can pass or fail an Exclusive Store transaction. A pass indicates that the transaction has been propagated to other cacheable masters. A fail indicates that the transaction has not been propagated to other masters and therefore the Exclusive Store cannot pass.

The interconnect is required to have a monitor that is used to ensure that an Exclusive Store transaction from a master is only successful if that master could not have received a snoop transaction relating to an Exclusive Store to the same address from another master after it issued its own Exclusive Store transaction. This monitor is referred to as the *PoS exclusive monitor* and it exists within the interconnect at the point of serialization.

D9.3.1 Minimum PoS Exclusive Monitor

The minimum requirement of PoS exclusive monitor is to record when any master performs a Shareable transaction that is related to an Exclusive sequence. The Shareable transactions that are related to an Exclusive sequence are:

- ReadClean with **ARLOCK** asserted
- ReadShared with **ARLOCK** asserted
- CleanUnique with **ARLOCK** asserted

If a master has performed a transaction that is related to an Exclusive sequence and it then performs an Exclusive Store transaction before a successful Exclusive Store transaction from another master is scheduled, then the Exclusive Store transaction must be successful.

The monitor must support the parallel monitoring of all Exclusive-capable masters in the system.

When the interconnect receives a transaction that is associated with an Exclusive Load or an Exclusive Store, the monitor registers that the master is attempting an Exclusive sequence. If an Exclusive Store fails, indicated by an OKAY response, the attempt must be recorded. If the Exclusive Store transaction is successful it is recommended, but not required, that the monitor registers that the master is attempting an Exclusive sequence.

If an Exclusive Store transaction from one master passes, the registered attempts of all other masters are reset. The other masters can only register a new Exclusive sequence when it is guaranteed that outstanding Exclusive Stores can complete without the line being invalidated by later Exclusive Stores. This can be achieved by observing the **RACK** from the Exclusive Store or through address hazarding in the interconnect.

When the interconnect receives an Exclusive Store transaction:

- If the PoS exclusive monitor has registered that the master is performing an Exclusive sequence, that is, it has not been reset by another master's Exclusive Store transaction, then the Exclusive Store transaction is successful and is allowed to proceed. An EXOKAY response is returned to the issuing master.
- If the PoS exclusive monitor has not registered that the master is performing an Exclusive sequence, that is, it has been reset by another master's Exclusive Store transaction, then the Exclusive Store transaction is failed and is not allowed to proceed. An OKAY response is returned to the issuing master.

Note

A successful Exclusive Store transaction from a master does not have to reset that the master is performing an Exclusive sequence. The master can continue to perform a sequence of Exclusive Store transactions that will all be successful, until another master performs a successful Exclusive Store transaction.

From reset, the first master to perform an Exclusive Store transaction can be successful, but is not required to be. At that point, all other masters must then register the start of their Exclusive sequence for their Exclusive Store transaction to be successful.

D9.3.2 Additional address comparison

The interconnect monitor can be enhanced to include some address comparison. A full address comparison is not required and it is permitted to only record a subset of address bits. This approach reduces the chances of an Exclusive Store transaction failing because of another master's Exclusive Store transaction to a different address location. The number of bits of address comparison that is used is an implementation choice.

Where additional address comparison monitor is used, the monitored address bits are recorded at the start of an Exclusive sequence on either a Load Exclusive or Store Exclusive transaction. It is reset by a successful Store Exclusive transaction from another master to a matching address.

A monitor with additional address comparison must include a minimum monitor of a single bit for every Exclusive-capable master to ensure progress.

An Exclusive Store transaction is allowed to progress if one of the following occurs:

- The address monitor has registered an Exclusive sequence for a matching address from the same master and has not been reset by an Exclusive Store transaction from a different master with a matching address.
- The minimum single bit monitor has registered an Exclusive sequence from the same master, and it has not been reset by an Exclusive Store transaction from a different master to any address.

Note

In the above description, the term *matching address* is used to describe where a monitor only records a subset of address bits. A matching address is where the address bits that are recorded are identical, but the address bits that are not recorded can be different.

An implementation does not require address monitor for each Exclusive-capable master. Because the address monitor provides a performance enhancement, it is acceptable to have fewer address monitors and for the use of these to be IMPLEMENTATION DEFINED. Examples of how the additional address monitors can be used include:

- Use on a first-come first-served basis
- Allocation to particular masters
- A more complex algorithm

Additional PoS Exclusive Monitor functionality can be provided to prevent interference, or denial of service, caused by one agent in the system issuing a large number of Exclusive access transactions. This specification recommends that Secure exclusive accesses are permitted to make progress independently of the progress of Non-secure exclusive accesses.

D9.3.3 Multiple interconnect PoS monitors

When the interconnect contains multiple points of serialization, as the serialization for different address ranges is done at different points within the interconnect, then a PoS exclusive monitor can be at each point of serialization.

D9.3.4 PoS Exclusive Monitor behavior

The following terms can be used to describe the behavior of a PoS exclusive monitor:

True pass Describes the case where an Exclusive Store transaction is permitted to progress and when the transaction completes the Exclusive Store will also pass, permitting the master to make forward progress.

True fail Describes the case where an Exclusive Store transaction is failed because another master has already performed a successful Exclusive Store transaction to the same address location, so at least one agent has made forward progress.

False pass Can occur for an Exclusive Store transaction, for which the Exclusive Store will eventually fail. This can only occur when a non-exclusive store transaction from another agent to the same location has occurred. This non-exclusive store transaction from another agent is considered as progress for that other agent.

False fail Can occur for an Exclusive Store transaction in the following circumstances:

- No transaction was issued for the Load Exclusive. This is resolved by re-issuing the Exclusive Store.
- An Exclusive Store transaction from another agent to a different location has been successful between the point that the Exclusive sequence is first registered and the point that the Exclusive Store transaction is scheduled. This is resolved by re-issuing the Exclusive Store.
- An Exclusive Store transaction from another agent to the same location has been successful, but that other agent is destined to eventually fail because a third party has performed a non-exclusive store transaction. This non-exclusive store transaction from the third party is considered as progress for that third party.

D9.4 Multiple Exclusive Threads

The protocol can support more than one Exclusive-capable master for each interface. This permits multiple masters to use the same interface for Exclusive accesses. In this scenario, the interconnect must be able to use the AXI ID values to differentiate between the different masters using the same interface.

D9.5 Exclusive Accesses from AXI components

An important consideration for the conversion of legacy AXI components for use in an ACE environment is the handling of Exclusive accesses. In a coherent environment, a monitor, which is associated with each master, is used for Shareable transactions to track whether another component has performed a store to an address that is being monitored for exclusivity. For non-cacheable transactions, a monitor that is remote from the master issuing an Exclusive access, is used to track all access to a location that is being monitored for exclusivity and this monitor can return a pass or fail response as part of the write response.

Therefore conversion of legacy AXI components for use in an ACE environment requires a monitor that is associated with the master interface that is being converted if the interface is capable of performing Exclusive accesses to Shareable locations.

D9.6 Transaction requirements

This section summarizes the requirements of transactions that are associated with Exclusive accesses.

The existing AXI rules apply for all transactions with **AxDOMAIN** set to Non-shareable or System Shareable.

For transactions with **ARDOMAIN** set to Inner Shareable or Outer Shareable:

- An Exclusive Load transaction must assert **ARLOCK**.
- An Exclusive Load transaction must use either ReadClean or ReadShared transaction type.

  Note

  It is not required that a transaction is issued for the execution of an `LDREX` instruction.

- Any slave that is capable of supporting Exclusive transactions must give an EXOKAY response to an Exclusive Load transaction.

  Note

  An OKAY response to an Exclusive Load transaction indicates that Exclusive transactions are not supported to that address location and all Exclusive Store transactions to that location will also return an OKAY response.

- An Exclusive Store transaction must assert **ARLOCK**.
- An Exclusive Store transaction must be a CleanUnique transaction.
- An Exclusive Store transaction response can be either EXOKAY or OKAY.
- Matching Exclusive Load transactions and Exclusive Store transactions are not required.

  Note

  An Exclusive Load transaction can occur with no matching Exclusive Store transaction. An Exclusive Store transaction can occur with no matching Exclusive Load transaction.

When multiple Exclusive-capable threads use a single interface:

- Transactions must use an AXI ID value that permits the Exclusive-capable thread to be identified.
- The bits of the AXI ID signal that are used to identify the Exclusive-capable thread must be the same for all Exclusive transactions from the same Exclusive-capable thread.

A single Exclusive-capable thread must not have an Exclusive Store transaction in progress at the same time as any transaction that registers that a master is performing an Exclusive sequence.

Chapter D10
Optional External Snoop Filtering

This chapter describes using an external snoop filter with an existing master component. It contains the following sections:

D10.1 About external snoop filtering

The ACE protocol supports a mechanism for constructing an optional external snoop filter that operates with an existing cached master component.

Note

External snoop filtering is optional. If a master component supports external snoop filtering, it must declare this in its data sheet.

To support the addition of a snoop filter, a cached master must ensure that it broadcasts sufficient information to permit a snoop filter to track Shareable allocations and evictions for cache lines that the master maintains. This ensures that a snoop filter can:

- Reduce the number of snoop transactions that are required to be passed to the master, which reduces cache intrusion.
- In certain circumstances, provide a faster response to snoop transactions.

For correct operation, a snoop filter must observe any transactions being issued by a master that could result in that master allocating a cache line in its local cache. The snoop filter must also observe activity that indicates an eviction from the local cache of that master. This includes:

- Local cache line evictions.
- WriteBack of cache lines to memory.
- Snoop transactions that cause an eviction.

A snoop filter can consider that a cache line is no longer allocated following a WriteEvict transaction.

Whether a transaction causes a cache line to be allocated depends on the transaction. A snoop filter can determine the expected allocation state of a particular cache line by observing the transactions that are issued by a master component. Table D10-1 shows the expected allocation state for both cache maintenance transactions and normal transactions. If the actual allocation for a cache line is different from what the filter expects, then an associated Evict operation must be performed to ensure that the snoop filter can correctly track the allocated cache line.

Table D10-1 External snoop filter expected cache line allocation states

Transaction	Expected cache line allocation
ReadOnce	Allocation does not change
ReadClean	Allocated
ReadNotSharedDirty	Allocated
ReadShared	Allocated
ReadUnique	Allocated
CleanUnique	Allocated
MakeUnique	Allocated
CleanShared	Allocation does not change
CleanInvalid	Evicted
MakeInvalid	Evicted
WriteUnique	Allocation does not change
WriteLineUnique	Allocation does not change
WriteClean	Allocation does not change

Table D10-1 External snoop filter expected cache line allocation states (continued)

Transaction	Expected cache line allocation
WriteBack	Evicted
WriteEvict	Evicted
Evict	Evicted

An error response to a transaction does not change the allocation behavior of a snoop filter. Any master that changes its allocation behavior when a transaction receives an error response must take appropriate action to ensure that the snoop filter is kept up to date.

D10.2 Master requirements to support snoop filters

The snoop filter monitors the snoop address and the snoop response channels to determine the effect of snoop transactions on the allocation of particular cache lines, and the IsShared response is used to determine if a cache line remains allocated in a cache after a snoop. A master component that is implementing snoop filter functionality must therefore provide an accurate IsShared response. See *Read response signaling* on page D3-186.

A master component must ensure that the transactions issued never indicate to the external snoop filter that a cache line is not allocated when the master still holds a copy of the cache line. A master component can ensure that the snoop filter always has correct allocation information using the following techniques.

This specification recommends that a master component must not:

- Issue a transaction that indicates that the cache line is to become allocated, while a transaction that indicates the same cache line is to be evicted is in progress.
- Issue a transaction that indicates that a cache line is to be evicted, while a transaction that indicates the same cache line is to be allocated is in progress.

If a master component does overlap allocating and evicting transactions, then the following must apply:

- From the first cycle that there is an overlapping allocating transaction and evicting transaction for the same cache line, the cached master must not have the line that is allocated.
- The line must remain de-allocated until a non-overlapping allocating transaction has completed.
- When the overlapping allocating and evicting transactions have all completed the allocation status of the line must be resolved, by issuing either:
  - An allocating transaction, to indicate that the line is allocated
  - An evicting transaction, to indicate that the line is de-allocated

D10.3 External snoop filter requirements

The snoop filter must ensure that it does not cause a capacity overflow by considering cache lines that are accessed using ReadNoSnoop and WriteNoSnoop transactions to be allocated. Such cache lines are Non-shareable locations, and master components are not required to issue Evict transactions to these locations.

Snoop filters must consider speculative reads. For example, a snoop filter cannot rely on an allocating transaction to determine whether it must add a cache line to its list of allocated cache lines. It must check the current list of allocated lines so that it does not hold two copies of the same line, potentially overflowing its resources.

The snoop filter must be able to track the allocation of all cache lines that could be allocated in the associated cache. Typically, the storage within a snoop filter will match the structure of the cache for which it is filtering snoops, in terms of:

- Associativity
- Size of the cache tags
- Total number of cache lines being tracked

Transactions in progress, and other caching structures within a master, can cause a situation where the snoop filter is required to track additional cache lines. A snoop filter can include additional storage to enable it to track these additional cache lines.

To avoid snoop filter overflow, where the tracking requirements exceed the total capabilities of the snoop filter, the snoop filter is permitted to issue snoop transactions. Transactions, such as CleanInvalid, are used to remove cache lines from the associated cache. Use of this technique ensures that the snoop filter can continue to correctly track all allocated cache lines.

Chapter D11
AMBA ACE-Lite

This chapter describes the ACE-Lite interface. It contains the following sections:

D11.1 About ACE-Lite

ACE-Lite is used by master components that do not have hardware coherent caches, but are required to:

- Indicate if issued transactions could be held in the hardware coherent caches of other masters.
- Issue barrier transactions.
- Issue broadcast cache maintenance operations.

ACE-Lite consists of an AXI4 interface with additional signals on the read address channel and write address channel. See *Read address channel (AR) signals* on page D2-170 and *Write address channel (AW) signals* on page D2-170 for more information.

ACE-Lite does not include:

- A snoop address channel
- A snoop response channel
- A snoop data channel
- A read acknowledge signal
- A write acknowledge signal
- The ACE-specific read response bits

D11.2 ACE-Lite signal requirements

An ACE-Lite interface can issue all Non-shareable transactions, but can only use a restricted set of Shareable transaction types.

Table D11-1 shows the permitted combinations of **ARSNOOP[3:0]**, **ARBAR[0]**, and **ARDOMAIN[1:0]** for each permitted category of Shareable read transaction.

Table D11-1 ACE-Lite permitted read address control signal combinations

Transaction type	ARSNOOP[3:0]	ARBAR[0]	ARDOMAIN[1:0]	Transaction
Non-snooping	0b0000	0b0	0b00 0b11	ReadNoSnoop
Coherent	0b0000	0b0	0b01 0b10	ReadOnce
Cache maintenance	0b1000	0b0	0b00 0b01 0b10	CleanShared
Cache maintenance	0b1001	0b0	0b00 0b01 0b10	CleanInvalid
Cache maintenance	0b1101	0b0	0b00 0b01 0b10	MakeInvalid
Barrier	0b0000	0b1	0b00 0b01 0b10 0b11	Barrier

Table D11-2 shows the permitted combinations of **AWBAR[0]**, and **AWDOMAIN[1:0]** for each permitted category of Shareable write transaction.

Table D11-2 ACE-Lite permitted write address control signal combinations

Transaction type	AWSNOOP[2:0]	AWBAR[0]	AWDOMAIN[1:0]	Transaction
Non-snooping	0b000	0b0	0b00 0b11	WriteNoSnoop
Coherent	0b000	0b0	0b01 0b10	WriteUnique
Coherent	0b001	0b0	0b01 0b10	WriteLineUnique
Barrier	0b000	0b1	0b00 0b01 0b10 0b11	Barrier

Chapter D12
Interface Control

This chapter describes the optional signals that can be used to configure the behavior of the ACE interface. It contains the following section:

D12.1 About the interface control signals

This section lists the signals that are available to configure interface behavior in components that support flexible interfaces.

Note

The signals in this section are optional, and are not part of the AMBA ACE protocol. However, if used, it is an architectural requirement that all interface control signals are stable, and remain static, on reset.

The AMBA ACE configuration signals are:

BROADCASTINNER

Tie-off input to control whether a master issues Inner Shareable transactions. If this signal is asserted, **BROADCASTOUTER** must also be asserted.

BROADCASTOUTER

Tie-off input to control whether a master issues Outer Shareable transactions.

BROADCASTCACHEMAINT

Tie-off input to control whether a master issues cache maintenance operations.

This signal controls the broadcast of cache maintenance operations and is asserted whenever a downstream cache exists below the coherent interconnect. Asserting this signal results in CleanShared, CleanInvalid, and MakeInvalid transactions that must be observed by a downstream cache being broadcast.

When this signal is deasserted, whether cache maintenance operations are broadcast depends on:

- Their shareability domain
- **BROADCASTINNER** and **BROADCASTOUTER** settings

Table D12-1 shows the valid combinations of the interface control signals and the corresponding transactions that are broadcast.

Table D12-1 Interface control signals

Signal			Transactions broadcast
BROADCASTINNER	**BROADCASTOUTER**	**BROADCASTCACHEMAINT**	
0	0	0	None
0	1	0	Outer Shareable
1	1	0	• Inner Shareable • Outer Shareable
0	0	1	Cache maintenance operations
0	1	1	• Outer Shareable • Cache maintenance operations
1	1	1	• Inner Shareable • Outer Shareable • Cache maintenance operations

Chapter D13
Distributed Virtual Memory Transactions

This chapter describes *Distributed Virtual Memory* (DVM) messages and the transactions that are used to pass them between components of a virtual memory system. It contains the following sections:

D13.1 About DVM transactions

DVM transactions are used to pass messages that support the maintenance of a virtual memory system.

There are two types of DVM transactions: DVM message and DVM Complete.

A DVM message supports the following operations:

- TLB Invalidate
- Branch Predictor Invalidate
- Physical Instruction Cache Invalidate
- Virtual Instruction Cache Invalidate
- Synchronization
- Hint

A DVM Complete transaction is issued in response to a DVM Synchronization message to indicate that all required operations and any associated transactions have completed.

DVM transactions are optionally supported on ACE, ACE5, and ACE5-LiteDVM interfaces, which can be designed to operate in one of the following ways:

- Initiate and receive DVM messages
- Only receive DVM messages

The DVM_Message_Support property is used to describe the types of messages that are supported by an interface. It can take one of the following values:

Bidirectional DVM transactions are supported in both directions on the interface. This requires the use of the AR/R and AC/CR channels.

Receiver DVM message and Synchronization transactions are supported from slave to master interfaces on the AC/CR channels.

DVM Complete transactions are supported from master to slave interfaces on the AR/R channels.

False DVM transactions are not supported.

For ACE, ACE5, and ACE5-LiteDVM interfaces, if DVM_Message_Support is not declared, it is considered to be Bidirectional.

DVM_Message_Support must be False or not declared for other interface types.

Interfaces with an address width smaller than 32 bits cannot support DVM messages, the DVM_Message_Support property must be False.

Table D13-1 describes the capability of master and slave interfaces with each value of DVM_Message_Support.

Table D13-1 Capability of master and slave interfaces

	Master	Slave
Bidirectional	• Initiate DVM requests on AR • Receive DVM Complete requests on AC • Receive DVM requests on AC • Issue DVM Complete requests on AR	• Receive DVM requests on AR • Issue DVM Complete requests on AC • Issue DVM requests on AC • Receive DVM Complete requests on AR
Receiver	• Receive DVM requests on AC • Issue DVM Complete requests on AR	• Issue DVM requests on AC • Receive DVM Complete requests on AR
False	No DVM requests can be issued or received.	No DVM requests can be issued or received.

D13.2 DVM transactions

A DVM transaction can be either a DVM message or DVM Complete.

D13.2.1 DVM message transactions

A DVM message can be sent in one or two transactions. Messages that include an address are sent using two transactions.

A one-part DVM message has one request transfer and one response transfer. The message is sent by the initiator master on the AR channel and is propagated to participating masters on the AC channel. Masters can use the Coherency Connection signaling to opt into receiving messages at runtime, see *Coherency Connection signaling* on page E1-364.

The flow of a DVM message is:

1. The originating master component issues the DVM message request on its read address channel.
2. The interconnect component distributes the request to participating components using the appropriate snoop address channel.
3. The request is not sent to the same component that issued the transaction.
4. Each participating component acknowledges receipt of the message using the snoop response channel. For ACE5-LiteDVM interfaces, this response must not be dependent on the forward progress of any transactions on the AR or AW channels.
5. The interconnect component collects the acknowledgements and responds to the original DVM request using the read data channel of the originating master component.

Figure D13-1 shows the flow when a master issues a DVM message to two other masters.

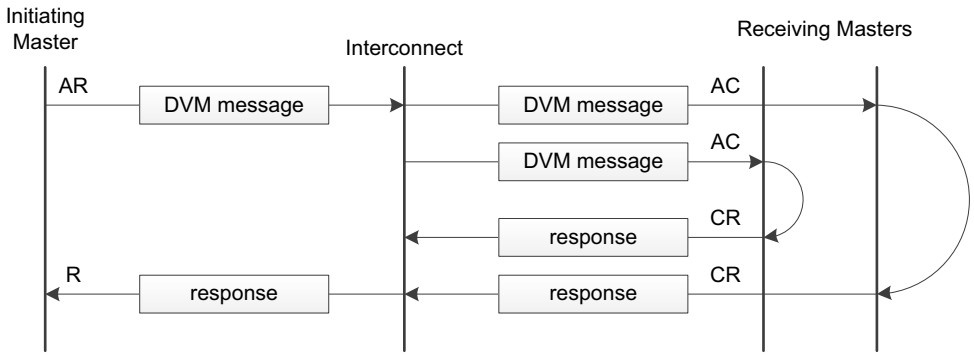


Figure D13-1 One-part DVM message flow

A two-part DVM message has two request transfers and two response transfers. Each transaction of a two-part DVM message has a response, both on the snoop response and read data channels. Figure D13-2 shows a two-part DVM message flow.

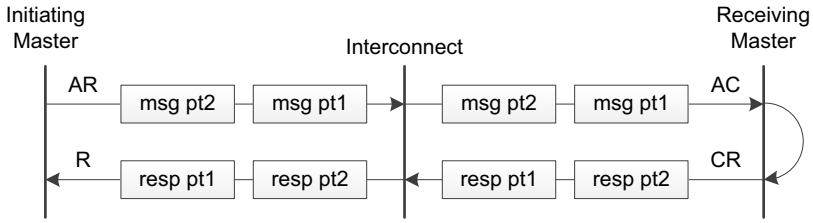


Figure D13-2 Two part DVM message flow

The following rules apply to two-part DVM messages:

- The requests are always sent as successive transfers, with no other transaction requests between them.
- Each transaction of a two-part DVM message must use the same AXI ID.
- A master issuing a two-part DVM message must be able to issue the latter parts of the message without requiring any other external actions.
- An interconnect component issuing a two-part DVM message on the snoop address channel must be able to issue the second part of the message without requiring a response to the first part of the message.

D13.2.2 DVM Synchronization and DVM Complete transactions

A DVM Synchronization message (Sync) has a response but also causes a DVM Complete transaction that indicates that the synchronization process is complete. The process is:

1. The initiating master issues the DVM Sync on its read address channel.
2. The interconnect component distributes the DVM Sync to participating masters using the appropriate snoop address channel. The transaction is not sent to the initiating master.
3. Each participating master acknowledges receipt of the DVM Sync using the snoop response channel. For ACE5-LiteDVM interfaces, this response must not be dependent on the forward progress of any transactions on the AR or AW channels.
4. The interconnect component collects the acknowledgements and responds to the original DVM Sync using the read data channel of the initiating master component.
5. Each receiving master must issue a DVM Complete when it has completed all the necessary actions, see *DVM Complete* on page D13-336. The DVM Complete is issued on the read address channel after the handshake of the associated DVM Sync on the snoop address channel of the same master.
6. The interconnect component can respond immediately to the DVM Complete transaction using the read data channel of the component that issued the DVM Complete.
7. When the interconnect component has received a DVM Complete from each participating component, it issues a DVM Complete using the snoop address channel of the initiating master.
8. The initiating master acknowledges the receipt of the DVM Complete using the snoop response channel.

Figure D13-3 shows the synchronization flow between an initiating master and one receiving master.

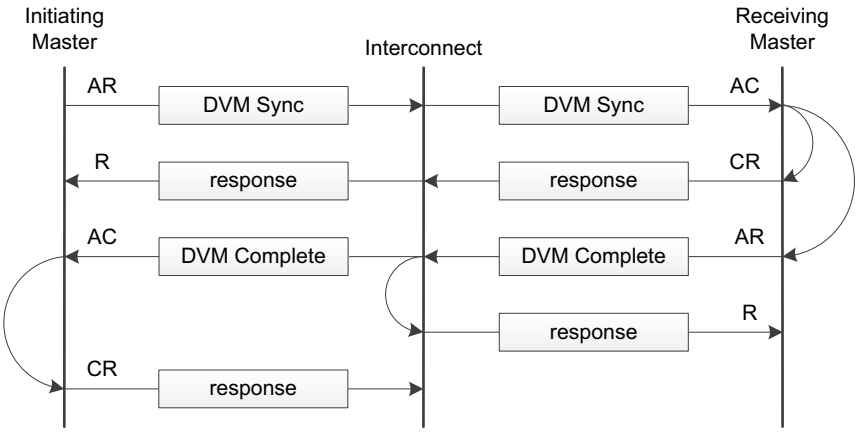


Figure D13-3 Synchronization flow between initiating master and receiving master

D13.2.3 DVM request attributes

Table D13-2 shows the constraints on AR and AC signals for DVM requests. If a signal is not listed, it is unconstrained.

Table D13-2 DVM transaction constraints

Attribute	Constraint
ARADDR	Must be zero for DVM Complete.
ARSNOOP	Must be either: • `0b1111` for a DVM Message • `0b1110` for a DVM Complete
ARBURST	Burst type must be INCR.
ARLEN	The burst length must be 1, which means **ARLEN[7:0]** must be `0x00`. See *Address structure* on page A3-48 for more information.
ARSIZE	The number of bytes in a transfer must be equal to the data bus width. See *Burst size* on page A3-49.
ARCACHE	Must be Modifiable and Non-cacheable, which means **ARCACHE[3:0]** must be `0b0010`. See Table A4-4 on page A4-68 for more information.
ARCHUNKEN	Must be zero
ARMMUATST	Must be zero
ARMMUFLOW	Must be `0b00`
ARTAGOP	Must be `0b00`
ARLOCK	Must be a normal access, which means **ARLOCK** must be zero.
ARDOMAIN	The domain must be Inner Shareable or Outer Shareable.

Table D13-2 DVM transaction constraints (continued)

Attribute	Constraint
ARBAR	Must be a normal access, which means **AxBAR[0]** must be zero.
ACADDR	Must be zero for DVM Complete.
ACSNOOP	Must be either: • `0b1111` for a DVM Message • `0b1110` for a DVM Complete
ACPROT	Not used for DVM messages, can take any value.

D13.2.4 DVM ID values

All the following rules apply to **ARID**/**RID** values for DVM transactions:

- DVM messages and non-DVM transactions on the read channels can use the same AXI ID value, if one transaction has fully completed before the other is issued.
- DVM messages are permitted to use the same AXI ID value as a transaction issued on the AW channel.
- Different DVM transactions can use either the same AXI ID, or different AXI IDs.
- If different AXI IDs are used for DVM requests, then the order of arrival of the different messages at the recipient of the message is not guaranteed.
- All DVM operations must be correctly ordered with respect to a DVM Sync from the same issuing master component, even if different AXI IDs are used.
- Each transaction of a two-part DVM message must use the same AXI ID.

D13.2.5 DVM responses

When a component receives a DVM request, it must respond as follows:

- If the component can perform the requested action, then it must respond by setting **CRRESP** to `0b00000`.
- If the component is unable to perform the requested action, then it must respond by setting **CRRESP** to `0b00010`. Typically, this response indicates an unsupported message.
- A component is not permitted to set **CRRESP** to `0b00010` in response to a DVM Sync or a DVM Complete.

The interconnect component gathers all response values and responds to the originator as follows:

- If **CRRESP** is `0b00000` for all responses, then the interconnect component sets **RRESP** to `0b0000`.
- If **CRRESP** is `0b00010` for any responses:
  - The interconnect component sets **RRESP** to `0b0010`. This response can be sent before all CR responses have been received.
  - This specification recommends that the interconnect component provides a fault log to record which components are unable to perform requested actions.

Other rules for DVM responses:

- No data transfer is associated with DVM transactions.
- For two-part DVM messages, the response to each transaction must be the same.
- The response to a DVM request must not be dependent on the forward progress of any transactions on the AR or AW channels.

D13.3 DVM messages

The following DVM messages are supported by the protocol:

- TLB Invalidate
- Branch Predictor Invalidate
- Physical Instruction Cache Invalidate
- Virtual Instruction Cache Invalidate
- Synchronization
- Hint

DVM messages are signaled using **ARSNOOP** or **ACSNOOP** set to `0b1111`.

DVM messages can require multiple transactions to convey the required information. In this case, the first transaction provides sufficient information to determine whether another transaction is required.

DVM transactions only operate on read-only structures, such as Instruction cache, Branch Predictor, and TLB, and therefore only invalidation operations are required. The concept of cleaning does not apply to a read-only structure. This means that it is functionally correct to invalidate more entries than the DVM message requires, although the extra invalidations can affect performance.

D13.3.1 DVM message versions

DVM messages were introduced in the Armv7 architecture and were extended in Armv8, Armv8.1 and Armv8.4 architectures. It is essential that interfaces initiating and receiving DVM messages support the same architecture versions.

The following properties define the version that is supported by an interface:

- DVM_v8
- DVM_v8.1
- DVM_v8.4 (ACE5-LiteDVM interfaces only)

Each property can take the values: True or False. If a property is not declared, then it is considered False.

Note

The only ACE interface that supports DVM_v8.4 is ACE5-LiteDVM

Table D13-3 shows which message versions are supported, depending on the property values. A component that supports DVM messages from a specific version must also support earlier architecture versions.

Table D13-3 DVM message versions

DVM property			Architecture support			
DVM_v8.4	**DVM_v8.1**	**DVM_v8**	**Armv8.4**	**Armv8.1**	**Armv8**	**Armv7**
True	True or False	True or False	Y	Y	Y	Y
False	True	True or False	-	Y	Y	Y
False	False	True	-	-	Y	Y
False	False	False	-	-	-	Y

In a system with heterogeneous components, it is the responsibility of the system configuration to determine the lowest common DVM specification that is supported in the system. The interconnect must have the knowledge of this lowest common denominator value by means of configuration, straps, parameterization, or some other IMPLEMENTATION DEFINED method.

To avoid deadlocks or denial of service the interconnect must detect unsupported DVM operations and suppress their propagation and respond in a protocol compliant manner. Error indication in such a response is optional.

D13.3.2 Addresses in DVM messages

DVM messages that require an address are transported using two transactions.

Table D13-4 shows the physical and virtual address space sizes that are supported.

Table D13-4 Supported physical and virtual address spaces

DVM version	Physical address space	Virtual address space
All versions	32-bit	32-bit
	40-bit	32-bit
v8 and above	40-bit	41-bit
	44-bit	49-bit
	48-bit	57-bit

If the physical address space exceeds the virtual address space, then virtual address operations might receive additional address information in a DVM message. In this case, any additional address information must be ignored and operations that are performed using only the supported address bits.

If a component supports a larger virtual address space than its physical address space, then a minimum ADDR_WIDTH is required to support the virtual address space size. The component must take appropriate action regarding the additional physical address bits. See *Address space size* on page D6-263 for more details.

Table D13-5 and Table D13-6 on page D13-325 show the allocation of address fields for the first and second parts of an invalidation by VA or IPA. The encodings are dependent on the physical and virtual address sizes. The address fields apply to the **ARADDR** and **ACADDR** signals.

Table D13-5 Address fields for the first of a two-part invalidation by VA or IPA

Address size		DVM version	First transaction address field mapping				
Physical	Virtual		[47:44]	[43:40]	[39:32]	[31:24]	[23:16]
32-bit	32-bit	All versions	-	-	-	VMID[7:0]	ASID[7:0]
40-bit	32-bit	All versions	-	-	SBZ	VMID[7:0]	ASID[7:0]
	41-bit	v8 and above	-	-	ASID[15:8]	VMID[7:0]	ASID[7:0]
44-bit	49-bit	v8 and above	-	VA[48:45]	ASID[15:8]	VMID[7:0]	ASID[7:0]
48-bit	57-bit	v8 and above	VA[56:53]	VA[48:45]	ASID[15:8]	VMID[7:0]	ASID[7:0]

Table D13-6 Address fields for the second of a two-part invalidation by VA or IPA

Address size		DVM version	Second transaction address field mapping					
Physical	Virtual		[47:44]	[43:40]	[39:32]	[31:12]	[11:4]	[3]
32-bit	32-bit	All versions	-	-	-	VA[31:12]	VA[11:4]	SBZ
40-bit	32-bit	All versions	-	-	SBZ	VA[31:12]	VA[11:4]	SBZ
	41-bit	v8 and above	-	-	VA[39:32]	VA[31:12]	VA[11:4]	VA[40]
44-bit	49-bit	v8 and above	-	VA[44:41]	VA[39:32]	VA[31:12]	VA[11:4]	VA[40]
48-bit	57-bit	v8 and above	VA[52:49]	VA[44:41]	VA[39:32]	VA[31:12]	VA[11:4]	VA[40]

Note

The bit positions for some higher-order address bits are both shifted by a single bit and also split between the first and second parts of a two-part DVM transaction. This bit position allocation might appear irregular, but is used to ease the translation between implementations with different physical address sizes.

For a message operating on a physical address, the first part contains the Virtual Address for *Virtually Indexed Physically Tagged* (VIPT) caches. Table D13-7 shows the first transaction, which contains the Virtual Address for VIPT caches.

Table D13-7 First transaction address mapping for any DVM operation, operating on a physical address

Physical address size	DVM version	First transaction address field mapping				
		[47:44]	[43:40]	[39:32]	[31:24]	[23:16]
32-bit	All versions	-	-	-	VA[27:20]	VA[19:12]
40-bit	All versions	-	-	SBZ	VA[27:20]	VA[19:12]
44-bit	v8 and above	-	SBZ	SBZ	VA[27:20]	VA[19:12]
48-bit	v8 and above	SBZ	SBZ	SBZ	VA[27:20]	VA[19:12]

Table D13-8 shows the second transaction, which contains the physical address.

Table D13-8 Second transaction address mapping for any DVM operation, operating on a physical address

Physical address size	DVM version	Second transaction address field mapping				
		[47:44]	[43:40]	[39:32]	[31:12]	[11:4]
32-bit	All versions	-	-	-	PA[31:12]	PA[11:4]
40-bit	All versions	-	-	PA[39:32]	PA[31:12]	PA[11:4]
44-bit	v8 and above	-	PA[43:40]	PA[39:32]	PA[31:12]	PA[11:4]
48-bit	v8 and above	PA[47:44]	PA[43:40]	PA[39:32]	PA[31:12]	PA[11:4]

D13.3.3 Support for 16-bit ASID

DVM_v8 and above support both 8-bit and 16-bit ASID. It cannot be determined from a DVM message whether the message uses an 8-bit or 16-bit VMID. All 8-bit ASID messages are required to set the ASID[15:8] field to zero.

It is expected that most systems will use a single ASID size across the entire system, either 8-bit ASID or 16-bit ASID.

In a system that contains a mix of 8-bit ASID and 16-bit ASID components, it is expected that all maintenance is done by an agent that uses 16-bit ASID. This ensures that the agent can perform maintenance on both the 8-bit ASID and 16-bit ASID components.

The interoperability requirements are:

- For an 8-bit ASID agent sending a message to a 16-bit ASID agent, a message appears as a 16-bit ASID with the upper 8 bits set to zero.
- For a 16-bit ASID agent sending a message to an 8-bit VMID agent:
  - If the upper 8 bits are zero, the message was received correctly.
  - If the upper 8 bits are nonzero, then over-invalidation will occur, since the 8-bit ASID agent ignores the upper 8 bits.

D13.3.4 Support for 16-bit VMID

DVM_v8.1 and above support both 8-bit and 16-bit VMIDs. It cannot be determined from a DVM message whether the message uses an 8-bit or 16-bit VMID. All 8-bit VMID messages are required to set the VMID[15:8] field to zero.

It is expected that most systems use a single VMID size across the entire system, either 8-bit VMID or 16-bit VMID.

In a system that contains a mix of 8-bit VMID and 16-bit VMID components, it is expected that all maintenance is done by an agent that uses 16-bit VMID. This ensures that the agent can perform maintenance on both the 8-bit VMID and 16-bit VMID components.

The interoperability requirements are:

- For an 8-bit VMID agent sending a message to a 16-bit VMID agent, a message appears as a 16-bit VMID with the upper 8 bits set to zero.
- For a 16-bit VMID agent sending a message to an 8-bit VMID agent:
  - If the upper 8 bits are zero, the message was received correctly.
  - If the upper 8 bits are nonzero, then over-invalidation will occur, since the 8-bit VMID agent ignores the upper 8 bits.

The following signals are defined for transporting the additional VMID bits:

Table D13-9 Signals for transporting the additional VMID bits

Signal	Source	Width	Description	Signal presence property condition
ARVMIDEXT	Master	4	Extension for 16-bit VMID on read address channel.	DVM_v8.1 and DVM_Message_Support is Bidirectional
ACVMIDEXT	Master	4	Extension for 16-bit VMID on snoop address channel.	DVM_v8.1 and DVM_Message_Support is not False

When using a 16-bit VMID, for a one-part message:

- VMID[11:8] is transported using **AxVMIDEXT[3:0]**
- VMID[15:12] is transported using **AxADDR[43:40]**

When using a 16-bit VMID, for a two-part message::

- VMID[11:8] is transported using **AxVMIDEXT[3:0]** in the first part
- VMID[15:12] is transported using **AxVMIDEXT[3:0]** in the second part

D13.3.5 DVM message encoding, first part

DVM message information is transported using **ARADDR**/**ARVMIDEXT** and **ACADDR**/**ACVMIDEXT**.

Table D13-10 shows how information is encoded for a one-part message or the first of a two-part message.

Table D13-10 Encoding for a one-part message or the first of a two-part message

AxADDR bits	Name	Function	
[47:44]	Virtual Address	VA[56:53][a]	
[43:40]	Virtual Address or VMID	VA[48:45][a] or VMID[15:12][b]	
[39:32]	ASID upper byte	ASID[15:8][a]	
[31:24]	VMID or Virtual Index	VMID[7:0] or VA[27:20]	
[23:16]	ASID or Virtual Index	ASID[7:0] or VA[19:12]	
[15]	Completion	**0b0**	DVM Complete transaction is not required
		0b1	DVM Complete transaction is required
[14:12]	Message type	**0b000**	TLB Invalidate
		0b001	Branch Predictor Invalidate
		0b010	Physical Instruction Cache Invalidate
		0b011	Virtual Instruction Cache Invalidate
		0b100	Synchronization
		0b101	Reserved
		0b110	Hint
		0b111	Reserved
[11:10]	Exception Level	**0b00**	Applies to Hypervisor and all Guest OS
		0b01	Applies to EL3[a]
		0b10	Applies to Guest OS
		0b11	Applies to Hypervisor
[9:8]	Security	**0b00**	Applies to Secure and Non-secure
		0b01	Applies to a Non-secure address from a Secure context[c]
		0b10	Applies to Secure only
		0b11	Applies to Non-secure only
[7]	SBZ or Range[c]	**0b0**	Message does not include address range information
		0b1	Message includes address range information
[6]	VMID or VA Valid	**0b0**	**AxADDR[31:24]** and **AxVMIDEXT[3:0]** must be zero, except for Hint operations.
		0b1	**AxADDR[31:24]** and **AxVMIDEXT[3:0]** contains VMID or VA information.

Table D13-10 Encoding for a one-part message or the first of a two-part message (continued)

AxADDR bits	Name	Function	
[5]	ASID or VA Valid	**0b0**	**AxADDR[23:16]** must be zero, except for Hint operations.
		0b1	Message includes ASID or VA information in **AxADDR[23:16]**.
[4]	Leaf	**0b0**	Invalidate all associated translations.
		0b1	Invalidate Leaf Entry only, that is the entry that is returned from the last level of the translation table walk.[a]
[3:2]	Stage	**0b00**	DVM_v7: Stage of invalidation varies with invalidation type. DVM_v8 and above: Stage 1 and Stage 2 invalidation
		0b01	Stage 1 only invalidation[a]
		0b10	Stage 2 only invalidation[a]
		0b11	Reserved
[1]	-	Reserved, SBZ	
[0]	Addr	**0b0**	The message does not require an address and has one part
		0b1	The message includes an address and has two parts

a. DVM_v8 and above
b. DVM_v8.1 and above
c. DVM_v8.4 and above

Address bits that are not used for a particular message must be driven to zero.

D13.3.6 DVM message encoding, second part

Table D13-11 shows how information is encoded for the second part of a two-part message.

Table D13-11 Information encoding for the second part of a two-part message

AxADDR bits	Description
[47:44]	VA[52:49] or PA[47:44]
[43:40]	VA[44:41] or PA[43:40]
[39:12]	VA[39:12] or PA[39:12]
[11:10]	VA[11:10] or PA[11:10] or TG[a]
[9:8]	VA[9:8] or PA[9:8] or TTL[a]
[7:6]	VA[7:6] or PA[7:6] or SCALE[a]
[5:4]	VA[5:4] or PA[5:4] or NUM[4:3][a]
[3]	VA[40]
[2:0]	NUM[2:0][a]

a. DVM_v8.4 and above

Address bits that are not used for a particular message must be driven to zero.

D13.3.7 TLB Invalidate operations in DVM v8.4

When the DVM_v8.4 property is True, the first and second part of a TLB Invalidate by IPA or VA contains additional information.

Table D13-12 shows DVM v8.4 additional information in the first part of a TLB Invalidate by IPA or VA.

Table D13-12 First part of a TLB Invalidate by IPA or VA

AxADDR bits	Name	Encoding	
[7]	Range	`0b0`	Message does not include address range information
		`0b1`	Message includes address range information.

Table D13-13 shows DVM v8.4 additional information in the second part of a TLB Invalidate by IPA or VA.

Table D13-13 Second part of a TLB Invalidate by IPA or VA

AxADDR bits	Name	Encoding	
[11:10]	Transaction granule, TG	`0b00`	No granule information
		`0b01`	Transaction_Granule_Size is 4KB
		`0b10`	Transaction_Granule_Size is 16KB
		`0b11`	Transaction_Granule_Size is 64KB
[9:8]	Transaction table level, TTL	`0b00`	No level hint information
		`0b01`	The leaf entry is on level 1 of the page table walk
		`0b10`	The leaf entry is on level 2 of the page table walk
		`0b11`	The leaf entry is on level 3 of the page table walk
[7:6]	SCALE	A constant used in the address range exponent calculation.	
[5:4]	NUM[4:3]	A constant used as a multiplication factor in the range calculation	
[2:0]	NUM[2:0]	A constant used as a multiplication factor in the range calculation	

If the DVM_v8.4 property is True and the message type is not TLB Invalidate by IPA or VA, the TG, TTL, SCALE and NUM bits must be zero.

The Range bit must be zero if either:

- DVM_v8.4 is not supported.
- The message type is not TLB Invalidate by IPA or VA.

The SCALE and NUM bits must be zero if the Range bit is deasserted.

When the Range bit is asserted, the range of addresses that are invalidated is determined by the following formula:

$$\texttt{BaseADDR} <= \texttt{VA} < \texttt{BaseADDR} + ((\texttt{NUM}+1) * 2^{(5*\texttt{SCALE}+1)} * \texttt{Translation\_Granule\_Size})$$

Where:

- `BaseADDR`: base address of the range, shifted based on the Transaction Granule (TG):
  - **4K** BaseADDR is VA[48:12]
  - **16K** BaseADDR is VA[50:14]
  - **64K** BaseADDR is VA[52:16]
- `SCALE` is a constant, it can take any value from 0-3.
- `NUM` is a constant, it can take any value from 0-31.

D13.3.8 TLB Invalidate

This section lists the *TLB Invalidate* (TLBI) operations that the DVM message supports.

Table D13-14 shows the fixed values for the TLBI message fields.

Table D13-14 Fixed values for the TLBI fields

AxADDR bits	Name	Value	Status
[15]	Completion	0b0	Completion not required
[14:12]	Message type	0b000	TLB Invalidate opcode
[1]	-	0b0	Reserved

Table D13-15 shows the TLBI messages.

Table D13-15 TLBI messages

Operation	AxADDR bit						
	[11:10] H'visor	[9:8] Secure	[6] VMID	[5] ASID	[4] Leaf	[3:2] Stage	[0] Addr
EL3 TLBI all[a]	0b01	0b10	0b0	0b0	0b0	0b00	0b0
EL3 TLBI by VA[a]	0b01	0b10	0b0	0b0	0b0	0b00	0b1
EL3 TLBI by VA, Leaf only[a]	0b01	0b10	0b0	0b0	0b1	0b00	0b1
Secure Guest OS TLBI by Non-secure IPA[b]	0b10	0b01	0b1	0b0	0b0	0b10	0b1
Secure Guest OS TLBI by Non-secure IPA, Leaf only[b]	0b10	0b01	0b1	0b0	0b1	0b10	0b1
Secure TLBI all	0b10	0b10	0b0	0b0	0b0	0b00	0b0
Secure TLBI by VA	0b10	0b10	0b0	0b0	0b0	0b00	0b1
Secure TLBI by VA, Leaf only[a]	0b10	0b10	0b0	0b0	0b1	0b00	0b1
Secure TLBI by ASID	0b10	0b10	0b0	0b1	0b0	0b00	0b0
Secure TLBI by ASID and VA	0b10	0b10	0b0	0b1	0b0	0b00	0b1
Secure TLBI by ASID and VA, Leaf only[a]	0b10	0b10	0b0	0b1	0b1	0b00	0b1
Secure Guest OS TLBI all[b]	0b10	0b10	0b1	0b0	0b0	0b00	0b0
Secure Guest OS TLBI by VA[b]	0b10	0b10	0b1	0b0	0b0	0b00	0b1
Secure Guest OS TLBI all, Stage 1 only[b]	0b10	0b10	0b1	0b0	0b0	0b01	0b0
Secure Guest OS TLBI by Secure IPA[b]	0b10	0b10	0b1	0b0	0b0	0b10	0b1
Secure Guest OS TLBI by VA, Leaf only[b]	0b10	0b10	0b1	0b0	0b1	0b00	0b1
Secure Guest OS TLBI by Secure IPA, Leaf only[b]	0b10	0b10	0b1	0b0	0b1	0b10	0b1
Secure Guest OS TLBI by ASID[b]	0b10	0b10	0b1	0b1	0b0	0b00	0b0
Secure Guest OS TLBI by ASID and VA[b]	0b10	0b10	0b1	0b1	0b0	0b00	0b1
Secure Guest OS TLBI by ASID and VA, Leaf only[b]	0b10	0b10	0b1	0b1	0b1	0b00	0b1

Table D13-15 TLBI messages (continued)

Operation	AxADDR bit						
	[11:10] H'visor	[9:8] Secure	[6] VMID	[5] ASID	[4] Leaf	[3:2] Stage	[0] Addr
All OS TLBI all	0b10	0b11	0b0	0b0	0b0	0b00	0b0
Guest OS TLBI all, Stage 1 and 2	0b10	0b11	0b1	0b0	0b0	0b00	0b0
Guest OS TLBI by VA	0b10	0b11	0b1	0b0	0b0	0b00	0b1
Guest OS TLBI all, Stage 1 only[a]	0b10	0b11	0b1	0b0	0b0	0b01	0b0
Guest OS TLBI by IPA[a]	0b10	0b11	0b1	0b0	0b0	0b10	0b1
Guest OS TLBI by VA, Leaf only[a]	0b10	0b11	0b1	0b0	0b1	0b00	0b1
Guest OS TLBI by IPA, Leaf only[a]	0b10	0b11	0b1	0b0	0b1	0b10	0b1
Guest OS TLBI by ASID	0b10	0b11	0b1	0b1	0b0	0b00	0b0
Guest OS TLBI by ASID and VA	0b10	0b11	0b1	0b1	0b0	0b00	0b1
Guest OS TLBI by ASID and VA, Leaf only[a]	0b10	0b11	0b1	0b1	0b1	0b00	0b1
Secure Hypervisor TLBI all[b]	0b11	0b10	0b0	0b0	0b0	0b00	0b0
Secure Hypervisor TLBI by VA[b]	0b11	0b10	0b0	0b0	0b0	0b00	0b1
Secure Hypervisor TLBI by VA, Leaf only[b]	0b11	0b10	0b0	0b0	0b1	0b00	0b1
Secure Hypervisor TLBI by ASID[b]	0b11	0b10	0b0	0b1	0b0	0b00	0b0
Secure Hypervisor TLBI by ASID and VA[b]	0b11	0b10	0b0	0b1	0b0	0b00	0b1
Secure Hypervisor TLBI by ASID and VA, Leaf only[b]	0b11	0b10	0b0	0b1	0b1	0b00	0b1
Hypervisor TLBI all	0b11	0b11	0b0	0b0	0b0	0b00	0b0
Hypervisor TLBI by VA	0b11	0b11	0b0	0b0	0b0	0b00	0b1
Hypervisor TLBI by VA, Leaf only[a]	0b11	0b11	0b0	0b0	0b1	0b00	0b1
Hypervisor TLBI by ASID[c]	0b11	0b11	0b0	0b1	0b0	0b00	0b0
Hypervisor TLBI by ASID and VA[c]	0b11	0b11	0b0	0b1	0b0	0b00	0b1
Hypervisor TLBI by ASID and VA, Leaf only[c]	0b11	0b11	0b0	0b1	0b1	0b00	0b1

a. DVM_v8 and above
b. DVM_v8.4 and above
c. DVM_v8.1 and above

D13.3.9 Branch Predictor Invalidate

This section lists the Branch Predictor Invalidate operations that the DVM message supports.

Table D13-16 shows the fixed values for the Branch Predictor Invalidate message fields.

Table D13-16 Fixed values for the Branch Predictor Invalidate message fields

AxADDR bits	Value	Status
[15]	0b0	Completion not required
[14:12]	0b001	Branch Predictor Invalidate opcode
[11:10]	0b00	Applies to all Guest OS and Hypervisor
[7]	0b0	No address range information
[6]	0b0	VMID field not valid
[5]	0b0	ASID field not valid
[4:1]	0b0000	Reserved

Table D13-17 shows the Branch Predictor Invalidate messages.

Table D13-17 Branch Predictor Invalidate messages

Operation	AxADDR [0]
Branch Predictor Invalidate all	0b0
Branch Predictor Invalidate by VA	0b1

D13.3.10 Instruction cache invalidations

Instruction caches can use either a physical address or a virtual address to tag the data they contain. A system might contain a mixture of both forms of cache.

The DVM protocol includes instruction cache invalidation operations that use physical addresses and operations that use virtual addresses. A component that receives DVM messages must support both forms of message, independent of the style of instruction cache implemented. It might be necessary to over-invalidate in the case where a message is received in a format that is not native to the cache type.

Physical Instruction Cache Invalidate

This section lists the *Physical Instruction Cache Invalidate* (PICI) operations that the DVM message supports. This message type is also used for Instruction Caches which are *Virtually Indexed Physically Tagged* (VIPT).

Table D13-18 shows the fixed values for the PICI message fields.

Table D13-18 Fixed values for the PICI message fields

AxADDR bits	Value	Description
[15]	0b0	Completion not required
[14:12]	0b010	Physical Instruction Cache Invalidate opcode

Table D13-18 Fixed values for the PICI message fields (continued)

AxADDR bits	Value	Description
[11:10]	0b00	Applies to all Guest OS and Hypervisor
[7]	0b0	No address range information
[4:1]	0b0000	Reserved

Table D13-19 shows the PICI messages.

Table D13-19 PICI messages

Operation	AxADDR bit		
	[9:8] Secure	[6:5] Virtual Index	[0] Addr
PICI all, Secure only	0b10	0b00	0b0
PICI by PA without Virtual Index, Secure only	0b10	0b00	0b1
PICI by PA with Virtual Index, Secure only	0b10	0b11	0b1
PICI all, Non-secure only	0b11	0b00	0b0
PICI by PA without Virtual Index, Non-secure only	0b11	0b00	0b1
PICI by PA with Virtual Index, Non-secure only	0b11	0b11	0b1

When Virtual Index is 0b11, then Virtual Index VA[27:12] at **AxADDR[29:14]** is used as part of the Physical Address.

Virtual Instruction Cache Invalidate

This section lists the *Virtual Instruction Cache Invalidate* (VICI) operations that the DVM message supports.

Table D13-20 shows the fixed values for the VICI message fields.

Table D13-20 Fixed values for VICI message fields

AxADDR bits	Value	Description
[15]	0b0	Completion not required
[14:12]	0b011	Virtual Instruction Cache Invalidate opcode
[7]	0b0	No address range information
[4:1]	0b0000	Reserved

Table D13-21 shows the VICI messages.

Table D13-21 VICI messages

Operation	AxADDR bit				
	[11:10] Level	[9:8] Secure	[6] VMID	[5] ASID	[0] Addr
Hypervisor and all Guest OS VICI all, Secure and Non-secure	0b00	0b00	0b0	0b0	0b0
Hypervisor and all Guest OS VICI all, Non-secure only	0b00	0b11	0b0	0b0	0b0
All Guest OS VICI by ASID and VA, Secure only	0b10	0b10	0b0	0b1	0b1
All Guest OS VICI by VMID, Secure only[a]	0b10	0b10	0b1	0b0	0b0
All Guest OS VICI by ASID, VA and VMID, Secure only[a]	0b10	0b10	0b1	0b1	0b1
All Guest OS VICI by VMID, Non-secure only	0b10	0b11	0b1	0b0	0b0
All Guest OS VICI by ASID, VA and VMID, Non-secure only	0b10	0b11	0b1	0b1	0b1
Hypervisor VICI by VA, Non-secure only	0b11	0b11	0b0	0b0	0b1
Hypervisor VICI by ASID and VA, Non-secure only[b]	0b11	0b11	0b0	0b1	0b1

a. DVM_v8.4 and above
b. DVM_v8.1 and above

D13.3.11 Synchronization

This section lists the Synchronization operation that the DVM message supports.

Table D13-22 shows the values for the Synchronization message fields.

Table D13-22 Values for Synchronization message fields

AxADDR bits	Value	Status
[15]	0b0	Completion not required
[14:12]	0b100	Synchronization opcode
[11:10]	0b00	Applies to all Guest OS and Hypervisor
[9:8]	0b00	Applies to Secure and Non-secure
[7]	0b0	No address range information
[6]	0b0	VMID field not valid
[5]	0b0	ASID field not valid
[4:1]	0b0000	Reserved
[0]	0b0	No address specified in this message

D13.3.12 Hint

A reserved message address space is provided for future Hint messages.

Table D13-23 shows the fixed values for the Hint message fields.

Table D13-23 Hint message fields

AxADDR bits	Value	Status
[15]	`0b0`	Completion not required
[14:12]	`0b110`	Hint opcode

Hint messages must have a response with **CRRESP** set to `0b00000`.

D13.4 DVM Complete

A DVM Complete transaction is sent when a component has received a DVM Sync message and all preceding invalidation operations are complete. The following rules apply in determining when an operation is complete:

TLB Invalidate

Complete when a master can no longer use an invalidated translation and all previous transactions that could have used an invalidated translation are complete.

Branch Predictor Invalidate

Complete when cached copies of predicted instruction fetches have been invalidated and can no longer be accessed by the associated master. The invalidated cached copies might be from any virtual address or from a specified virtual address.

Instruction Cache Invalidate

Complete when cached instructions have been invalidated and can no longer be accessed by the associated master.

A component must have only one outstanding DVM Sync transaction. A component must receive a DVM Complete transaction before it issues another DVM Sync transaction.

Components must be able to accept DVM Sync messages and continue processing snoop transactions while waiting for earlier transactions to complete. This processing might be needed before a DVM Complete message can be sent. The maximum number of outstanding DVM Sync messages that a master must be able to accept is 256.

A DVM Sync must complete in a timely manner, even if the component continues to receive more DVM invalidation operations and more DVM Sync messages.

Support for multiple outstanding DVM Sync messages only requires the component to be aware of the number of DVM Complete responses required. No additional information about the individual DVM Sync messages is necessary.

A component that has issued a DVM Sync and also received a DVM Sync from another component is not permitted to wait for the DVM Complete to its DVM Sync before providing a DVM Complete response for the other component.

Chapter D14
Master Design Recommendations

This chapter presents a set of recommendations for the design of master components that improve the ability to bridge the master to different protocol interfaces. It contains the following sections:

D14.1 Recommended design restrictions

This specification recommends that all new master components are designed to meet the following restrictions:

- A single cache line size of 64 bytes.
- A constrained number of WriteBack, WriteClean, WriteEvict, and WriteNoSnoop transactions in progress:
  - The total number of data bytes within each transaction must be considered as well as the total number of transactions.
  - There is no fixed limit, it is only required that the limit is specified. This permits a buffer to be designed which can accept the maximum number of transactions.
- A snoop transaction must make forward progress unless there is an outstanding WriteBack, WriteClean, or WriteEvict transaction to the same line.
- Any address hazard that prevents forward progress of a snoop transaction while a WriteBack, WriteClean, or WriteEvict transaction is in progress must be precise to cache line granularity:
  - It is not permitted to prevent forward progress of a snoop transaction while a WriteBack, WriteClean, or WriteEvict transaction is in progress to a different cache line and there is no WriteBack, WriteClean, or WriteEvict transaction in progress to the same cache line.
- The use of the CD channel to respond to snoops must be supported if the cache holds dirty cache lines:
  - This is required to permit the forward progress of a snoop transaction when the maximum number of WriteBack, WriteClean, WriteEvict, and WriteNoSnoop transactions have been reached and these transactions are not guaranteed to complete before the snoop must complete.
  - Partial dirty cache lines cannot be supported because the CD channel does not support the use of byte strobes.
- All WriteBack, WriteClean, WriteEvict, and Evict transactions in progress must use a unique AXI ID transaction identifier. This allows the interconnect to respond to WriteBack, WriteClean, WriteEvict, and Evict transactions in any order.
- The single-copy atomicity guarantee for a Device transaction is no greater than the number of bytes in a single data transfer, as defined by **AxSIZE**.

This set of restrictions is sufficient to permit a master component to be bridged to a protocol that does not support the free-flowing write channel that ACE provides.

This set of restrictions has no impact on the compatibility of the master with different revisions of the specification.

Part E

AMBA 5 Protocol Features

Chapter E1
Additional Features in AMBA 5

This chapter describes features that can be used with AMBA 5 interfaces:

All these features are optional. Table G3-1 on page G3-472 shows which features can be included on which interface type.

E1.1 Atomic transactions

AMBA 5 introduces Atomic transactions, which perform more than just a single access and have an operation that is associated with the transaction. Atomic transactions enable sending the operation to the data, permitting the operation to be performed closer to where the data is located. Atomic transactions are suited to situations where the data is located a significant distance from the agent that must perform the operation.

Compared with using Exclusive Accesses, this approach reduces the amount of time during which the data must be made inaccessible to other agents in the system.

The Atomic_Transactions property is used to indicate whether a component supports Atomic transactions:

True Atomic transactions are supported.

False Atomic transactions are not supported. If not declared, Atomic_Transactions property is considered False.

The Atomic_Transactions extension is supported in the following interfaces:

- AXI5
- ACE5-Lite
- ACE5-LiteDVM

If a slave or interconnect component declares that it supports Atomic transaction, then it must support all operation types, sizes, and endianness.

This specification does not support the use of Atomic transactions by ACE5 masters.

E1.1.1 Overview

In an atomic transaction, the master sends an address, control information, and outbound data. The slave sends inbound data (except for AtomicStore) and a response. This specification supports four forms of Atomic transaction:

AtomicStore

- Sends a single data value with an address and the atomic operation to be performed.
- The target performs the operation using the sent data and value at the addressed location as operands.
- The result is stored in the address location.
- A single response is given without data.
- Outbound data size is 1, 2, 4, or 8 bytes.

AtomicLoad

- Sends a single data value with an address and the atomic operation to be performed.
- The original data value at the addressed location is returned.
- The target performs the operation using the sent data and value at the addressed location as operands.
- The result is stored in the address location.
- Outbound data size is 1, 2, 4, or 8 bytes.
- Inbound data size is the same as the outbound data size.

AtomicSwap

- Sends a single data value with an address.
- The target swaps the value at the addressed location with the data value that is supplied in the transaction.
- The original data value at the addressed location is returned.
- Outbound data size is 1, 2, 4, or 8 bytes.
- Inbound data size is the same as the outbound data size.

AtomicCompare

- Sends two data values, the compare value and the swap value, to the addressed location. The compare and swap values are of equal size.
- The data value at the addressed location is checked against the compare value:
  - If the values match, the swap value is written to the addressed location.
  - If the values do not match, the swap value is not written to the addressed location.
- The original data value at the addressed location is returned.
- Outbound data size is 2, 4, 8, 16, or 32 bytes.
- Inbound data size is half of the outbound data size because the outbound data contains both compare and swap values, whereas the inbound data has only the original data value.

E1.1.2 Atomic transaction operations

This specification supports eight different operations which can be used with AtomicStore and AtomicLoad transaction. Table E1-1 shows the operators.

Table E1-1 Atomic transaction operators

Operator	Description
ADD	The value in memory is added to the sent data and the result stored in memory.
CLR	Every set bit in the sent data clears the corresponding bit of the data in memory.
EOR	Bitwise exclusive OR of the sent data and value in memory.
SET	Every set bit in the sent data sets the corresponding bit of the data in memory.
SMAX	The value stored in memory is the maximum of the existing value and sent data. This operation assumes signed data.
SMIN	The value stored in memory is the minimum of the existing value and sent data. This operation assumes signed data.
UMAX	The value stored in memory is the maximum of the existing value and sent data. This operation assumes unsigned data.
UMIN	The value stored in memory is the minimum of the existing value and sent data. This operation assumes unsigned data.

E1.1.3 Atomic transactions attributes

Rules for atomic transactions:

- **AWLEN** and **AWSIZE** specifies the number of bytes of write data in the transaction. For AtomicCompare, the number of bytes must include both the compare and swap values.
- If **AWLEN** indicates a burst length greater than one, **AWSIZE** is required to be the full data bus width.
- Write strobes that are not within the data window, as specified by **AWADDR** and **AWSIZE**, must be deasserted.
- Write strobes within the data window must be asserted.

For AtomicStore, AtomicLoad, and AtomicSwap:

- The write data is 1, 2, 4, or 8 bytes and read data is 1, 2, 4, or 8 bytes respectively.
- **AWADDR** must be aligned to the data size.
- **AWBURST** must be INCR.

For AtomicCompare:

- The write data is 2, 4, 8, 16, or 32 bytes and read data is 1, 2, 4, 8, or 16 bytes.
- **AWADDR** must be aligned to a single write data value, half the total write data size.
- If **AWADDR** points to the lower half of the transaction:
  - The compare value is sent first. The compare value is in the lower bytes of a single-beat transaction, or in the first beats of a multi-beat transaction.
  - **AWBURST** must be INCR.
- If **AWADDR** points to the upper half of the transaction:
  - The swap value is sent first. The swap value is in the lower bytes of a single-beat transaction, or in the first beats of a multi-beat transaction.
  - **AWBURST** must be WRAP.

For AtomicCompare transactions, there are relaxations to the usual rules for transactions of type WRAP:

- A burst of length 1 is permitted, **AWLEN**=0
- **AWADDR** is not required to be aligned to the transfer size

Example E1-1 shows some permitted combinations of attributes for a 64-bit data bus and the location of the Compare and Swap data values.

Example E1-1 Location of the Compare and Swap data values

AWADDR	AWSIZE	AWLEN	AWBURST	7 6 5 4 3 2 1 0
0x00	1 (2B)	0	INCR	- - - - - - S C
0x01	1 (2B)	0	WRAP	- - - - - - C S
0x04	2 (4B)	0	INCR	S S C C - - - -
0x06	2 (4B)	0	WRAP	C C S S - - - -
0x00	3 (8B)	0	INCR	S S S S C C C C
0x04	3 (8B)	0	WRAP	C C C C S S S S
0x00	3 (8B)	1	INCR	C C C C C C C C 1st Beat S S S S S S S S 2nd Beat
0x08	3 (8B)	1	WRAP	S S S S S S S S 1st Beat C C C C C C C C 2nd Beat

E1.1.4 ID use for Atomic transactions

A single AXI ID is used for an Atomic transaction. The same AXI ID is used for the request, write response, and the read data. This requirement means that the master must only use ID values that can be signaled on both **AWID** and **RID** signals.

Atomic transactions must not use AXI ID values that are used by Non-atomic transactions that are outstanding at the same time. This rule applies to transactions on either the AR or AW channel. This rule ensures that there are no ordering constraints between Atomic transactions and Non-atomic transactions.

If one transaction has fully completed before the other is issued, Atomic transactions and Non-atomic transactions can use the same AXI ID value.

Multiple Atomic transactions that are outstanding at the same time must not use the same AXI ID value.

E1.1.5 Request attributes for Atomic transactions

For Atomic transactions, the following restrictions apply for request attributes:

- **AWCACHE** and **AWDOMAIN** are permitted to be any combination valid for the interface type. See Table D3-3 on page D3-177.
- **AWSNOOP** must be set to all zeros.
- **AWLOCK** must be `0b0`, Normal access.

E1.1.6 Atomic transaction signaling

An extra signal is added to the interface to support Atomic transactions.

The signal is *AW Atomic Operation*, **AWATOP**. Table E1-2 and Table E1-3 on page E1-346 show the **AWATOP** encodings.

Table E1-2 AWATOP encodings

AWATOP[5:0]	Description
`0b000000`	Non-atomic operation
`0b01exxx`	AtomicStore
`0b10exxx`	AtomicLoad
`0b110000`	AtomicSwap
`0b110001`	AtomicCompare

For AtomicStore and AtomicLoad transactions **AWATOP[3]** indicates the endianness that is required for the atomic operation:

- When deasserted, this bit indicates that the operation is little-endian.
- When asserted, this bit indicates that the operation is big-endian.

The value of **AWATOP[3]** applies to arithmetic operations only and is ignored for bitwise logical operations.

For AtomicStore and AtomicLoad transactions, Table E1-3 shows the encodings for the operations on the lower-order **AWATOP[2:0]** signals.

Table E1-3 Lower-order AWATOP[2:0] encodings

AWATOP[2:0]	Operation	Description
0b000	ADD	Add
0b001	CLR	Bit clear
0b010	EOR	Exclusive OR
0b011	SET	Bit set
0b100	SMAX	Signed maximum
0b101	SMIN	Signed minimum
0b110	UMAX	Unsigned maximum
0b111	UMIN	Unsigned minimum

E1.1.7 Transaction structure

For AtomicLoad, AtomicSwap, and AtomicCompare transactions, the transaction structure is as follows:

- The request is issued on the AW channel.
- The associated transaction data is sent on the W channel.
- The number of write data transfers required on the W channel is determined by the **AWLEN** signal.
- The relative timing of the Atomic transaction request and the Atomic transaction write data is not specified.
- The slave returns the original data value using the R channel.
- The number of read data transfers is determined from both **AWLEN** and the **AWATOP** signals.

  Note

  For the AtomicCompare operation, if **AWLEN** indicates a burst length greater than 1, then the number of read data transfers is half that specified by **AWLEN**.

- A slave is permitted to wait for all write data before sending read data. A master must be able to send all write data without receiving any read data.
- A slave is permitted to send all read data before accepting any write data. A master must be able to accept all read data without any write data being accepted.
- A single write response is returned on the B channel. The write response must be given by the slave only after it has received all write data transfers and the result of the atomic transaction is observable.

Figure E1-1 shows the flow of information and data for AtomicLoad, AtomicSwap, and AtomicCompare transactions.

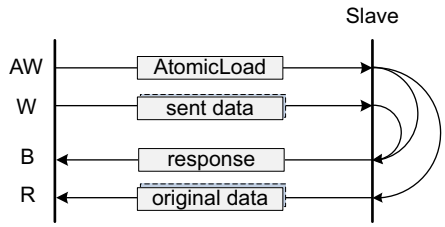


Figure E1-1 AtomicLoad, AtomicSwap, or AtomicCompare transaction

For AtomicStore transactions, the transaction structure is as follows:

- The request is issued on the AW channel.
- The associated transaction data is sent on the W channel.
- The number of write data transfers required on the W channel is determined by the **AWLEN** signal.
- The relative timing of the Atomic transaction request and the Atomic transaction write data is not specified.
- A single write response is returned on the B channel. The write response must be given only by the slave after it has received all write data transfers and the result of the atomic transaction is observable.

Figure E1-2 shows the flow of information and data for AtomicStore transactions.

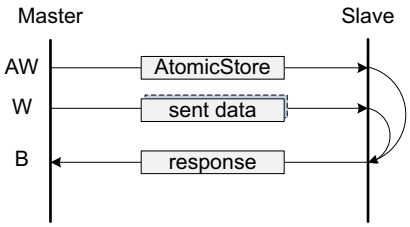


Figure E1-2 AtomicStore transaction

E1.1.8 Response signaling

An Atomic transaction requires a write response. The write response indicates that the transaction is visible to all required observers.

Atomic transactions that include a read response are visible to all required observers from the point of receiving the first item of read data.

Note

Both the read response and write response indicate that a transaction is visible to all required observers. It is permitted for a master to use either response.

There is no concept of an error that is associated with the operation, such as overflow. An operation is fully specified for all input combinations.

For transactions, such as AtomicCompare, where there are multiple outcomes for the transaction, no indication is provided on the outcome of the transaction. To determine if a Compare and Swap instruction has updated the memory location, it is necessary to inspect the original data value that is returned as part of the transaction.

It is permitted to give an error response to an Atomic transaction when the transaction reaches a component that does not support Atomic transactions. For AtomicLoad, AtomicSwap and AtomicCompare transactions:

- A slave must send the correct number of read data beats, even if the write response is DECERR or SLVERR.
- A master might ignore the write response and only use the response that comes with read data.

E1.1.9 Atomic transaction dependencies

For AtomicLoad, AtomicSwap, and AtomicCompare transactions, Figure E1-3 on page E1-349 shows the following Atomic transaction handshake signal dependencies:

- The master must not wait for the slave to assert **AWREADY** or **WREADY** before asserting **AWVALID** or **WVALID**.
- The slave can wait for **AWVALID** or **WVALID**, or both, before asserting **AWREADY**.
- The slave can assert **AWREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave can wait for **AWVALID** or **WVALID**, or both, before asserting **WREADY**.
- The slave can assert **WREADY** before **AWVALID** or **WVALID**, or both, are asserted.
- The slave must wait for **AWVALID**, **AWREADY**, **WVALID**, and **WREADY** to be asserted before asserting **BVALID**.

  The slave must also wait for **WLAST** to be asserted before asserting **BVALID**, because the write response **BRESP**, must be signaled only after the last data transfer of a write transaction.
- The slave must not wait for the master to assert **BREADY** before asserting **BVALID**.
- The master can wait for **BVALID** before asserting **BREADY**.
- The master can assert **BREADY** before **BVALID** is asserted.
- The slave must wait for both **AWVALID** and **AWREADY** to be asserted before it asserts **RVALID** to indicate that valid data is available.
- The slave must not wait for the master to assert **RREADY** before asserting **RVALID**.
- The master can wait for **RVALID** to be asserted before it asserts **RREADY**.
- The master can assert **RREADY** before **RVALID** is asserted.
- The master must not wait for the slave to assert **RVALID** before asserting **WVALID**.
- The slave can wait for **WVALID** to be asserted, for all write data transfers, before it asserts **RVALID**.
- The master can assert **WVALID** before **RVALID** is asserted.

In the dependency diagram that Figure E1-3 shows:

- A single-headed arrow points to a signal that can be asserted before or after the signal at the start of the arrow.
- A double-headed arrow points to a signal that must be asserted only after assertion of the signal at the start of the arrow.

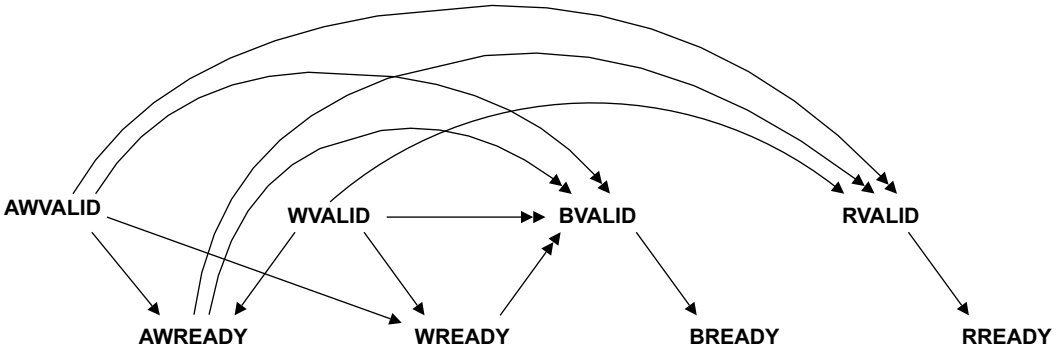


Figure E1-3 Atomic transaction handshake dependencies

E1.1.10 Support for Atomic transactions

Master support

Atomic transactions are not supported for ACE masters. An ACE master is able to perform an atomic operation to a Cacheable location by obtaining a unique copy of the cache line and performing the atomic operation locally within its own cache. An ACE master cannot support Atomic transactions to Non-cacheable or Device locations. No specific support for Atomic transactions is required on the Snoop channel and therefore an ACE master needs no added functionality to be compatible with Atomic transactions that are performed by other components.

A master component that supports Atomic transactions is required to support a mechanism to suppress the generation of Atomic transactions to ensure compatibility in systems where Atomic transactions are not supported. An optional **BROADCASTATOMIC** pin is specified. When the pin is tied HIGH, the interface is permitted to generate Atomic transactions. When tied LOW, the interface must not generate Atomic transactions.

Slave support

It is optional for a slave component to support Atomic transactions.

If a slave component only supports Atomic transactions for particular memory types, or for particular address regions, then the slave must give an appropriate error response for the Atomic transactions that it does not support.

Interconnect support

It is optional for an interconnect to support Atomic transactions.

If an interconnect does not support Atomic transactions, all attached master components must be configured to not generate Atomic transactions. The **BROADCASTATOMIC** pin can be used for this purpose.

Atomic transactions, can be supported at any point within an interconnect that supports them, including passing Atomic transactions downstream to slave components.

Atomic transactions are not required to be supported for every address location. If Atomic transactions are not supported for a given address location, then an appropriate error response can be given for the transaction. See *Response signaling* on page E1-347.

For Device transactions, the Atomic transaction must be passed to the endpoint slave. If the slave is configured to indicate that it does not support Atomic transactions, then the interconnect must give an error response for the transaction. An Atomic transaction must not be passed to a component that does not support Atomic transactions.

For Cacheable transactions, the interconnect can either:

- Perform the atomic operation within the interconnect. This method requires that the interconnect performs the appropriate read, write, and snoop transactions to complete the operation.
- If the appropriate endpoint slave is configured to indicate that it does support atomic operations, then the interconnect can pass the atomic operation to the slave.

E1.2 Cache stashing

Cache stashing enables one component to indicate that a particular cache line should be placed in the cache of another component in the system. This technique can be used to ensure that data is located close to its point of use, potentially improving the performance of the overall system.

The Cache_Stash_Transactions property is used to indicate whether an interface supports cache stashing:

True Cache stashing is supported.

False Cache stashing is not supported and associated signals are omitted. If Cache_Stash_Transactions is not declared, it is considered False.

The Cache_Stash_Transactions extension is supported in the following interfaces:

- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

This specification does not support the use of cache stashing by, or into, ACE5 masters. ACE5-Lite masters can cause data to be stashed in fully coherent masters with AMBA CHI interfaces. For more information on stashing into a CHI master, see *AMBA 5 CHI Architecture Specification*.

An identification mechanism is required with the transaction to use cache stashing. The identification indicates the specific cache in the system that is the intended target for the stash operation. This specification does not define the precise details of this identification mechanism. It is expected that any agent that is performing a stash operation knows the identifier to use for a given stash transaction.

This specification does define two levels of identification, one to identify the physical interface that the cache stash should be sent to, and one to identify a functional unit that is associated with that physical interface. For example, a stash transaction can specify a processor cluster interface and specific cache within that cluster.

E1.2.1 Stash transaction types

This specification defines four stash transaction types:

WriteUniquePtlStash

A write to memory which also indicates that the data should be allocated into a particular cache. For a WriteUniquePtlStash transaction, any number of bytes within the cache line are written, including all bytes or zero bytes.

WriteUniqueFullStash

A write to memory which also indicates that the data should be allocated into a particular cache. For a WriteUniqueFullStash transaction, it is required that all bytes within the cache line are written.

StashOnceShared A data-less transaction which indicates that a cache line should be fetched into a particular cache. For a StashOnceShared transaction, it is required that existing cached copies of the cache line are not invalidated.

StashOnceUnique A data-less transaction which indicates that a cache line should be fetched into a particular cache. For a StashOnceUnique transaction, this specification recommends that the cache line is stashed in Unique state. Stashing in a Unique state permits a store to the cache line to occur with no further action.

Note

A StashOnceUnique transaction can cause the invalidation of a cached copy of a cache line and care must be taken to ensure that such transactions do not interfere with Exclusive access sequences.

For an interface that supports the Untranslated_Transactions feature, an extra stash transaction is supported. The StashTranslation transaction is used to indicate to a *System Memory Management Unit* (SMMU) that a translation should be obtained for the address that is supplied with the StashTranslation transaction. See *StashTranslation* on page E1-374.

E1.2.2 Stash transaction signaling

An additional set of signaling is provided on the ACE5-Lite interface to support the use of cache stashing. This includes the extension of the **AWSNOOP** signal to 4 bits.

A stash transaction is sent using the AW channel, with or without an associated transfer on the W channel. The permitted combinations of control signals for stash requests is shown in Table E1-4. WriteUniqueStash and StashOnce transactions must not cross a cache line boundary.

Table E1-4 Permitted Stash transaction write address control signal combinations

Stash transaction	AWSNOOP	AWBAR[0] AWLOCK	AWDOMAIN	AWCACHE[1]	AWLEN AWSIZE
WriteUniquePtlStash	0b1000	0b0	0b01, 0b10	0b1	Cache line or smaller
WriteUniqueFullStash	0b1001	0b0	0b01, 0b10	0b1	Cache line sized and Regular
StashOnceShared	0b1100	0b0	0b00, 0b01, 0b10	0b1	Cache line sized and Regular
StashOnceUnique	0b1101	0b0	0b00, 0b01, 0b10	0b1	Cache line sized and Regular

Table E1-5 shows the additional signals that are required on the AW channel to support stash transactions.

Table E1-5 Additional AW channel signaling

Signal	Description
AWSTASHNID[10:0]	Node Identifier of the target for a stash operation
AWSTASHNIDEN	Indicates whether the AWSTASHNID signal is valid
AWSTASHLPID[4:0]	Logical Processor Identifier within the target for a stash operation
AWSTASHLPIDEN	Indicates whether the **AWSTASHLPID** signal is valid

The following rules apply to the AW channel signaling associated with the Cache_Stash_Transactions property:

- **AWSTASHNID** and **AWSTASHNIDEN** must either be both present or both absent.
- **AWSTASHLPID** and **AWSTASHLPIDEN** must either be both present or both absent.
- **AWSTASHNID[10:0]** must be driven to all zeros when **AWSTASHNIDEN** is deasserted.
- **AWSTASHLPIDEN** must be driven to all zeros when **AWSTASHLPIDEN** is deasserted.

It is permitted, but not recommended, to send a stash transaction with a stash target that indicates a component that does not support cache stashing. The indication of a stash target within a stash transaction does not affect which components are permitted to access and cache a given cache line.

Table E1-6 shows the permitted combinations of the enable signals associated with the Node and Logical Processor identifiers.

Table E1-6 Permitted combinations of the enable signals

AWSTASHNIDEN	AWSTASHLPIDEN	Permitted behavior
0	0	Required value for any transaction that is not a WriteUniqueStash or StashOnce transaction. Permitted for a WriteUniqueStash or StashOnce transaction.
1	0	Permitted for a WriteUniqueStash or StashOnce transaction. Only the physical interface that is the target for the stash operation is provided.
0	1	Permitted for a WriteUniqueStash or StashOnce transaction. This combination is only expected to be used on an ACE5-LiteACP interface, where the Node ID is not required. See Chapter F4 *ACE5-LiteACP*.
1	1	Permitted for a WriteUniqueStash or StashOnce transaction.

E1.2.3 Rules and recommendations

It is permitted to send a stash transaction without a stash target. In this situation, this specification recommends the following behavior for each of the different types of stash transaction:

- For WriteUniquePtlStash and WriteUniqueFullStash transactions:
  - If the interconnect is able to determine that the cache line is held in a single cache before the write occurs, then stash the cache line back to that cache.
  - If the cache line is not held in any cache before the write occurs, then stash the cache line in a shared system cache.
- For StashOnceShared and StashOnceUnique transactions, if the interconnect is able to determine that the cache line is not in any cache, then stash the cache line in a shared system cache.

Note

- For StashOnceShared or StashOnceUnique transactions, care is required to avoid any action that could result in the deallocation of the cache line from the cache where it is expected to be used.
- A common use of StashOnce without a stash target is for a component to prefetch a cache line to a downstream cache for its own use.

E1.2.4 Transaction structure

A WriteUniqueStash has the same transaction structure as other WriteUnique transactions.

A StashOnce transaction has no data transfers. The address and control information is provided on the AW channel, and a single response is provided on the B channel. The response must be provided only after the address has been accepted.

E1.2.5 ID use for stash transactions

WriteUniquePtlStash and WriteUniqueFullStash transactions impose no additional constraints on the use of AXI ID values.

StashOnceShared and StashOnceUnique can be referred to as StashOnce transactions. StashOnce transactions must not use the same AXI ID values that are used by non-StashOnce transactions that are outstanding at the same time. This rule ensures that there are no ordering constraints between StashOnce transactions and other transactions. Both StashOnce transactions and non-StashOnce transactions are permitted to use the same AXI ID value, provided that the same ID value is not used by both a StashOnce transaction and a non-StashOnce at the same time. There can be multiple outstanding StashOnce transactions with the same ID. There can be multiple outstanding non-StashOnce transactions with the same ID.

Note

The use of a unique ID value for a StashOnce transaction ensures that these transactions can be given an immediate response if they are not supported.

E1.2.6 Support for stash transactions

The Cache_Stash_Transactions property is used to indicate whether a component supports stash transactions.

Table E1-7 shows the conversion of transactions between components that issue stash transactions and components that do not support them.

Table E1-7 Conversion between Stash and Non-stash transactions

Stash transaction	Action
WriteUniquePtlStash	Convert to WriteUnique, optionally named WriteUniquePtl.
WriteUniqueFullStash	Convert to WriteLineUnique, optionally named WriteUniqueFull.
StashOnceShared	Do not propagate and give an immediate response.
StashOnceUnique	Do not propagate and give an immediate response.

Note

See *Full and partial cache line write transaction naming* on page G1-458 for a description of WriteUniqueFull and WriteUniquePtl.

E1.3 Deallocating transactions

The primary use of a deallocating transaction is to deallocate the associated cache lines when it is known that these cache lines are no longer required. This mechanism helps to ensure better availability of the cache resources for other address locations.

The DeAllocation_Transactions property is used to indicate whether a component supports deallocating transactions:

True Deallocating transactions are supported.

False Deallocating transactions are not supported. If the DeAllocation_Transactions property is not declared, it is considered False.

The Deallocation_Transactions extension is supported in the following interfaces:

- ACE5-Lite
- ACE5-LiteDVM

This specification does not support the use of deallocating transactions by ACE5 masters.

Interoperability between a component that issues deallocating transactions and a component that does not support them can be performed by converting the transaction to a ReadOnce transaction.

E1.3.1 Deallocating transaction types

This specification defines two deallocating transactions:

ReadOnceCleanInvalid (ROCI)

This transaction reads a snapshot of the current value of the cache line. This specification recommends, but does not require, that any cached copy of the cache line is deallocated. If a Dirty copy of the cache line exists, and the cache line is deallocated, then the Dirty copy must be written back to main memory.

ReadOnceMakeInvalid (ROMI)

This transaction reads a snapshot of the current value of the cache line. This specification recommends, but does not require, that any cached copy of the cache line is deallocated. It is permitted, but not required, that a Dirty copy of the cache line is discarded. The Dirty copy of the cache line does not need to be written back to main memory.

E1.3.2 Rules and recommendations

Deallocating transactions are only permitted to access one cache line at a time. Accessing less than a cache line is permitted, but it is not permitted to cross a cache line boundary.

Note

Use of a ReadOnceMakeInvalid transaction to access less than a cache line can result in the invalidation of the entire cache line.

For a ReadOnceMakeInvalid transaction, it is required that the invalidation of the cache line is committed before the return of the first item of read data for the transaction. The invalidation of the cache line is not required to have completed at this point. However, it must be ensured that any later write transaction from any agent that starts after this point, is guaranteed not to be invalidated by this transaction.

The following considerations apply to the use of these transactions:

- Caution is needed when deallocating transactions are issued to the same cache line that other agents are using for Exclusive accesses. This is because the deallocation can cause an exclusive sequence to fail.
- Apart from the interaction with Exclusive accesses, the ReadOnceCleanInvalid transaction only provides a hint for deallocation of a cache line and has no other impact on the correctness of a system.

- The use of the ReadOnceMakeInvalid transaction can cause the loss of a Dirty cache line. The use of this transaction must be strictly limited to scenarios when it is known that it is safe to do so.
- These transactions do not guarantee the invalidation of cache lines and cannot be used to ensure the visibility of downstream caches.

Note

This specification permits the use of ReadOnceCleanInvalid and ReadOnceMakeInvalid transactions to access less than a cache line. However, some implementations might not support the deallocation behavior for transactions that are less than a cache line and instead convert the transaction to ReadOnce in such cases.

E1.3.3 Deallocating transaction signaling

A deallocating transaction is sent using the AR channel. A deallocating transaction is indicated using the extended **ARSNOOP** encodings that are shown in Table E1-8.

Table E1-8 ARSNOOP encodings

ARSNOOP	Transaction
0b0100	ReadOnceCleanInvalid
0b0101	ReadOnceMakeInvalid

These transactions are only supported for transactions to the Inner or Outer Shareable domain.

The permitted response and permitted cache line state changes for these transactions is identical to the permitted response and permitted cache line state changes for ReadOnce transactions.

There is no snoop channel equivalent of these transactions. An interconnect is permitted to use any appropriate snoop transaction to obtain the required data and deallocate the cache line.

Conversion between a component that issues deallocation transactions and one that does not support them can be performed by simply converting the transaction to a ReadOnce transaction.

E1.4 Trace signals

An optional Trace signal can be associated with each channel to support the debugging, tracing, and performance measurement of systems.

The Trace_Signals property is used to indicate whether a component supports Trace signals:

True Trace_Signals are supported.

False Trace_Signals are not supported. If Trace_Signals is not declared, it is considered False.

The Trace_Signals extension is supported in the following interfaces:

- AXI5
- AXI5-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

Table E1-9 shows the Trace signal that is associated with each channel.

Table E1-9 Trace signal associated with each channel

Trace signal	Channel	Description
ARTRACE	AR	Associated with the Read Address channel
RTRACE	R	Associated with the Read Data channel
AWTRACE	AW	Associated with the Write Address channel
WTRACE	W	Associated with the Write Data channel.
BTRACE	B	Associated with the Write Response channel
ACTRACE[a]	AC	Associated with the Snoop Address channel
CRTRACE[a]	CR	Associated with the Snoop Response channel
CDTRACE[b]	CD	Associated with the Snoop Data channel

a. ACE5 and ACE5-LiteDVM only.
b. ACE5 only.

If the Trace_Signals property is True, then the appropriate Trace signal must be present for all channels that are present.

The expected use of the Trace signal is as follows:

- A component, such as a master or an interconnect, can assert the Trace signal along with the address of a transaction that should be tracked through the system.
- This specification expects that any component that provides a response to a transaction with the Trace signal asserted in the request provides a response with the Trace signal asserted.
- For transactions that have the Trace signal asserted and which generate extra related transactions, such as snoop transactions, this specification recommends asserting the Trace signal for the related transactions:
  - Any related transaction using the same address, such as a snoop transaction, has the Trace signal propagated to it.
  - For other transactions, which might have an unrelated address, it is IMPLEMENTATION DEFINED whether the Trace signal is propagated.

It is permitted for an interconnect or slave component to use Trace signals.

It is permitted for a component to assert the Trace signal of a transaction response for a transaction that did not have the Trace signal asserted in the request. In this case, it is not required that the Trace signal is asserted for all responses of the same transaction.

This specification recommends that all behavior relating to the propagation of the Trace signaling is adopted, but this recommendation is not a requirement. Therefore, any component that uses the Trace signaling must not always require the correct propagation of the Trace signaling.

This specification expects that the use of Trace signaling is coordinated across the entire system and only one use of the Trace signaling occurs at a given time.

For Write transactions the following behavior is recommended:

- A slave that receives a write request with **AWTRACE** asserted should assert the **BTRACE** signal alongside the write response.
- **WTRACE** should be propagated through interconnect components.
- For Atomic transactions that require a response on the read channel, the **RTRACE** signal should be asserted if **AWTRACE** was asserted.

For Read transactions the following behavior is recommended:

- A slave that receives a read request with the **ARTRACE** signal asserted should assert the **RTRACE** signal alongside every beat of the read response.

For Snoop transactions the following behavior is recommended:

- A master that receives a snoop request with the **ACTRACE** signal asserted should assert the **CRTRACE** signal alongside the snoop response.The master should also assert **CDTRACE** alongside every data beat of the snoop data that is associated with the snoop transaction.

E1.5 User Loopback signaling

User Loopback signaling permits an agent that is issuing transactions to store information that is related to the transaction in an indexed table. The response to the transaction can then use a fast table index to obtain the required information, rather than requiring a more complex lookup that uses the transaction **AxID**.

The Loopback_Signals property is used to indicate whether a component supports User Loopback signals:

True Loopback signals are supported.

False Loopback signals are not supported. If Loopback_Signals is not declared, it is considered False.

The Loopback_Signals extension is supported in the following interfaces:

- AXI5
- ACE5
- ACE5-Lite
- ACE5-LiteDVM

Table E1-10 shows the User Loopback signals.

Table E1-10 User Loopback signals

Signal	Width	Description
ARLOOP	LOOP_R_WIDTH	Loopback value for a read transaction
AWLOOP	LOOP_W_WIDTH	Loopback value for a write transaction
RLOOP	LOOP_R_WIDTH	Returns the value that is provided on **ARLOOP**
BLOOP	LOOP_W_WIDTH	Returns the value provided on **AWLOOP**

If the Loopback_Signals property is True, then all loopback signals must be present. See Table E1-10.

The usage rules and recommendations are:

- LOOP_R_WIDTH and LOOP_W_WIDTH are recommended to be configurable up to 8 bits wide.
- The value of **RLOOP** must be identical to the value that was presented on the **ARLOOP** signal.
- The value of **BLOOP** must be identical to the value that was presented on the **AWLOOP** signal.
- If an interface includes **BCOMP**, **BLOOP** can take any value for responses with **BCOMP** deasserted.
- For Atomic transactions that require a response on the read channel, the value of **RLOOP** must be identical to the value that was presented on the **AWLOOP** signal. This requirement means that master must use loop values that can be signaled on both **AWLOOP** and **RLOOP** signals.

This specification does not require that the loopback value is unique. Multiple outstanding transactions from the same master are permitted to use the same value.

This specification does not require that the loopback value is preserved as a transaction progresses through a system. An intermediate component is permitted to store the loopback value of a transaction it receives and use its own loopback value for a transaction that it propagates downstream. When the component receives a response to the downstream transaction, it can retrieve the loopback value that is required for the response to the original transaction.

Loopback signaling is not supported on the snoop channels. All snoop transaction responses are required to be in order, which simplifies the process of associating a response with a request.

E1.6 QoS Accept signaling

AXI4 introduced two interface signals to indicate the QoS value of a transaction. AMBA 5 introduced two additional interface signals that enable a slave to indicate the minimum QoS value of transactions that it accepts. The QoS_Accept property is used to indicate whether an interface includes these signals:

True The interface includes both **VARQOSACCEPT** and **VAWQOSACCEPT** signals.

False The interface does not include **VARQOSACCEPT** or **VAWQOSACCEPT**. If QoS_Accept is not declared, it is considered False.

The QoS_Accept extension is applicable to the following interfaces:

- AXI5
- ACE5
- ACE5-Lite
- ACE5-LiteDVM

QoS Accept signaling is intended for slave components that have different resources available for different QoS levels, which is typically the case with memory controllers. The slave can indicate that it only accepts transactions at a certain QoS level or above when the resources available to lower QoS levels are in use.

QoS Accept signaling can be used as an input to a master interface that might have several different transactions to select from. This permits the master interface to only issue transactions that are likely to be accepted, which avoids unnecessary blocking of the interface. By preventing the issue of transactions that might be stalled for a significant period, the interface remains available for the issue of higher priority transactions that might arrive at a later point in time. The two signals are shown in Table E1-11:

Table E1-11 QoS Accept signals

Signal	Description
VARQOSACCEPT[3:0]	QoS acceptance level for read transactions
VAWQOSACCEPT[3:0]	QoS acceptance level for write transactions.

Each signal is an output from a slave and an input to a master that indicates the QoS value for which the slave accepts transactions. Any transactions at this QoS level or higher are accepted by the slave. Any transaction below this QoS level might be stalled for a significant time.

Note

This specification does not define a time period during which the slave is required to accept a transaction at, or above, the QoS level indicated. However, it is expected that for a given slave there will be a deterministic maximum number of clock cycles taken to accept a transaction, after taking into account implementation aspects such as clock domain crossing ratios.

In this specification, the term **VAxQOSACCEPT** refers collectively to the **VARQOSACCEPT** and **VAWQOSACCEPT** signals.

It is permitted for a master interface to issue a transaction that is below the QoS level indicated by the **VAxQOSACCEPT** signal. However, such a transaction might be stalled for a significant time.

It is permitted for a slave interface to accept a transaction that is below the QoS level indicated by the **VAxQOSACCEPT** signal, but it is expected that the transaction might be subject to a significant delay.

While it is acceptable for a slave to delay a transaction that has a lower priority than the QoS Acceptance level, this specification recommends that such a transaction is not delayed indefinitely. There are several reasons for a lower-priority transaction to be issued on the interface, for example:

- A delay between a change in the QoS Acceptance value and the ability of the component to adapt to that change.
- A requirement to make progress on a transaction that is Head-of-Line Blocking a higher priority transaction.
- A requirement to make progress on a transaction for reasons of starvation prevention.

The **VAxQOSACCEPT** signal is synchronous to the interface, but it is unrelated to any other AXI channel.

The default value for the **VAxQOSACCEPT** signals is zero.

E1.7 Wake-up Signaling

The wake-up signals are used to indicate that there is activity associated with the interface. These are:

- **AWAKEUP**
- **ACWAKEUP**

The Wakeup_Signals property is used to indicate whether a component supports wake-up signaling:

True Wake-up signals are supported.

False Wake-up signals are not supported. If Wakeup_Signals is not declared, it is considered False.

The signals can be routed to a clock controller, or similar component, to enable power and clocks to the connected components. The wake-up signals must be glitch-free and generated directly from a register. They are synchronous to the interface that it relates to, but are appropriate for crossing clock domains to a controller.

Wake-up signals must be asserted to guarantee that a transaction can be accepted, but once the transaction is in progress the assertion or deassertion of the wake-up signal is IMPLEMENTATION DEFINED. This specification recommends, but does not require, that the wake-up signal be deasserted when no further transactions are required.

E1.7.1 AWAKEUP rules and recommendations

The **AWAKEUP** signal is applicable to interfaces:

- AXI5
- AXI5-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

AWAKEUP is an output signal from a master interface and is asserted at the start of a transaction to indicate that there is a transaction to be processed:

- This specification recommends that **AWAKEUP** is asserted at least one cycle before the assertion of **ARVALID**, **AWVALID**, or **WVALID** to prevent the acceptance of a new transaction being delayed.
- It is permitted for **AWAKEUP** to be asserted at any point before or after the assertion of **ARVALID**, **AWVALID**, or **WVALID**.
- A slave is permitted to wait for **AWAKEUP** to be asserted before asserting **ARREADY**, **AWREADY**, or **WREADY**.
- If **AWAKEUP** is asserted in a cycle where **AWVALID** is asserted and **AWREADY** is deasserted, then **AWAKEUP** must remain asserted until **AWREADY** is asserted.
- If **AWAKEUP** is asserted in a cycle when **ARVALID** is asserted and **ARREADY** is deasserted, then **AWAKEUP** must remain asserted until **ARREADY** is asserted.
- After the **ARVALID**, **ARREADY** handshake, or the **AWVALID**, **AWREADY** handshake, the interconnect must remain active until the transaction has completed.
- It is required that the **AWAKEUP** signal is asserted to guarantee progress of a transition on the Coherency Connection signaling. See *Coherency Connection signaling* on page E1-364:
  - It is permitted for **AWAKEUP** to be asserted at any point before or after the assertion of **SYSCOREQ**. However, it is required to be asserted to guarantee the corresponding assertion of **SYSCOACK**. When **AWAKEUP** is asserted with **SYSCOREQ** asserted and **SYSCOACK** deasserted, it must remain asserted until **SYSCOACK** is asserted.
  - It is permitted for **AWAKEUP** to be asserted at any point before or after the deassertion of **SYSCOREQ**. However, it is required to be asserted to guarantee the corresponding deassertion of **SYSCOACK**. When **AWAKEUP** is asserted with **SYSCOREQ** deasserted and **SYSCOACK** asserted, it must remain asserted until **SYSCOACK** is deasserted.

- It is permitted, but not recommended, to assert **AWAKEUP** then deassert it without a transaction taking place.

Note

There is no requirement relating to the assertion of **AWAKEUP** relative to **WVALID**. However, for components that can assert **WVALID** before **AWVALID**, the assertion of **AWAKEUP** at least one cycle before **WVALID** can prevent the acceptance of a new transaction being delayed.

If a slave has an **AWAKEUP** input, but the attached master does not have an **AWAKEUP** output, then either:

- Tie **AWAKEUP** high, however this might prevent the slave interface from using low-power states.
- Derive **AWAKEUP** from **AxVALID** and **SYSCOREQ**/**ACK**. This method enables the slave to use low-power states, but might introduce latency while the clock is enabled.

E1.7.2 ACWAKEUP rules and recommendations

The **ACWAKEUP** signal is only applicable to:

- ACE5
- ACE5-LiteDVM

ACWAKEUP is an output signal from an interconnect interface and is asserted at the start of a snoop transaction to indicate that there is a transaction to be processed. This rule applies to either a normal coherency snoop transaction or a DVM snoop transaction:

- This specification recommends that **ACWAKEUP** is asserted at least one cycle before the assertion of **ACVALID** to prevent the acceptance of a new snoop transaction being delayed unnecessary.
- **ACWAKEUP** must remain asserted until the associated **ACVALID** / **ACREADY** handshake to ensure progress of the snoop transaction.
- After the **ACVALID** / **ACREADY** handshake, the master must remain active until the snoop transaction has completed.
- It is permitted for **ACWAKEUP** to be asserted at any point before or after the assertion of **ACVALID**.
- It is permitted, but not recommended, to assert **ACWAKEUP** and then deassert it without **ACVALID** being asserted.

E1.8 Coherency Connection signaling

A four-phase Coherency Connection signaling scheme is added, which can safely cross clock domains. These signals are used by a master to connect and disconnect from a coherency domain. When a master is connected to the coherency domain, it might receive snoop requests or DVM messages on the AC channel. When disconnected, no snoop requests or DVM messages are received.

The Coherency_Connection_Signals property is used to indicate whether a component supports the additional signals:

True Coherency Connection signaling is supported.

False Coherency Connection signaling is not supported. If not declared, Coherency_Connection_Signals property is considered False.

Coherency Connection signaling is only applicable to

- ACE5
- ACE5-LiteDVM

A master must be connected to a coherency domain before it can cache locations that must be kept hardware-coherent. A master can disconnect from a coherency domain when it no longer holds cache lines that must be kept hardware-coherent. When disconnected from a coherency domain, the master does not receive snoop transactions and therefore does not need to provide any snoop responses. Disconnecting from a coherency domain is typically used before entering a low-power state in which snoop transactions cannot be processed.

The connection to, or disconnection from, a coherency domain includes both normal coherency transactions, and DVM transactions that are sent on the snoop channel. Throughout the rest of this section, the connection to, or disconnection from, a coherency domain applies to whichever of these transaction types are applicable to the component.

The following two signals are used for coherency connect and disconnect signaling:

SYSCOREQ Coherency connect request.

SYSCOACK Coherency connect acknowledge.

The usage rules are:

- **SYSCOREQ** and **SYSCOACK** must either be both present or both absent.
- No default signaling is associated with **SYSCOREQ** and **SYSCOACK** signaling.

When disconnected from coherency, a master must not issue allocating transactions to Shareable memory. The following transactions are permitted:

- IO Coherent transactions:
  - ReadOnce
  - WriteUnique
- Cache Maintenance Operation transactions:
  - CleanShared
  - CleanSharedPersist
  - CleanInvalid
  - MakeInvalid
- All Non-shareable transactions

Note

RACK and **WACK** signaling is still used when a component is disconnected from coherency.

E1.8.1 Coherency Connection Handshake

SYSCOREQ and **SYSCOACK** must be deasserted when **ARESETn** is asserted. When not in reset, the following requests are permitted:

- A master requests to be connected to system coherency by asserting **SYSCOREQ** HIGH. The interconnect indicates that coherency is enabled by asserting **SYSCOACK** HIGH.
- The master requests disconnection from system coherency by deasserting **SYSCOREQ** LOW. The interconnect indicates that coherency is disabled by deasserting **SYSCOACK** LOW.

Requests to enter and exit coherency are always initiated by the master.

Figure E1-4 shows the system coherency interface handshake timing:

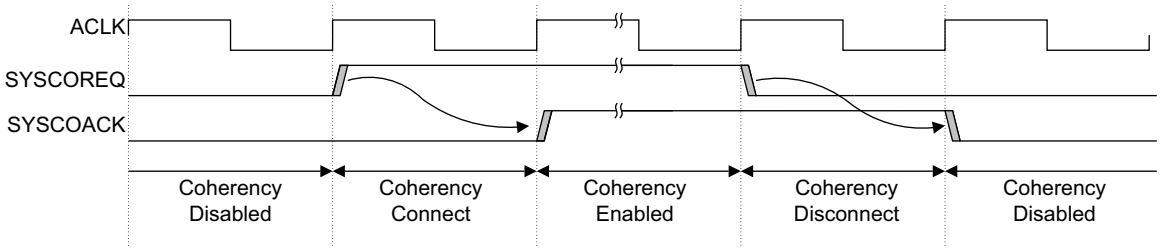


Figure E1-4 System coherency interface handshake timing

The interface signaling obeys the four-phase handshake rules:

- A master can only change **SYSCOREQ** when **SYSCOACK** is at the same level.
- An interconnect can only change **SYSCOACK** when **SYSCOREQ** is at the opposite level.

Master rules

A master:

- Must be able to respond to snoop transactions when it asserts **SYSCOREQ** HIGH.
- Must not issue a transaction that permits it to cache a coherent location until it observes **SYSCOACK** HIGH.
- Must not hold any cached copies of a coherent location when it deasserts **SYSCOREQ** LOW. A snoop that is received by the master, after it has deasserted **SYSCOREQ**, must give a snoop response indicating that the cache line is invalid.
- Must be able to respond to snoop transactions until it observes **SYSCOACK** LOW.

Interconnect rules

An interconnect:

- Must be able to service transactions to a coherent location when it asserts **SYSCOACK** HIGH.
- Must have completed all snoop transactions to this interface before it deasserts **SYSCOACK** LOW.

The transactions that would permit a coherent location to be cached are:

- ReadUnique
- ReadClean
- ReadNotSharedDirty
- ReadShared
- CleanUnique
- MakeUnique

E1.8.2 Coherency Connection signaling states

Figure E1-5 shows the state diagram for the Coherency Connection signaling.

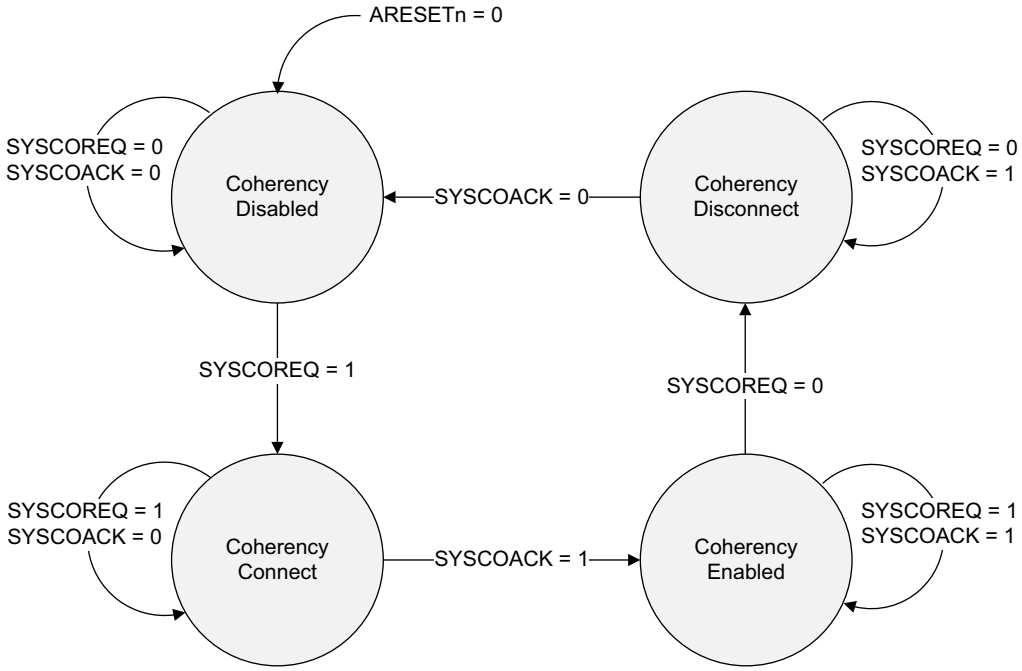


Figure E1-5 Coherency Connection signaling state diagram

Table E1-12 shows a summary of the states that are associated with the system coherency interface and the requirements for the master and the interconnect.

Table E1-12 Coherency Connection signaling states

State	SYSCOREQ	SYSCOACK	Description
Disabled	0	0	Master: • Must not hold any cached copies of coherent locations. • Must not issue transactions that allow a coherent location to be cached. • Not required to respond to snoop transactions. Interconnect: • Not required to service transactions that allow a coherent location to be cached. • Must not issue snoop transactions.
Connect	1	0	Master: • Must not issue transactions that allow a coherent location to be cached. • Must respond to snoop transactions. Interconnect: • Not required to service transactions to a coherent location.
Enabled	1	1	Master: • Can issue transactions that allow a coherent location to be cached. • Must respond to snoop transactions. Interconnect: • Must service transactions to a coherent location. • Can issue snoop transactions.
Disconnect	0	1	Master: • Must not hold any cached copies of coherent locations. • Must not issue transactions that allow a coherent location to be cached. • Must respond to snoop transactions. Interconnect: • Not required to service transactions that allow a coherent location to be cached. • Can complete all required snoop transactions.

E1.8.3 Coherency Connection signaling and DVM messages

A master that supports DVM can:

- Issue DVM messages on its AR channel.
- Receive DVM messages on its AC channel.
- Issue a DVM Complete message on its AR channel, in response to a DVM Sync received on its AC channel.

A master must not issue any DVM messages, except DVM Complete, on its AR channel after it has deasserted **SYSCOREQ**.

An interconnect must not issue any new DVM messages on the AC channel after it has deasserted **SYSCOACK**. It is permitted to deassert **SYSCOACK** when all DVM requests on the AC snoop channel have completed, including the second part of a 2-part message.

If an interconnect has sent a DVM Sync message that requires a DVM Complete message on the AR channel, then the interconnect is permitted to deassert **SYSCOACK** when all DVM requests on the AC snoop channel have completed. The master is still required to send the DVM Complete transaction on the AR channel, even when coherency is fully disconnected.

E1.8.4 Incompatible support for Coherency Connection signaling

Coherency Connection signaling does not have a default set of values that can be used. If one side of an interface supports Coherency Connection signaling and the other side does not, then a third-party component, such as a power controller, must be connected to the Coherency Connection signaling. This component is required to coordinate the Coherency Connection signaling and it must ensure that the requirements of the signaling are met.

E1.9 Untranslated transactions

AMBA 5 extends support of an SMMU by providing a means to identify untranslated transactions.

The Untranslated_Transactions property is used to indicate whether a component supports the required signals.

v2	The required signals are supported for version 2 of untranslated transactions.
v1	The required signals are supported for version 1.
True	The required signals are supported for version 1.
False	The required signals are not supported. If Untranslated_Transactions is not declared, it is considered False.

The Untranslated Transactions version 1 extension is applicable to the following interfaces:

- AXI5
- ACE5
- ACE5-Lite

The Untranslated Transactions version 2 extension is applicable to the following interfaces:

- AXI5
- ACE5-Lite

Support in ACE5 is for version 1 only, and has additional restrictions. See *Use of Untranslated Transactions with ACE5* on page E1-372.

Address translation is the process of translating an input address to an output address based on address mapping and memory attribute information that is held in translation tables. This process permits agents in the system to use their own virtual address space, but ensures that the addresses for all transactions are eventually translated to a single physical address space for the entire system.

The use of a single physical address space is required for the correct operation of hardware coherency and therefore the SMMU functionality is typically located before a coherent interconnect.

The additional signals that are specified in this section provide sufficient information for an SMMU to determine the translation that is required for a particular transaction and permit different transactions on the same interface to use different translation schemes.

All signals in the Untranslated Transactions extension are prefixed with **ARMMU** for read transactions and **AWMMU** for write transactions.

In this specification, **AxMMU** indicates **ARMMU** or **AWMMU**.

E1.9.1 Untranslated Transaction signaling

Table E1-13 shows the signals that support Untranslated Transactions.

Table E1-13 Write address channel signals that support untranslated transactions

Signal	Width	Description
AxMMUSECSID	1	Secure Stream Identifier. • When deasserted indicates a Non-secure stream. • When asserted indicates a Secure stream. This is an optional signal, the default value is 0.
AxMMUSID	SID_WIDTH	Stream Identifier. Secure and Non-secure streams use different name-spaces, qualified with **AxMMUSECSID**, so they can use the same stream identifier values. This is an optional signal, the default value is 0.
AxMMUSSIDV	1	Substream Identifier Valid. Indicates that the transaction has an substream identifier. • When deasserted, this signal indicates that the transaction does not have a substream identifier. • When asserted, this signal indicates that the transaction has a substream identifier. This is an optional signal, the default value is 0.
AxMMUSSID	SSID_WIDTH	Substream Identifier. This signal is only valid if **AxMMUSSIDV** is asserted. For a single stream, the stream with substream 0 is a different stream from the stream with no valid substream. This is an optional signal, the default value is 0.
AxMMUATST	1	Address Translated. Indicates that the transaction has already undergone PCIe ATS translation. This translation might be a full or partial translation in cases where two stages of translation are supported. • When deasserted, this signal indicates that the transaction has not been translated. • When asserted, this signal indicates that the transaction has been translated. This signal is optional for Untranslated Transactions v1, it is not present for Untranslated Transactions v2. If not present the default value is 0. **AxMMUATST** is equivalent to **AxMMUFLOW[0]** when **AxMMUFLOW[1]** is deasserted.

Table E1-13 Write address channel signals that support untranslated transactions (continued)

Signal	Width	Description
AxMMUFLOW	2	Indicates the SMMU flow for managing translation faults for this transaction. Encoded as: `0b00` Stall `0b01` ATST `0b10` No stall `0b11` PRI This signal is optional for Untranslated Transactions v2, it is not present for Untranslated Transactions v1. If not present the default is `0b00`.
RRESP	3	When Untranslated_Transactions is v2, **RRESP** extended to 3-bits to accommodate the signaling of an additional response. The following response is permitted when using untranslated transactions v2: `0b101` TRANSFAULT. Transaction was terminated because of a translation fault which might be resolved by a PRI request. Read data is not valid.
BRESP	3	When Untranslated_Transactions is v2, **BRESP** extended to 3-bits to accommodate the signaling of an additional response.The following response is permitted when using untranslated transactions v2: `0b101` TRANSFAULT. Transaction was terminated because of a translation fault which might be resolved by a PRI request.

E1.9.2 Optional signals and default values

During the building of a system, it is possible that the stream identifiers for a given component have some ID bits provided by the component and some ID bits that are tied off for that component. This fixes the range of values in the stream identifier name space that can be used by that component. Typically, the low-order bits are provided by the component and the high-order bits are tied off.

Any additional identifier field bits for **AxMMUSID** or **AxMMUSSID**, that are not supplied by the component or hard coded by the interconnect, must be tied LOW.

All signals are optional with defined default values, with the restrictions:

- **ARMMUSSID** and **ARMMUSSIDV** must either be both present or both absent.
- **AWMMUSSID** and **AWMMUSSIDV** must either be both present or both absent.

E1.9.3 PCIe considerations

When the Untranslated_Transactions signaling is used for interfacing to PCIe Root Complex, the following considerations apply:

- All PCIe transactions must be Non-secure.
  - **AxMMUSECSID** must either not be present, or must be tied LOW.
- For PCIe transactions:
  - **AxMMUSID** corresponds to the PCIe Requester ID.
  - **AxMMUSSID** corresponds to the PCIe PASID.
  - **AxMMUSSIDV** is asserted if the transaction had a PASID prefix, otherwise it is deasserted.

E1.9.4 Use of Untranslated Transactions with ACE5

It is possible to use address translation on untranslated transactions from an ACE5 master. There are restrictions, however, depending on the type of transaction being translated.

In general, the translation process:

- Must not convert Shareable transactions into Non-shareable transactions, since this can break coherency.
- Must not convert Allocating Shareable transactions into Non-allocating Shareable transactions, since this can mislead a downstream snoop filter.
- Must ensure that when converting write transactions from Non-shareable to Shareable transactions, a WriteUnique or WriteLineUnique transaction is not outstanding at the same time as a WriteBack, WriteClean, or WriteEvict transaction.

For transactions that are IO Coherent or Non-shareable, the following rules apply:

- The translation process is used for protection checking and can also be used for address translation.
- Protection checks can result in any combination of permissions. Both read and write transactions must be checked.
- The master must not permit a snoop to hit a cache line that has been fetched using an IO Coherent or Non-shareable transaction.
- Transactions within this group are:
  - ReadNoSnoop
  - WriteNoSoop
  - ReadOnce
  - ReadOnceMakeInvalid
  - ReadOnceCleanInvalid
  - WriteUnique
  - WriteLineUnique
  - WriteBack to Non-shareable locations
  - WriteClean to Non-shareable locations
  - WriteEvict to Non-shareable locations

For transactions that are Allocating Coherent, the following rules apply:

- The translation process can be used for protection checking, but must always result in either full read and write access or no access. Read-only or write-only permissions are not supported.
- The translation process must result in the same address after translation as before translation.
- Shareable WriteBack, WriteClean, WriteEvict, and Evict transactions are permitted, but do not need to be checked. They can only occur after the successful permission check of a transaction that permits the cache line to be allocated in the cache.
- A transaction that results in an error response must not be allocated in the cache.
- Transactions within this group are:
  - ReadShared
  - ReadClean
  - ReadNotSharedDirty
  - ReadUnique
  - CleanUnique
  - MakeUnique
  - WriteBack to Shareable locations
  - WriteClean to Shareable locations
  - WriteEvict to Shareable locations
  - Evict

E1.9.5 SMMU flows

This section describes the different SMMU flows for managing translation faults.

Stall flow

When the Stall flow is used, software can configure the SMMU to take one of the following actions when a translation fault occurs:

- Terminate the transaction with a SLVERR response.
- Terminate the transaction with an OKAY response, data is RAZ/WI.
- Stall the translation and inform software that the translation is stalled. Software can then instruct the SMMU to terminate the transaction, or update the page tables and retry the translation. The master is not aware of the stall.

This flow enables software to manage translation faults and demand paging without the master being aware. However, it has some limitations:

- The master can see very long transaction latency, potentially triggering timeouts.
- Due to the dependence of software activity, the Stall flow can cause deadlocks in some systems.

  For example, it is not recommended for use with PCIe, because of dependencies between outgoing transactions to PCIe from a CPU, and incoming transactions from PCIe through the SMMU.

Enabling the Stall flow does not necessarily cause a stall when a translation fault occurs. Stalls only occur when enabled by software. Software does not normally enable stalling for PCIe endpoints.

ATST flow

The Address Translation Service Translated (ATST) flow indicates that the transaction has already been translated by ATS. It is only used by PCIe Root Ports.

The ATST flow is indicated when **AxMMUATST** is asserted or **AxMMUFLOW** is `0b01`.

When the flow is ATST, the transaction might still undergo some translation, depending on the configuration of the SMMU. For more information on the SMMUv3 architecture, see *Arm System Memory Management Unit Architecture Specification*. If a translation fault occurs, the transaction must be terminated with a SLVERR response.

When the flow is ATST, the following constraints apply:

- **AxMMUSECSID** must be LOW. Secure translated transactions are not supported.
- **AxMMUSSIDV** must be LOW. Substream IDs for translated transactions are not supported.

NoStall flow

The NoStall flow is used by a master that is not able to be stalled, it is indicated when **AxMMUFLOW** is `0b10`.

If a translation fault occurs when using this flow, the slave must terminate the transaction with a SLVERR or OKAY response, even if software has configured the device to be stalled when a translation fault occurs.

This flow is recommended for masters such as PCIe Root Ports which might deadlock if stalling is enabled by software.

PRI flow

The PRI flow is designed for use with a PCIe integrated endpoint. It is indicated when **AxMMUFLOW** is `0b11`. The master uses the PRI flow to enable software to respond to translation faults without risking deadlock.

When the flow is PRI and a translation fault occurs, the transaction is terminated with a TRANSFAULT response on **BRESP** or **RRESP**. The master can then use a separate mechanism to request that the page is made available, before retrying the transaction. This mechanism is normally PCIe PRI.

When this flow is used, software enables ATS but no ATS features are required in hardware.

A transaction that uses this flow might still be terminated by the SMMU with a SLVERR, if the translation failed for a reason which cannot be resolved by a PRI request, for example because the SMMU is incorrectly configured.

E1.9.6 Protocol rules

The following rules apply to Untranslated Transactions signals.

- For transactions that do not specify a substream ID, as indicated by **AxMMUSSIDV** deasserted. **AxMMUSSID** must be driven to all zeros.
- For transactions that are in a Non-secure stream, as indicated by **AxMMUSECSID** deasserted, **AxPROT[1]** must be HIGH, which indicates a Non-secure transaction.
- A TRANSFAULT is indicated using a value of `0b101` for **RRESP** and **BRESP**.
- If **AWMMUFLOW** is not PRI, **BRESP** must not be TRANSFAULT.
- If **ARMMUFLOW** is not PRI, **RRESP** must not be TRANSFAULT.
- If TRANSFAULT is used for one response beat, it must be used for all response beats of a transaction.

E1.9.7 StashTranslation

The Untranslated Transactions extension also supports a StashTranslation transaction. This indicates that the translation for the given transaction address should be cached, since it is likely to be needed.

The StashTranslation transaction must be supported in any of the following cases::

- Untranslated_Transactions is True and Cache_Stash_Transactions is True
- Untranslated_Transactions is v1
- Untranslated_Transactions is v2

A StashTranslation transaction has no data transfers. The address and control information is provided on the AW channel, and a single response is provided on the B channel. The response must be provided only after the address has been accepted.

The following restrictions apply for the StashTranslation transaction:

- No stash target is supported. If present, **AWSTASHNID[10:0]**, **AWSTASHNIDEN**, **AWSTASHLPID[4:0]**, and **AWSTASHLPIDEN** must be driven LOW.
- Any legal combination of **AxCACHE** and **AxDOMAIN** values is permitted. See *AxCACHE and AxDOMAIN signal combinations* on page D3-177.
- StashTranslation transactions must not use the same AXI ID values that are used by non-StashTranslation transactions that are outstanding at the same time. This rule ensures that there are no ordering constraints between StashTranslation transactions and other transactions, so a slave that does not stash translations can respond immediately.

E1.10 Non-secure access identifiers

To support the storage and processing of protected data, AMBA 5 provides a set of signals that enable access to particular Non-secure memory locations to be controlled. The signals supply a Non-secure Access Identifier (NSAID) alongside the transaction request. The NSAID can be checked to permit or deny access to a memory location.

The NSAccess_Identifiers property is used to indicate whether a component supports these additional signals:

True NSAID signaling is present on the interface.

False NSAID signaling is not present on the interface. If NSAccess_Identifiers is not declared, it is considered False.

The non-secure access identifiers extension is applicable to the following interfaces:

- AXI5
- ACE5
- ACE5-Lite
- ACE5-LiteDVM

E1.10.1 NSAID signaling

Table E1-14 shows the signals that are associated with each channel.

Table E1-14 NSAID signals associated with each channel

Signal	Source	Description
AWNSAID[3:0]	Master	Non-secure Access Identifier for a write transaction.
ARNSAID[3:0]	Master	Non-secure Access Identifier for a read transaction.
CRNSAID[3:0][a]	Master	Non-secure Access Identifier for a snoop response.

a. Implemented in ACE5 only.

If the NSAccess_Identifiers property is True, then **AWNSAID** and **ARNSAID** must both be present on the interface. **CRNSAID** can only be included on ACE5 interfaces and is optional when NSAccess_identifiers is True.

AWNSAID and **ARNSAID** are provided alongside write and read transaction requests, respectively. **CRNSAID** is supplied alongside a snoop response and is used to indicate the NSAID value that was originally used to fetch data that is held in the cache of a coherent master. **CRNSAID** is only used for ACE5 masters that can provide data in response to a snoop transaction.

A 4-bit NSAID value supports up to 16 unique identifiers. For each NSAID there is a set of access permission that is defined which determine how locations in memory are permitted to be accessed.

The access permissions can be:

- No access
- Read-only access
- Write-only access
- Read/write access

The mechanism that is used to define the access permissions for each NSAID is IMPLEMENTATION DEFINED. However, this mechanism is typically implemented using some form of *Memory Protection Unit* (MPU).

It is permitted for transactions with different NSAID values to have access to overlapping locations in memory. It is permitted for transactions with different NSAID values to have any combination of access permissions for a given location in memory.

A default NSAID value of zero is supported. Typically, masters use a default NSAID value of zero when accessing data that is not protected, or when they do not have an assigned NSAID value.

If a master is required to use a single NSAID value, then it is permitted for NSAID signals to be tied to a fixed value.

The NSAID signals are only used for Non-secure transactions.

For Secure transactions, as indicated by **AxPROT[1]** = 0, a value of zero must be used for NSAID.

E1.10.2 Caching and NSAID

Where caching and system coherency is performed upstream of permission checking, accesses with different NSAID values that pass data between them must be subjected to permission checks. The rules that are associated with NSAID use and coherency are as follows:

- When an agent caches a line of data that has been fetched using a particular NSAID value, it must ensure that any subsequent write to main memory or any response to a snoop uses the same NSAID value. This rule ensures that a master cannot move a cache line of data from one protected region to another.
- For a read request with a given NSAID value, if a snoop is used to obtain the data:
  - If the NSAID value of the snoop response matches the read request then data can be provided directly.
  - If the NSAID value of the snoop response does not match the read request, then the cache line must first be written to memory using the NSAID value obtained via the snoop response, and then read from memory using the NSAID value of the original request.

    Note

    The write and subsequent read are only required to reach a point at which permission checking has occurred.
- Snoop transactions that invalidate cached copies, such as MakeInvalid, must not be used if memory protection is used. All such snoop transactions must be replaced with transactions that also clean the cache line to main memory, such as CleanInvalid.
- Any interconnect-generated write to main memory that occurs as the result of a snoop must use the NSAID value that is obtained from the snoop response.
- If a single master can issue transactions with multiple NSAID values, it must ensure that internal accesses to cached copies use the NSAID value that was used to fetch the cache line initially:
  - An access that has a cache line hit with the same address, but a different NSAID value, must clean and invalidate the cache line before refetching the cache line with the appropriate NSAID value. This process ensures that a protection check is performed.
  - If it is guaranteed that the master never accesses the same cache line with a different NSAID value, clean and invalidation operations are not necessary.

    This guarantee can be by design or be assured by using appropriate cache maintenance operations.
- Appropriate cache maintenance must be performed when changing the access permissions for NSAID values.

Note

It is permitted for a master to write to a cache line when that agent does not have write permission to the location. It is also permitted for the updated cache line to be passed to other masters using the same NSAID value. However, it is not permitted for the update to propagate to main memory or to an access using a different NSAID value.

E1.11 Read data chunking

The read data chunking option enables a slave interface to send read data for a transaction in any order using a 128 bit granule. The start address might be used as a hint to determine which chunk to send first, but the slave is permitted to return data in any order.

The property Read_Data_Chunking is used to indicate whether an interface supports the return of read data in reorderable chunks:

True Read data reordering is supported; the interface includes the chunking signals.

False Read data reordering is not supported; the interface does not include the chunking signals and read data must be sent in-order.

If Read_Data_Chunking is not declared, it is considered False.

The Read_Data_Chunking property is supported in the following interfaces:

- AXI5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

To enable read data chunking, signals are added to the read address and data channel.

Table E1-15 Signals that enable read data chunking

Signal	Source	Width	Description
ARCHUNKEN	Master	1	Read data chunking enable. If asserted, read data for this transaction can be returned out of order in 128 bit chunks.
RCHUNKV	Slave	1	Indicates the validity of **RCHUNKNUM** and **RCHUNKSTRB**.
RCHUNKNUM	Slave	8 (128-bit read data) 7 (256-bit read data) 6 (512-bit read data) 5 (1024-bit read data)	Read data chunk number. Indicates the chunk number being transferred. Chunks are numbered incrementally from zero, according to the data width and base address of the transaction. For read data widths of 64 bits or smaller, this signal can be 1 bit wide or omitted.
RCHUNKSTRB	Slave	2 (256-bit read data) 4 (512-bit read data) 8 (1024-bit read data)	Read data chunk strobe. Indicates the read data chunks that are valid for this transfer. Each bit corresponds to 128 bits of data. The least significant bit of **RCHUNKSTRB** corresponds to the least significant 128 bits of **RDATA**. For read data widths of 128 bits or smaller, this signal can be 1 bit wide or omitted.

E1.11.1 Read data chunking protocol rules

In the read data chunking protocol, all the following rules apply:

- **ARCHUNKEN** must only be asserted for transactions with the following attributes:
  - **ARSIZE** is equal to the data bus width or **ARLEN** is one beat.
  - **ARSIZE** is 128 bits or larger.

  - **ARADDR** is aligned to 16 bytes.
  - **ARBURST** is INCR or WRAP.
  - **ARSNOOP** is ReadNoSnoop, ReadOnce, ReadOnceCleanInvalid or ReadOnceMakeInvalid.
- The ID value must be unique-in-flight, which means:
  - **ARCHUNKEN** can only be asserted if there are no outstanding read transactions using the same **ARID** value.
  - The master must not issue a request on the read channel with the same **ARID** as an outstanding request that had **ARCHUNKEN** asserted.
  - If present on the interface, **ARIDUNQ** must be asserted.
- If **ARCHUNKEN** is deasserted, **RCHUNKV** must be deasserted for all response beats of the transaction.
- If **ARCHUNKEN** is asserted, **RCHUNKV** can be asserted for response beats of the transaction.
- **RCHUNKV** must be the same for every response beat of a transaction.
- When **RVALID** and **RCHUNKV** are asserted, **RCHUNKNUM** must be between zero and **ARLEN**.
- When **RVALID** and **RCHUNKV** are asserted, **RCHUNKSTRB** must not be zero.
- When **RVALID** and **RCHUNKV** are asserted, **RLAST** must only be asserted for the final response beat of the transaction, irrespective of **RCHUNKNUM** and **RCHUNKSTRB**.
- When **RVALID** is asserted and **RCHUNKV** is deasserted, **RCHUNKNUM** and **RCHUNKSTRB** can take any value.
- The number of data chunks transferred must be consistent with **ARLEN** and **ARSIZE**, the number of bytes transferred in a burst is the same whether chunking is enabled or not. For unaligned transactions, chunks at addresses lower than **ARADDR** are not transferred.

E1.11.2 Interoperability

If a master supports read data chunking, then downstream interconnect and slaves can reduce their buffering if they also support chunking. An interconnect which connects to components with a mixture of chunking support can drive **ARCHUNKEN** and **RCHUNKV** according to the capabilities of the attached components.

When connecting interfaces with different values for the Read_Data_Chunking property, the following rules apply:

Table E1-16 Connecting interfaces with different values for Read_Data_Chunking

	Slave: False	**Slave: True**
Master: False	**ARCHUNKEN** is not present. **RCHUNKV** is not present. **RCHUNKNUM** is not present. **RCHUNKSTRB** is not present. Full data beats are sent in natural order.	Slave **ARCHUNKEN** input is tied low. Slave **RCHUNKV** output is unconnected. Slave **RCHUNKNUM** output is unconnected. Slave **RCHUNKSTRB** output is unconnected. Full data beats are sent in natural order.
Master: True	Master **ARCHUNKEN** output is unconnected. Master **RCHUNKV** input is tied low. Master **RCHUNKNUM** input is tied. Master **RCHUNKSTRB** input is tied. Full data beats are sent in natural order.	Chunking signals are connected. Read data can be reordered and sent in chunks.

E1.11.3 Chunking examples

In these examples, each row in the figure represents a transfer and the shaded cells indicate bytes that are not transferred.

Figure E1-6 shows a transaction on a 256-bit width read data bus, where:

- **ARADDR** is `0x00`.
- **ARLEN** is 2 beats.
- **ARSIZE** is 256 bits.
- **ARBURST** is INCR.

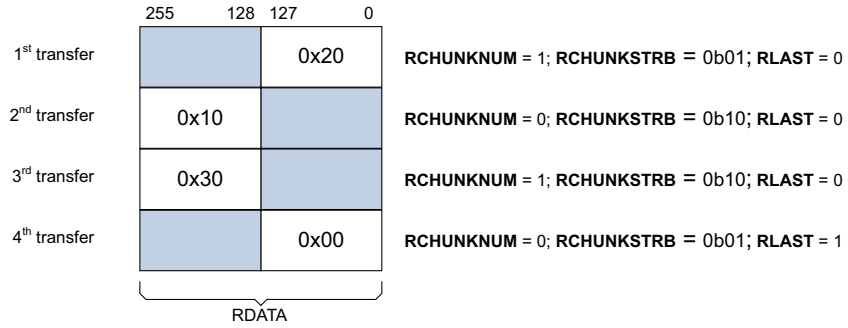


Figure E1-6 Example of read data returned in 128 bit chunks

Figure E1-7 shows transactions on a 256-bit width data read bus, where:

- **ARADDR** is `0x10`.
- **ARLEN** is 2 beats.
- **ARSIZE** is 256 bits.
- **ARBURST** is INCR.

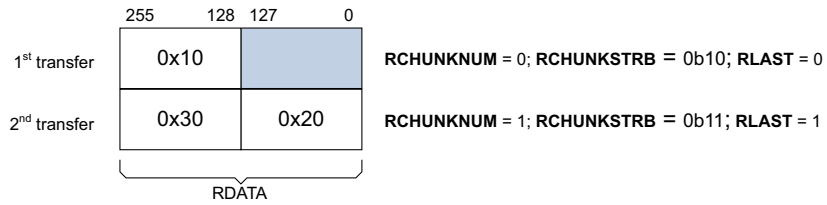


Figure E1-7 Example with an unaligned address and a mixture of 128-bit and 256-bit chunks

Figure E1-8 shows transactions on a 128-bit width data read bus, where:

- **ARADDR** is `0x10`.
- **ARLEN** is 4 beats.
- **ARSIZE** is 128 bits.
- **ARBURST** is WRAP.
- **RCHUNKSTRB** is not present.

The slave uses the start address as a hint and sends the chunk at `0x10` first.

RCHUNKNUM numbering is not dependent on whether the burst is INCR or WRAP.

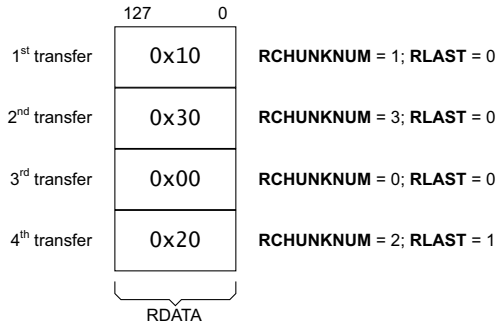


Figure E1-8 Example of a wrapping burst

E1.12 Read interleaving property

Read data from transactions with different **ARID** values can be interleaved within the AXI ordering model.

Some AXI master and interconnect components can be more efficiently designed if it is determined at design-time whether the attached slave interface will interleave read data from different transactions.

The property Read_Interleaving_Disabled is used to indicate whether an interface supports the interleaving of read data beats from different transactions.

For master interfaces, Read_Interleaving_Disabled indicates:

True The master is not capable of receiving read data that is interleaved. The connected slave must have the property as True.

False The master can receive read data that is interleaved. This is legacy AXI-compliant behavior.

For slave interfaces, Read_Interleaving_Disabled indicates:

True The slave is guaranteed not to interleave read data.

False The slave might interleave data from read transactions with different ARID values. The connected master must have the property as False.

If the Read_Interleaving_Disabled property is not declared, it is considered to be False.

For some interfaces, this property can be used as a configuration control, for others it is a capability indicator. All masters that issue bursts with different IDs must be designed to accept interleaved data. Masters might use a configuration option to disable interleaving as an optimization, when the attached slave supports the disabling of interleaving.

The Read_Interleaving_Disabled property is supported in the following interfaces:

- AXI5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

E1.13 Unique ID indicator

The unique ID indicator is an optional flag that indicates when a request on the read and write address channels is using an AXI identifier that is unique for in-flight transactions. A corresponding signal is also on the read and write response channels to indicate that a transaction is using a unique ID.

The unique ID indicator can be used downstream of the AXI master to determine when a transaction needs to be ordered with respect to other transactions from that master. Transactions that do not require ordering might not require tracking in downstream components. Responses with the indicator set are not required to look up in the trackers.

The Unique_ID_Support property is used to indicate if an interface supports the Unique ID Indicator:

True The interface has the unique ID indicator on read and write address and response channels.

False The interface does not have unique ID indicator signals.

If the Unique_ID_Support property is not declared, it is considered to be False.

The Unique ID Indicator is supported in the following interfaces:

- AXI5
- AXI5-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

Table E1-17 shows the signals that are added to provide unique ID indication.

Table E1-17 Signals for unique ID indication

Signal	Source	Description
ARIDUNQ	Master	Read address channel unique ID indicator, active HIGH.
RIDUNQ	Slave	Read data channel unique ID indicator, active HIGH.
AWIDUNQ	Master	Write address channel unique ID indicator, active HIGH.
BIDUNQ	Slave	Write response channel unique ID indicator, active HIGH.

The following rules apply to the unique ID indicator:

- When **ARIDUNQ** is asserted, there must be no outstanding read transactions from this master with the same **ARID** value.
- A master must not issue a read request with the same **ARID** as an outstanding read transaction that had **ARIDUNQ** asserted.
- If **ARIDUNQ** is deasserted for a request, the corresponding **RIDUNQ** signals must be deasserted for all response beats for that transaction.
- If **ARIDUNQ** is asserted for a request, the corresponding **RIDUNQ** signals must be asserted for all response beats for that transaction.
- When **AWIDUNQ** is asserted, there must be no outstanding write transactions from this master with the same **AWID** value.
- A master must not issue a write request with the same **AWID** as an outstanding write transaction that had **AWIDUNQ** asserted.
- If **AWIDUNQ** is deasserted for a request, the corresponding **BIDUNQ** signal must be deasserted for all response beats for that transaction.

- If **AWIDUNQ** is asserted for a request, the corresponding **BIDUNQ** signal must be asserted for all response beats for that transaction.

A transaction is outstanding from the cycle that had **AxVALID** asserted until the cycle when the final response transfer is accepted by the master. If an interface includes **BCOMP**, the transaction is considered to be outstanding until a response is received with **BCOMP** asserted.

An Atomic transaction is outstanding until both write and read responses are accepted by the master.

Some transaction types specify that **AxIDUNQ** is required to be asserted, if present. If not specified, asserting **AxIDUNQ** is optional, even if there are no outstanding transactions using the same **AxID**.

E1.14 Memory Partitioning and Monitoring (MPAM)

MPAM is a technology for partitioning and monitoring memory system resources for physical and virtual machines. The full MPAM architecture is described in the Armv8.4 extensions.

Each MPAM-enabled master adds MPAM information to its read and write requests. The MPAM information is propagated through the system to memory components, where it can be used to influence resource allocation decisions. Monitoring memory usage based on MPAM information can also enable the tuning of performance and accurate costing between machines.

The MPAM_Support property is used to indicate whether an interface supports MPAM:

MPAM_9_1 The interface is enabled for partitioning and monitoring, it must include the MPAM signal on all address channels. The width of PARTID is 9 and PMG is 1.

False The interface is not MPAM-enabled. No MPAM signals are present on the interface.

If the MPAM_Support property is not declared, it is considered to be False.

The MPAM extension is supported in the following interfaces:

- AXI5
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

E1.14.1 MPAM signaling

When MPAM_Support is MPAM_9_1, the MPAM signaling is shown in Table E1-18.

Table E1-18 MPAM signaling

Signal	Source	Description
ARMPAM[10:0]	Master	Read address channel MPAM information
AWMPAM[10:0]	Master	Write address channel MPAM information

The MPAM information has three fields. They are mapped on the MPAM signals is shown in Table E1-19.

Table E1-19 MPAM fields

Field	Description	Width	Default	Mapping
MPAM_NS	Security indicator	1	**AxPROT[1]**	**AxMPAM**[0]
PARTID	Partition identifier	9	`0x000`	**AxMPAM**[9:1]
PMG	Performance monitor group	1	`0b0`	**AxMPAM**[10]

For details on how to use MPAM information, refer to the MPAM architecture document.

For DVM operations, MPAM values do not apply and the **ARMPAM** signal can take any value.

E1.14.2 MPAM component interactions

Implementation of MPAM technology has impacts on:

- Master components
- Interconnect components
- Slave components

Master components

Master components that are MPAM-enabled must drive MPAM signals when the corresponding **AxVALID** is asserted. Values used are IMPLEMENTATION DEFINED for all transaction types. It is expected, but not required, that a master use the same sets of values for read and write requests. A master might not use all the PARTID or PMG values that can be signaled on the AXI interface.

If a master component is included in an MPAM-enabled system, but does not support MPAM signaling, then the system must add the MPAM information. Default values are shown in Table E1-19 on page E1-385. If a master is required to drive MPAM_NS differently to **AxPROT**[1], it must include **AxMPAM** signals.

Interconnect components

MPAM identifiers have global scope. There is no requirement for interconnect components to remaster or uniquify MPAM identifiers. When an interconnect master interface is connected to an MPAM-enabled slave, it can use propagated values or IMPLEMENTATION DEFINED values.

Slave components

A slave component that is MPAM-enabled can use the MPAM information for memory partitioning and monitoring. MPAM signals are sampled when the corresponding **AxVALID** is asserted.

If an attached master does not support MPAM, the system must supply a value for the MPAM information required for the interface. Values used are IMPLEMENTATION DEFINED.

E1.15 Memory tagging

The Memory Tagging Extension (MTE) provides a mechanism that can be used to detect memory safety violations.

When a region of memory is allocated for a particular use, it is given an Allocation Tag value. When the memory is subsequently accessed, a Physical Tag value is provided that corresponds to the physical address of the access. If the Physical Tag does not match the Allocation Tag, a warning is generated.

Allocation Tags are stored in the memory system and can be cached in the same way as data. Each tag is 4 bits and is associated with a 16-byte aligned address location.

The following operations are supported:

- Updating the Allocation Tag value using a write transaction, with or without updating the associated data value.
- Reading of data with associated Allocation tag. The Requestor can then perform the check of Physical Tag against the Allocation Tag.
- Writing to memory with a Physical Tag to be compared with the Allocation Tag. The result is indicated in the transaction response.

When memory tagging is supported in a system, it is not required that every transaction uses memory tagging. It is also not required that every component in the system supports memory tagging.

The Memory Tagging Extension architecture is described as part of the *Architecture Reference Manual v8.5*.

E1.15.1 MTE support

The MTE_Support property is used to indicate that a component supports the Memory Tagging Extension.

Standard Memory tagging is supported on the interface, all MTE signals are present.

Basic Memory tagging is supported on the interface at a basic level. A limited set of tag operations are permitted. **BTAGMATCH** is not present. **BCOMP** is not required.

False Memory tagging is not supported on the interface and no MTE signals are present. If MTE_Support is not declared, it is considered False.

The MTE_Support property can be set for the following interface types:

- ACE5-Lite
- ACE5-LiteDVM (Basic only)
- AXI5

MTE_Support must be False on interfaces with a data width smaller than 32 bits.

Table E1-20 shows compatibility between master and slave interfaces, according to the values of the MTE_Support property.

Table E1-20 MTE_Support

	Slave: False	**Slave: Basic**	**Slave: Standard**
Master: False	Compatible.	Compatible.	Compatible.
Master: Basic	Protocol compliant. The slave ignores **AxTAGOP**, so write tag values are lost and read tag values are static.	Compatible.	Compatible.
Master: Standard	Not compatible. If the master uses the Match operation, the response will not be protocol-compliant.	Not compatible. If the master uses the Match operation, the response will not be protocol-compliant.	Compatible.

E1.15.2 MTE signaling

Table E1-21 shows the signals that are required to support MTE.

Table E1-21 MTE Signaling

Signal	**Width**	**Description**
ARTAGOP	2	Read request tag operation, encoded as: `0b00` Invalid `0b01` Transfer `0b10` Reserved `0b11` Fetch
RTAG	ceil(DATA_WIDTH/128)*4	The tag that is associated with read data. There is a 4-bit tag per 128 bits of data, with a minimum of 4 bits. **RTAG[4n-1:4(n-1)]** corresponds to **RDATA[128n-1:128(n-1)]**. **RTAG** has the same validity rules as **RDATA**.
AWTAGOP	2	Write request tag operation, encoded as: `0b00` Invalid `0b01` Transfer `0b10` Update `0b11` Match
WTAG	ceil(DATA_WIDTH/128)*4	The tag that is associated with write data. There is a 4-bit tag per 128 bits of data, with a minimum of 4 bits. **WTAG[4n-1:4(n-1)]** corresponds to **WDATA[128n-1:128(n-1)]**. **WTAG** has the same validity rules as **WDATA**.

Table E1-21 MTE Signaling (continued)

Signal	Width	Description
WTAGUPDATE	ceil(DATA_WIDTH/128)	Indicates which tags must be written to memory in an Update operation. • If a bit is asserted, then the corresponding tags must be written to memory. • If a bit is deasserted, then the corresponding tags are invalid. There is 1 bit per 4 bits of tag. **WTAGUPDATE[n]** corresponds to **WTAG[4n+3:4n]**. **WTAGUPDATE** bits outside of the transaction container must be deasserted. For operations other than Update, **WTAGUPDATE** must be deasserted. It can be asserted or deasserted for Update operations.
BTAGMATCH	2	Indicates the result of a tag comparison on a write transaction: **0b00** Not a match transaction **0b01** Match result in separate response **0b10** Fail **0b11** Pass
BCOMP	1	Indicates that the write is observable. This signal is also required for persistent CMOs on the write channels.

E1.15.3 Caching tags

Allocation Tags that are cached must be kept hardware-coherent. The coherence mechanism is the same as data coherence.

Applicable tag cached states are: Invalid, Clean, and Dirty. A line that is either Clean or Dirty is Valid.

Constraints on the combination of data cache state and tag cache state are:

- Tags can be Valid only when data is Valid.
- Tags can be Invalid when data is Valid.
- When a cached line is evicted and tags are Dirty, then it is permitted to treat clean data that is evicted as dirty.
- When Dirty tags are evicted from a cache, they must be either written back to memory or passed dirty to another cache.
- When Clean tags are evicted from a cache, they can be sent to other caches or dropped silently.
- A CMO which hits a line with Valid tags applies to the data and the tag.
- When a MakeInvalid or ROMI transaction hits a line with dirty tags, the tags must be written back to memory.

E1.15.4 Transporting tags

Tag values are transported using the **RTAG** signal when **ARTAGOP** is not Invalid.

Tag values are transported using the **WTAG** signal when **AWTAGOP** is not Invalid.

When transporting tags, the following rules apply in addition to other constraints based on the transaction type:

- The transaction must be cache-line-sized or smaller.
- **AxBURST** must be INCR or WRAP, not FIXED.

- The transaction must be to Normal Write-Back memory, which means:
  - **AxCACHE[3:2]** is not `0b00`
  - **AxCACHE[1:0]** is `0b11`
- The ID value must be unique-in-flight, which means:
  - A read with tag Transfer or Fetch can only be issued if there are no outstanding read transactions using the same **ARID** value.
  - A master must not issue a request on the read channel with the same **ARID** as an outstanding read with tag Transfer or Fetch.
  - If present, **ARIDUNQ** must be asserted for a read with tag Transfer or Fetch.
  - A write with tag Transfer, Update or Match can only be issued if there are no outstanding write transactions using the same **AWID** value.
  - A master must not issue a request on the write channel with the same **AWID** as an outstanding write with tag Transfer, Update, or Match.
  - If present, **AWIDUNQ** must be asserted for a write with tag operations Transfer, Update, or Match.
- For data widths wider than 128 bits, the tag signal carries multiple tags. The tags are driven appropriate to the data being transported, with the least significant tag bits used to transport the tag for the least significant 128 bits of data.
- For read transactions that use read data chunking, only tags which correspond to valid chunk strobes are required to be valid.
- For write transactions where multiple beats address the same tag, **WTAG** and **WTAGUPDATE** values must be consistent for each 4-bit tag that is accessed by the transaction.

E1.15.5 Reads with tags

A read can request that Allocation Tags are returned along with data, which is determined by the value of **ARTAGOP**. Table E1-22 shows reads with tags operations.

Table E1-22 Reads with tags operations

Tag Operation	ARTAGOP	Description
Invalid	`0b00`	Tags are not required to be returned with the data. In the response to this request, **RTAG** is invalid and must be zero.
Transfer	`0b01`	Each beat of read data must have a valid tag value. Tags must be sent for every 16-byte granule that is accessed, even if the address is not aligned to 16 bytes.
-	`0b10`	Reserved.
Fetch	`0b11`	Only tags are required to be fetched. Data is not required to be valid and must not be used by the master. Transactions using Fetch must be cache line sized. Tags must be sent for every 16-byte granule that is accessed.

Table E1-23 on page E1-391 indicates tag operations that are legal for read transactions.

Transactions that can only be signaled on ACE interfaces are not included in the table because MTE_Support cannot be enabled for ACE interfaces.

Table E1-23 Legal tag operations for read transactions

	MTE_Support = Basic			MTE_Support = Standard		
Transaction	**Invalid**	**Transfer**	**Fetch**	**Invalid**	**Transfer**	**Fetch**
ReadNoSnoop	Y	Y	-	Y	Y	Y
ReadOnce	Y	Y	-	Y	Y	-
ReadOnceCleanInvalid	Y	-	-	Y	-	-
ReadOnceMakeInvalid	Y	-	-	Y	-	-
CleanInvalid / MakeInvalid	Y	-	-	Y	-	-
CleanShared / CleanSharedPersist	Y	-	-	Y	-	-
DVM Message / Complete	Y	-	-	Y	-	-

E1.15.6 Writes with tags

Table E1-24 shows the write tag operations that are defined.

Table E1-24 Writes with tag operations

Tag Operation	AWTAGOP	Description
Invalid	0b00	The tags in the write are not valid; no tag updating or checking is required. **WTAGUPDATE** must be deasserted. **WTAG** must be zero.
Transfer	0b01	The tags are Clean. Tag check does not need to be performed. The completer of the write can cache the tags if it is allocating the data in its cache. **WTAGUPDATE** must be deasserted. **WTAG** bits must be valid for every byte in the transaction container.
Update	0b10	Tag values have been updated and are dirty; the tags in memory must be updated, according to **WTAGUPDATE**. **WTAGUPDATE** can have any number of bits asserted, including none. Tags that are only partially addressed in the transaction must have **WTAGUPDATE** deasserted. WriteUniqueFull and WriteFullCMO with Update must have all associated **WTAGUPDATE** bits asserted. **WTAG** must be valid for every associated **WTAGUPDATE** bit that is asserted.
Match	0b11	The tags in the write must be checked against the Allocation Tag values that are obtained from memory. The Match operation must be performed for all tags where any corresponding write data strobes are asserted. It is required to update memory with the data, even if the match fails. **WTAGUPDATE** must be deasserted. **WTAG** bits must be valid for byte lanes that are enabled by **WSTRB**. For interfaces with more than 4 bits of tags, the Match operation is performed only on those tags that correspond to active byte lanes.

Table E1-25 indicates which tag operations are legal for which write transactions. The asterisk (*) indicates all variants of the transaction.

Table E1-25 Legal tag operations for write transactions

	MTE_Support = Basic				MTE_Support = Standard			
Transaction	**Invalid**	**Transfer**	**Update**	**Match**	**Invalid**	**Transfer**	**Update**	**Match**
WriteNoSnoop	Y	-	Y	-	Y	Y	Y	Y
WriteUnique*	Y	-	Y	-	Y	-	Y	-
Atomic transaction	Y	-	-	-	Y	-	-	Y
CMO	Y	-	-	-	Y	-	-	-
Write*CMO	Y	-	-	-	Y	Y	Y	-

Table E1-25 Legal tag operations for write transactions (continued)

Transaction	MTE_Support = Basic				MTE_Support = Standard			
	Invalid	Transfer	Update	Match	Invalid	Transfer	Update	Match
WriteZero	Y	-	-	-	Y	-	-	-
WriteUniqueStash*	Y	-	-	-	Y	-	-	-
StashOnce*	Y	-	-	-	Y	-	-	-
StashTranslation	Y	-	-	-	Y	-	-	-
Prefetch	Y	-	-	-	Y	Y	-	-

Write transactions with a tag Match operation (**AWTAGOP** is `0b11`) have two parts to the response:

- A Completion response, which indicates that the write is observable.
- A Match response, which indicates whether the tag comparison passes or fails.

A two-part response enables components with separate data and tag storage parts to respond independently. The two parts can be optionally combined into a single response beat.

Response beats can be sent in any order.

Completion response

The Completion response indicates that the write is observable. It has the following rules:

- **BCOMP** must be asserted.
- **BTAGMATCH** must be `0b01` (Match result in separate response).
- **BID** must have the same value as **AWID**.
- If loopback signaling is supported, **BLOOP** must have the same value as **AWLOOP**.
- **BRESP** can be OKAY, EXOKAY, SLVERR, or DECERR.
- The Completion response must follow normal response ordering rules.
- The ID value can be reused when this response is received.

Match response

The Match response indicates the result of the tag comparison on a write.

- If the tags match for every beat of the entire transaction, then the response is Pass.
- If any bits of any tags in the transaction do not match those already stored, then the response is Fail.

A Match response has the following rules:

- **BCOMP** must be deasserted.
- **BTAGMATCH** must be `0b11` (Pass) or `0b10` (Fail).
- **BID** must have the same value as **AWID**.
- **BLOOP** can take any value, it is not required to have the same value as **AWLOOP**.
- **BRESP** can be OKAY, SLVERR, or DECERR.
- The Match response has no ordering requirements, it can overtake or be overtaken by any other response beats.

Combined response

A slave can optionally combine the two responses into a single beat. The following rules apply:

- **BCOMP** must be asserted.
- **BTAGMATCH** must be `0b11` (Pass) or `0b10` (Fail).
- **BID** must have the same value as **AWID**.
- If loopback signaling is supported, **BLOOP** must have the same value as **AWLOOP**.
- **BRESP** can be OKAY, EXOKAY, SLVERR, or DECERR.
- The combined response must follow normal response ordering rules.
- The ID value can be reused when this response is received.

Table E1-26 summarizes possible responses to a Match operation:

Table E1-26 Possible responses to a Match operation

BTAGMATCH	BCOMP	Description
`0b00`	0	Not legal
`0b00`	1	Not legal
`0b01`	0	Not legal
`0b01`	1	Completion response, part of a two-part response
`0b10`	0	Match Fail, part of a two-part response
`0b10`	1	Match Fail or MTE Match not supported, one-part response
`0b11`	0	Match Pass, part of a two-part response
`0b11`	1	Match Pass, one-part response

E1.15.7 MTE and Atomic transactions

An Atomic transaction to a location that is protected with memory tagging can use a write Match operation. Atomic transactions cannot be used with Transfer or Update operations.

For AtomicCompare transactions of 32 bytes, the same tag value must be used for tag bits associated with the compare and swap bytes.

AtomicCompare transactions with Match can be 16 bytes or 32 bytes. If the transaction is 32 bytes, the same tag value must be used for tag bits associated with the compare and swap bytes.

Read data that is returned within an Atomic Transaction does not have valid **RTAG** values, so **RTAG** is recommended to be zero.

E1.15.8 MTE and Prefetch transactions

A Prefetch transaction with **AWTAGOP** of Transfer indicates that the data should be prefetched with tags if possible. A Prefetch transaction has no write data, so no tag Transfer operation occurs within the transaction.

E1.15.9 MTE and Poison

There is no poison signaling directly associated with Allocation Tags. When writing a tag with poisoned data, the stored tag might be marked as poisoned. The exact mechanism for this is IMPLEMENTATION DEFINED.

E1.15.10 Memory tagging interoperability

When an MTE operation is performed to a memory location that does not support memory tagging, the resultant data must be the same as if a non-MTE operation was performed to that location.

- For a read with Transfer or Fetch, **RTAG** is recommended to be zero.
- For a write with Transfer or Update, the data must be written normally. The tag is discarded.
- For a write with Match, the data must be written normally and a single Combined response is given. **BTAGMATCH** must be `0b10` (Fail).

A slave is expected to give an OKAY response to an MTE operation, unless it would have given a different response to an equivalent non-MTE operation.

E1.16 Prefetch request and response

When a master has indication that it might need data for an address but does not want to commit to reading it yet, it can send a request to the system that it might be advantageous to prepare the location for reading. This request to the system can cause reading the data into a downstream cache or from off-chip memory before the master makes the actual read request.

When, for example, a CPU within a shareable domain issues a Prefetch request, it is routed directly to the memory controller, bypassing any coherency checks. The CPU accompanies this with a coherent request to the same address which will be routed through the coherency logic. If the snoop lookup misses, then the request will be propagated to memory, where it might hit on the prefetched data.

The PREFETCHED response indicates that a read transaction has hit upon prefetched data. The master can use this as part of a heuristic to determine if it will continue issuing Prefetch requests.

E1.16.1 Prefetch transaction

A Prefetch is a data-less transaction. The transaction indicates that a master might read data from a specified location at a later time.

The Prefetch_Transaction property is used to indicate whether a component supports prefetching.

True The Prefetch transaction is supported. The **RRESP** signal is extended to 3 bits.

False The Prefetch transaction is not supported.

If Prefetch_Transaction is not declared, it is considered False.

The Prefetch_Transaction property can be set True for the following interface types:

- ACE5-Lite
- ACE5-LiteDVM

The rules for Prefetch transactions are:

- The Prefetch transaction consists of a request on the AW channel and a single response beat on the B channel, there is no data.
- The Prefetch request indicates that the request is a prefetch and the slave can choose to move data from a long-latency store, such as DRAM, into a low-latency store, such as an on-chip RAM or buffer.
- A Prefetch request is signaled using the **AWSNOOP** opcode of `0b1111`.
- A Prefetch request must be cache line sized with the following constraints:
  - The transaction is Regular, see *Regular transactions* on page A3-53.
  - **AWCACHE[1]** is asserted, that is a Normal transaction.
  - **AWDOMAIN** is Non-shareable, Inner Sharable or Outer Shareable.
  - **AWLOCK** is Normal access.
- The ID value must be unique-in-flight, which means:
  - A Prefetch transaction can only be issued if there are no outstanding write transactions using the same **AWID**.
  - The master must not issue a request on the write channel with the same **AWID** as an outstanding Prefetch request.
  - If present on the interface, **AWIDUNQ** must be asserted for Prefetch transactions.
- The master may or may not follow a Prefetch request with a non-Prefetch request to the same address.
- A slave interface at any level can chose to propagate or respond to a prefetch request.
- It is permitted to respond to a Prefetch request with OKAY, DECERR or SLVERR. An OKAY response can be sent irrespective of whether the slave acts on the Prefetch request.

E1.16.2 Response for prefetched data

If a read request hits on data which has been prepared due to a previous Prefetch request, the slave may return a PREFETCHED response. This can be used by the master to determine the success rate of its Prefetch requests.

When an interface supports prefetching, **RRESP** is extended to 3-bits to enable the signaling of a PREFETCHED response. The PREFETCHED response uses the **RRESP** opcode of `0b100`; it has the following rules and recommendations:

- PREFETCHED indicates that read data is valid and has come from a prefetched source.
- PREFETCHED can be used for a response to the following transaction types:
  - ReadNoSnoop
  - ReadOnce
  - ReadOnceCleanInvalid
  - ReadOnceMakeInvalid
- A PREFETCHED response cannot be sent for an exclusive read.
- This specification recommends that PREFETCHED is signaled for all beats or no beats of a response.
- A PREFETCHED response can only be sent if the Prefetch_Transaction property is True for the interface.
- A PREFETCHED response can be sent to a master even if the master has not sent a Prefetch request to that location. For example, if a master happens to read data which was prefetched by another master.

E1.17 Write zero with no data

Many writes in a system, particularly from a CPU have data set to zero. For example while initializing or allocating memory. These writes with a zero value consume write data bandwidth and interconnect power that can be saved by using a data-less request.

The WriteZero transaction is used to zero a cache-line-sized data location. The transaction consists of a write request and write response but has no associated write data transfer. It is functionally equivalent to a regular write to the same location with fully populated data lanes where all data has a value of zero.

E1.17.1 WriteZero transaction configuration

The WriteZero_Transaction property is used to indicate whether an interface supports the WriteZero transaction.

True The WriteZero transaction is supported.

False The WriteZero transaction is not supported.

If WriteZero_Transaction is not declared, it is considered False.

The WriteZero_Transaction property can be set True for the following interface types:

- ACE5-Lite
- ACE5-LiteDVM

E1.17.2 WriteZero transaction

The rules for a WriteZero transaction are:

- A WriteZero request indicates that the data at the locations indicated by address, size and length attributes must be set to zero.
- A WriteZero transaction consists of a request on the AW channel and a single response on the B channel.
- A WriteZero transaction is cache line sized and Regular, see *Regular transactions* on page A3-53.
- **AWSNOOP** must be `0b0111`.
- **AWLOCK** must be normal access.
- **AWTAGOP** must be Invalid.
- **AWID** must be unique-in-flight, which means:
  - A WriteZero transaction can only be issued if there are no outstanding write transactions using the same **AWID** value.
  - A master must not issue a request on the write channel with the same **AWID** as an outstanding WriteZero transaction.
  - If present, **AWIDUNQ** must be asserted for a WriteZero transaction.
- **AWDOMAIN** can take any value. If **AWDOMAIN** is InnerShareable or Outer Shareable, a WriteZero acts as a WriteUniqueFull with zeros as data.

E1.17.3 Interoperability

A master that issues WriteZero requests cannot be connected to a slave that does not support WriteZero.

When the slave does not support WriteZero, the master must be configured to issue write transactions that include write data beats of zero value, rather than WriteZero requests.

E1.18 Additional interface properties

There are many different implementations of components that use interfaces from the AXI family of specifications. Additional interface properties are defined that can help define interface capability or act as means for configuring interfaces.

E1.18.1 Exclusive accesses

Exclusive accesses are used for semaphore type accesses and tend to be used only by a limited subset of masters and slave components.

The Exclusive_Accesses property is used to define whether a master issues exclusive accesses or whether a slave supports them:

True Exclusive accesses are supported. **ARLOCK** and **AWLOCK** are present on the interface.

False Exclusive accesses are not supported. **ARLOCK** and **AWLOCK** are not present on the interface.

If Exclusive_Accesses is not declared, it is considered:

False For AXI4-Lite, AXI5-Lite, and ACE5-LiteACP interfaces

True For other interface types

The Exclusive_Accesses property can be True for the following interfaces:

- AXI3
- AXI4
- AXI5
- ACE
- ACE-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM

Table E1-27 provides guidance that applies when connecting master and slave components with different property values:

Table E1-27 Exclusive_Accesses interoperability

	Slave: False	**Slave: True**
Master: False	Compatible.	Compatible. **ARLOCK** and **AWLOCK** are tied LOW.
Master: True	Not-compatible. Exclusive Accesses will continually fail, but the interface will not deadlock.	Compatible.

E1.18.2 Shareable transactions

Inner Shareable and Outer Shareable transactions are used when accessing locations that might be stored in the caches of masters which are not downstream of the transaction issuer. Some interfaces are not required to identify Shareable transactions, for example if they are downstream of all caches.

The Shareable_Transactions property is used to define whether a master issues Inner Shareable or Outer Shareable transactions and whether a slave supports them:

True Shareable and Non-shareable transactions are supported. **ARDOMAIN** and **AWDOMAIN** are present on the interface.

False Inner Shareable and Outer Shareable transactions are not supported. **ARDOMAIN** and **AWDOMAIN** are not present on the interface.

If Shareable_Transactions is not declared, it is considered:

False For AXI3, AXI4, AXI4-Lite, AXI5, and AXI5-Lite interfaces

True For ACE, ACE-Lite, ACE5, ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACP interfaces

The Shareable_Transactions property can be True for the following interfaces:

- ACE
- ACE-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

Table E1-28 gives guidance that applies when connecting master and slave components with different property values:

Table E1-28 Shareable_Transactions interoperability

	Slave: False	**Slave: True**
Master: False	Compatible.	Compatible. **AxDOMAIN** inputs are tied to {~**AxCACHE[1]**, ~**AxCACHE[1]**}.
Master: True	Compatible. **AxDOMAIN** outputs are unconnected.	Compatible.

E1.18.3 Maximum transaction size and boundary

The maximum size of a transaction is 4KB and transactions are not permitted to cross a 4KB boundary. However, many masters generate transactions which are guaranteed to be smaller than this.

A slave or interconnect might benefit from this information. For example, a slave might be able to optimize away some decode logic. An interconnect striping at a granule smaller than 4KB might be able to avoid burst splitting if it knows that transactions will not cross the stripe boundary.

The property Max_Transaction_Bytes defines the maximum size of a transaction in bytes. It can take the following values:

- 64
- 128
- 256
- 512
- 1024
- 2048
- 4096

If Max_Transaction_Bytes is not declared, it is considered to be 4096.

The Max_Transaction_Bytes property can be declared for a master and slave interface:

- A master is guaranteed to not issue transactions larger than Max_Transaction_Bytes and transactions do not cross a Max_Transaction_Bytes boundary.
- A slave is guaranteed to accept transactions up to Max_Transaction_Bytes in size.

The Max_Transaction_Bytes property is supported in all interface types.

Table E1-29 gives guidance that applies for connecting master and slave components with different property values:

Table E1-29 Max_Transaction_Bytes interoperability

Master < Slave	Master = Slave	Master > Slave
Compatible	Compatible	Not compatible

E1.18.4 Consistent DECERR response

Read transactions have multiple beats of data. The value of **RRESP** is not constrained within a transaction, so each beat can take any legal value. A response of DECERR is generally used when there is a problem accessing a slave, and in this case DECERR is signaled consistently in every beat of read data. There may be a benefit if a master can inspect just one beat of read data to determine whether a DECERR has occurred.

The Consistent_DECERR property is used to define whether a slave signals DECERR consistently within a transaction:

True DECERR is signaled for every beat of read data, or no beats of read data within each cache line of data. For example, a transaction which crosses a cache line boundary can receive a DECERR response for every read data beat on one cache line and no data beats on the next cache line.

False DECERR may be signaled on any number of read data beats.

If Consistent_DECERR is not declared, it is considered to be:

True For AXI4-Lite and AXI5-Lite interfaces, since they do not support multiple beats of data.

False For all other interfaces.

The Consistent_DECERR property can be True for the following interfaces:

- AXI3
- AXI4
- ACE
- ACE-Lite
- ACE5
- ACE5- LiteDVM
- ACE5- Lite
- ACE5- LiteACP
- AXI5

A slave interface that does not use the DECERR response can set the Consistent_DECERR property to True.

Setting this property to True can be helpful when bridging between AXI and CHI, where DECERR translates to a non-data error.

A master with Consistent_DECERR set True can inspect a single beat of data to determine whether a DECERR has occurred.

Table E1-30 shows interoperability guidance when connecting master and slave components with different property values:

Table E1-30 Consistent_DECERR interoperability

	Slave: False	**Slave: True**
Master: False	Compatible.	Compatible.
Master: True	Not compatible. A DECERR response might be missed by the master.	Compatible.

E1.19 Signal width properties

Properties are added to define the width of signals where the width is not fixed by the specification.

If a property is zero, then the associated signals are not present on the interface.

The maximum values are for guidance only, in order to set a reasonable maximum for configurable interfaces.

Table E1-31 Signal width properties

Name	Min	Max	Applies to	Constraints
ADDR_WIDTH	1	64	**AWADDR**, **ARADDR**, **ACADDR**	
DATA_WIDTH	8	1024	**WDATA**, **RDATA**, **CDDATA**	• 32 or 64 for AXI4-Lite. • 128 for ACE5-LiteACP. • 8, 16, 32, 64, 128, 256, 512, 1024 for other interfaces.
ID_R_WIDTH	0	32	**ARID**, **RID**	
ID_W_WIDTH	0	32	**AWID**, **BID**	
LOOP_R_WIDTH	0	8	**ARLOOP**, **RLOOP**	
LOOP_W_WIDTH	0	8	**AWLOOP**, **BLOOP**	
SID_WIDTH	0	32	**AWMMUSID**, **ARMMUSID**	
SSID_WIDTH	0	20	**AWMMUSSID**, **ARMMUSSID**	
USER_REQ_WIDTH	0	128	**AWUSER**, ARUSER	
USER_DATA_WIDTH	0	DATA_WIDTH/2	**WUSER**, **RUSER**	
USER_RESP_WIDTH	0	16	**BUSER**, **RUSER**	

Chapter E2
Interface and data protection

This chapter specifies schemes for the protection of data and interfaces using poison and parity signaling:

E2.1 Poison

Poison signaling is used to indicate that a set of data bytes have been previously corrupted. Passing the Poison signaling alongside the data permits any future user of the data to be notified that the data might be corrupt. Poison signaling is supported at the granularity of 1 bit for every 64 bits of data.

The Poison property is used to indicate whether a component supports Poison signaling:

True Poison signaling is supported.

False Poison signaling is not supported. If the Poison property is not declared, it is considered to be False.

The Poison extension is supported in the following interfaces:

- AXI5
- AXI5-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

When the Poison signal is asserted, it indicates that the associated 64-bit data granule is corrupt.

Table E2-1 shows the Poison signaling.

Table E2-1 Poison signaling

Signal	Source	Width	Description
RPOISON	Slave	ceil(DATA_WIDTH/64)	Indicates that the read data in this transfer has been corrupted.
WPOISON	Master	ceil(DATA_WIDTH/64)	Indicates that the write data in this transfer has been corrupted.
CDPOISON	ACE master	ceil(DATA_WIDTH/64)	Indicates that the snoop data in this transfer has been corrupted.

If the Poison property is True, then the appropriate Poison signal must be present for all data channels that are present on that interface.

The validity of the Poison signaling is identical to the validity of the associated data.

Poison signaling is independent of error response signaling:

- It is permitted to signal an error with no Poison violation.
- It is permitted to signal a Poison violation without signaling an error response.

A 64-bit granule is defined to be an 8-byte address range that is aligned to an 8-byte boundary.

Where the transaction size, as indicated by **AxSIZE**, is less than 64-bits then it is permitted for the Poison bit to be different on each data beat. In this situation the receiving component must examine all data beats to determine if the 64-bit granule is poisoned.

Poison bits can be set for data lanes that are invalid for a transfer. For example, a 64-bit transfer on a 128-bit bus can have both Poison bits set.

E2.2 Parity use in AMBA

For safety-critical applications it is necessary to detect and possibly correct, transient and functional errors on individual wires within an SoC.

An error in a system component can propagate and cause multiple errors within connected components. Error detection and correction (EDC) is required to operate end-to-end, covering all logic and wires from source to destination.

One way to implement end-to-end protection, is to employ customized EDC schemes in components and implement a simple error detection scheme between components. Between these components there is no logic and single bit errors do not propagate to multi-bit errors. This section describes a parity scheme for detecting single-bit errors on the AMBA interface between components. Multi-bit errors can be detected if they occur in different parity signal groups. Figure E2-1 shows locations where parity can be used in AMBA.

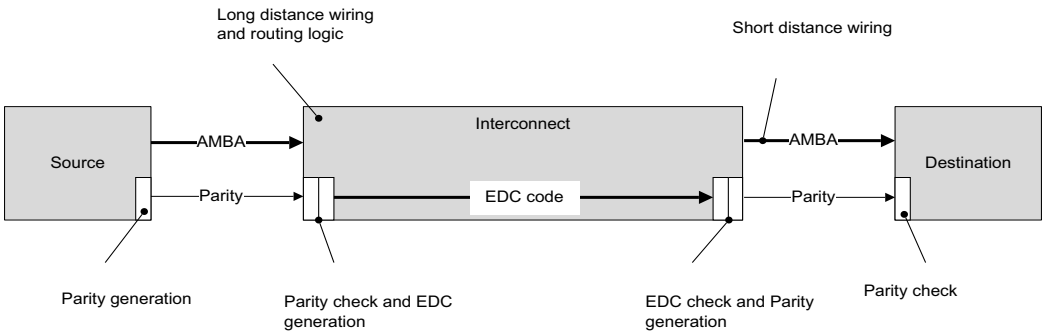


Figure E2-1 Parity use in AMBA

E2.3 Configuration of interface protection

The protection scheme employed on an interface is defined by the property Check_Type. The following Check_Type values are defined:

False

No checking signals on the interface. If the Check_Type property is not declared, it is considered to be False.

Odd_Parity_Byte_Data

Odd parity checking included for data signals with names that end in DATA. Each bit of the parity signal covers exactly 8 bits.

Odd_Parity_Byte_All

Odd parity checking included for all signals. Each bit of the parity signal generally covers up to 8 bits. However, a parity bit can cover more than 8 bits if the configuration requires it.

Interface protection is supported in the following interfaces:

- AXI5
- AXI5-Lite
- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

E2.4 Byte parity check signals

The following attributes are common to all the check signals added for byte parity interface protection:

- Odd parity is used.

  Odd parity means that check signals are added to groups of signals on the interface and driven such that there is always an odd number of asserted bits in that group.

- Parity signals covering data and payload are defined such that in most cases, there are no more than 8 bits per group.

  This limitation assumes that there is a maximum of 3 logic levels available in the timing budget for generating each parity bit.

- Parity signals covering critical control signals, which are likely to have a smaller timing budget available, are defined with a single odd parity bit. This single odd parity bit is the inversion of the original critical control signal.

- For a check signal that is wider than 1 bit:
  - Check bit [n] corresponds to bits [(8n+7):8n] in the payload.
  - If the payload is not an integer number of bytes, the most significant bit of the check signal covers fewer than 8-bits in the most significant portion of the payload.

- Check signals must be driven correctly in every cycle that the Check Enable term is True, see Table E2-2 on page E2-411.

- Parity signals must be driven appropriate to all the bits in the associated payload, irrespective of whether those bits are actively used in the transfer. For example, all bits of **WDATACHK** must be driven correctly when **WVALID** is asserted, even if some byte lanes are not being used.

- If none of the signals covered by a check signal are present on an interface, then the check signal is omitted from the interface.

- If some of the signals covered by a check signal are not present on an interface, then the missing signals are assumed to be LOW.

E2.5 Error detection behavior

This specification is not prescriptive regarding component or system behavior when a parity error is detected. Depending on the system and affected signals, a flipped bit can have a wide range of effects. It might be harmless, cause performance issues, cause data corruption, cause security violations, or deadlock. The transaction response is independent of parity error detection.

When an error is detected, the receiver can to do any of the following:

- Terminate or propagate the transaction. Termination might or might not be protocol compliant.
- Correct the parity check signal or propagate the signal in error.
- Update its memory or leave untouched. The location might be marked as poisoned.
- Signal an error response through other means, for example with an interrupt.

E2.6 Parity check signals

All of the following check signals are synchronous to **ACLK** and must be driven correctly every cycle that the Check Enable signal is HIGH.

Table E2-2 Parity check signals

Check signal	Signals covered	Check signal width	Granularity	Check enable
AWVALIDCHK	**AWVALID**	1	1	**ARESETn**
AWREADYCHK	**AWREADY**	1	1	**ARESETn**
AWIDCHK	Packed as {**AWIDUNQ**, **AWID**}	ceil((ID_W_WIDTH+1)/8) if **AWIDUNQ** is not present: ceil(ID_W_WIDTH/8)	1-8	**AWVALID**
AWADDRCHK	**AWADDR**	ceil(ADDR_WIDTH/8)	1-8	**AWVALID**
AWLENCHK	**AWLEN**	1	8	**AWVALID**
AWCTLCHK0	**AWSIZE**, **AWBURST**, **AWLOCK**, **AWPROT**	1	1-9	**AWVALID**
AWCTLCHK1	**AWREGION**, **AWCACHE**, **AWQOS**	1	4-12	**AWVALID**
AWCTLCHK2	**AWDOMAIN**, **AWSNOOP**, **AWUNIQUE**, **AWBAR**	1	2-9	**AWVALID**
AWCTLCHK3	**AWATOP**, **AWCMO**, **AWTAGOP**	1	2-10	**AWVALID**
AWNSAIDCHK	**AWNSAID**	1	4	**AWVALID**
AWUSERCHK	**AWUSER**	ceil(USER_REQ_WIDTH)/8)	1-8	**AWVALID**
AWSTASHNIDCHK	**AWSTASHNID**, **AWSTASHNIDEN**	1	12	**AWVALID**
AWSTASHLPIDCHK	**AWSTASHLPID**, **AWSTASHLPIDEN**	1	6	**AWVALID**
AWTRACECHK	**AWTRACE**	1	1	**AWVALID**
AWLOOPCHK	**AWLOOP**	1	1-8	**AWVALID**
AWMMUCHK	**AWMMUATST**, **AWMMUFLOW**, **AWMMUSECSID**, **AWMMUSSIDV**	1	3-4	**AWVALID**
AWMMUSIDCHK	**AWMMUSID**	ceil(SID_WIDTH/8)	1-8	**AWVALID**
AWMMUSSIDCHK	**AWMMUSSID**	ceil(SSID_WIDTH/8)	1-8	**AWVALID**
AWMPAMCHK	**AWMPAM**	1	11	**AWVALID**

Table E2-2 Parity check signals (continued)

Check signal	Signals covered	Check signal width	Granularity	Check enable
WVALIDCHK	**WVALID**	1	1	**ARESETn**
WREADYCHK	**WREADY**	1	1	**ARESETn**
WDATACHK	**WDATA**	DATA_WIDTH/8	8	**WVALID**
WSTRBCHK	**WSTRB**	ceil(DATA_WIDTH/64)	1-8	**WVALID**
WTAGCHK	**WTAG**, **WTAGUPDATE**	ceil(DATA_WIDTH/128) **WTAGCHK[n]** is the parity of {**WTAGUPDATE[n]**, **WTAG[4n+3:4n]**}	5	**WVALID**
WLASTCHK	**WLAST**	1	1	**WVALID**
WUSERCHK	**WUSER**	ceil(USER_DATA_WIDTH/8)	1-8	**WVALID**
WPOISONCHK	**WPOISON**	ceil(DATA_WIDTH/512)	1-8	**WVALID**
WTRACECHK	**WTRACE**	1	1	**WVALID**
BVALIDCHK	**BVALID**	1	1	**ARESETn**
BREADYCHK	**BREADY**	1	1	**ARESETn**
BIDCHK	**Packed as {BIDUNQ**, **BID}**	ceil((ID_W_WIDTH+1)/8) if **BIDUNQ** is not present: ceil(ID_W_WIDTH/8)	1-8	**BVALID**
BRESPCHK	**BRESP**, **BCOMP**, **BPERSIST**, **BTAGMATCH**	1	2-7	**BVALID**
BUSERCHK	**BUSER**	ceil(USER_RESP_WIDTH/8)	1-8	**BVALID**
BTRACECHK	**BTRACE**	1	1	**BVALID**
BLOOPCHK	**BLOOP**	1	1-8	**BVALID**
ARVALIDCHK	**ARVALID**	1	1	**ARESETn**
ARREADYCHK	**ARREADY**	1	1	**ARESETn**
ARIDCHK	**Packed as {ARIDUNQ**, **ARID}**	ceil((ID_R_WIDTH+1)/8) if **ARIDUNQ** is not present: ceil(ID_R_WIDTH/8)	1-8	**ARVALID**
ARADDRCHK	**ARADDR**	ceil(ADDR_WIDTH/8)	1-8	**ARVALID**
ARLENCHK	**ARLEN**	1	8	**ARVALID**
ARCTLCHK0	**ARSIZE**, **ARBURST**, **ARLOCK**, **ARPROT**	1	1-9	**ARVALID**
ARCTLCHK1	**ARREGION**, **ARCACHE**, **ARQOS**	1	4-12	**ARVALID**

Table E2-2 Parity check signals (continued)

Check signal	Signals covered	Check signal width	Granularity	Check enable
ARCTLCHK2	**ARDOMAIN**, **ARSNOOP**, **ARBAR**	1	2-8	**ARVALID**
ARCTLCHK3	**ARVMIDEXT**, **ARCHUNKEN**, **ARTAGOP**	1	1-7	**ARVALID**
ARNSAIDCHK	**ARNSAID**	1	4	**ARVALID**
ARUSERCHK	**ARUSER**	ceil(USER_REQ_WIDTH/8)	1-8	**ARVALID**
ARTRACECHK	**ARTRACE**	1	1	**ARVALID**
ARLOOPCHK	**ARLOOP**	1	1-8	**ARVALID**
ARMMUCHK	**ARMMUATST**, **ARMMUFLOW**, **ARMMUSECSID**, **ARMMUSSIDV**	1	3-4	**ARVALID**
ARMMUSIDCHK	**ARMMUSID**	ceil(SID_WIDTH/8)	1-8	**ARVALID**
ARMMUSSIDCHK	**ARMMUSSID**	ceil(SSID_WIDTH/8)	1-8	**ARVALID**
ARMPAMCHK	**ARMPAM**	1	11	**ARVALID**
RVALIDCHK	**RVALID**	1	1	**ARESETn**
RREADYCHK	**RREADY**	1	1	**ARESETn**
RIDCHK	Packed as {**RIDUNQ, RID**}	ceil((ID_R_WIDTH+1)/8) if **RIDUNQ** is not present: ceil(ID_R_WIDTH/8)	1-8	**RVALID**
RDATACHK	**RDATA**	DATA_WIDTH/8	8	**RVALID**
RTAGCHK	**RTAG**	ceil(DATA_WIDTH/128) **RTAGCHK[n]** is the parity of **RTAG[4n+3:4n]**	4	**RVALID**
RRESPCHK	**RRESP**	1	2-4	**RVALID**
RLASTCHK	**RLAST**	1	1	**RVALID**
RUSERCHK	**RUSER**	ceil(USER_DATA_WIDTH + USER_RESP_WIDTH/8)	1-8	**RVALID**
RPOISONCHK	**RPOISON**	ceil(DATA_WIDTH/512)	1-8	**RVALID**
RTRACECHK	**RTRACE**	1	1	**RVALID**
RLOOPCHK	**RLOOP**	1	1-8	**RVALID**
RCHUNKCHK	**RCHUNKV**, **RCHUNKNUM**, **RCHUNKSTRB**	1	1-14	**RVALID**

Table E2-2 Parity check signals (continued)

Check signal	Signals covered	Check signal width	Granularity	Check enable
ACVALIDCHK	**ACVALID**	1	1	**ARESETn**
ACREADYCHK	**ACREADY**	1	1	**ARESETn**
ACADDRCHK	**ACADDR**	ceil(ADDR_WIDTH/8)	1-8	**ACVALID**
ACCTLCHK	**ACSNOOP**, **ACPROT**	1	3-7	**ACVALID**
ACVMIDEXTCHK	**ACVMIDEXT**	1	4	**ACVALID**
ACTRACECHK	**ACTRACE**	1	1	**ACVALID**
CRVALIDCHK	**CRVALID**	1	1	**ARESETn**
CRREADYCHK	**CRREADY**	1	1	**ARESETn**
CRRESPCHK	**CRRESP**	1	5	**CRVALID**
CRTRACECHK	**CRTRACE**	1	1	**CRVALID**
CRNSAIDCHK	**CRNSAID**	1	4	**CRVALID**
CDVALIDCHK	**CDVALID**	1	1	**ARESETn**
CDREADYCHK	**CDREADY**	1	1	**ARESETn**
CDDATACHK	**CDDATA**	DATA_WIDTH/8	8	**CDVALID**
CDLASTCHK	**CDLAST**	1	1	**CDVALID**
CDPOISONCHK	**CDPOISON**	ceil(DATA_WIDTH/512)	1-8	**CDVALID**
CDTRACECHK	**CDTRACE**	1	1	**CDVALID**
RACKCHK	**RACK**	1	1	**ARESETn**
WACKCHK	**WACK**	1	1	**ARESETn**
VAWQOSACCEPTCHK	**VAWQOSACCEPT**	1	4	**ARESETn**
VARQOSACCEPTCHK	**VARQOSACCEPT**	1	4	**ARESETn**
AWAKEUPCHK	**AWAKEUP**	1	1	**ARESETn**
ACWAKEUPCHK	**ACWAKEUP**	1	1	**ARESETn**
SYSCOREQCHK	**SYSCOREQ**	1	1	-
SYSCOACKCHK	**SYSCOACK**	1	1	-

Part F

AMBA ACE5, ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACP Interface Specification

Chapter F1
AMBA ACE5

This chapter specifies the new capabilities in the ACE5 protocol specification. It contains the following sections:

F1.1 About the ACE5 protocol

ACE5 extends the capabilities of the ACE protocol that is described in Part D *AMBA ACE and ACE-Lite Protocol Specification*. Table F1-1 summarizes the properties used to declare capabilities.

Table F1-1 Properties that can be set on an ACE5 interface

Property	Description
DVM_v8	Specifies that a component supports DVMv8 and DVMv7 message protocols. See *DVM message versions* on page D13-323.
DVM_v8.1	Specifies that a component supports DVMv8.1, DVMv8 and DVMv7 message protocols. See *DVM message versions* on page D13-323.
CMO_On_Read	Indicates whether an interface supports cache maintenance operations on the read channels. See *CMO signaling* on page D7-271.
Persist_CMO	Adds an additional cache maintenance operation that is used to provide a cache clean to the *Point of Persistence* operation. See *CMO transactions* on page D7-267.
Check_Type	Adds data or interface level parity signals for error detection. See Chapter E2 *Interface and data protection*.
Poison	Adds Poison signaling that is used to indicate that a set of data bytes have been corrupted. See Chapter E2 *Interface and data protection*.
QoS_Accept	Adds two additional QoS interface signals that enable a slave to indicate the QoS value of transactions that it will accept. See *QoS Accept signaling* on page E1-360.
Trace_Signals	Adds a Trace signal, which is associated with each channel, to support the debugging, tracing, and performance measurement of systems. See *Trace signals* on page E1-357.
Loopback_Signals	Adds loopback signaling that permits an agent that is issuing transactions to store information relating to the transaction in an indexed table. See *User Loopback signaling* on page E1-359.
Wakeup_Signals	Adds two wakeup signals that are used to indicate that there is activity that is associated with the interface. See *Wake-up Signaling* on page E1-362.
Coherency_Connection_Signals	Adds signaling to connect or disconnect this interface from the coherency system. See *Coherency Connection signaling* on page E1-364
Untranslated_Transactions[a]	Adds untranslated transaction support and permits different transactions on the same interface to use different translation schemes. See *Untranslated transactions* on page E1-369.
NSAccess_Identifiers	Adds Non-secure access identifiers that support the storage and processing of protected data. See *Non-secure access identifiers* on page E1-376.

Table F1-1 Properties that can be set on an ACE5 interface (continued)

Property	Description
MPAM_Support	Used to indicate whether an interface supports MPAM. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
Unique_ID_Support	Indicates whether an interface supports the Unique ID Indicator. See *Unique ID indicator* on page E1-383.
Consistent_DECERR	Used to define whether a slave signals DECERR consistently across all beats of read and write response. See *Consistent DECERR response* on page E1-401.
DVM_Message_Support	Used to describe the types of DVM messages that are supported by an interface. See *About DVM transactions* on page D13-318
Exclusive_Accesses	Used to define whether a master issues exclusive accesses or whether a slave supports them. See *Exclusive accesses* on page E1-399.
Max_Transaction_Bytes	Defines the maximum size of a transaction in bytes. See *Maximum transaction size and boundary* on page E1-400.
Regular_Transactions_Only	Used to define whether a master issues only Regular type transactions and if a slave only supports Regular transactions. See *Regular transactions property* on page A3-53.
Shareable_Transactions	Used to define whether a master issues Inner Shareable or Outer Shareable transactions and whether a slave supports them. See *Shareable transactions* on page E1-399.

a. Support in ACE5 has restrictions. See *Use of Untranslated Transactions with ACE5* on page E1-372.

F1.2 Signal descriptions

This section introduces the additional ACE5 interface signals that support the new capabilities. It contains the following subsections:

- *Changes to existing ACE channels*
- *Additional signaling* on page F1-424

See Chapter D2 *Signal Descriptions* for details of the ACE interface signals.

F1.2.1 Changes to existing ACE channels

Additional signals are required on the following ACE channels:

- *Write address channel*
- *Write data channel* on page F1-421
- *Write response channel* on page F1-421
- *Read address channel* on page F1-422
- *Read data channel* on page F1-423
- *Snoop address channel* on page F1-423
- *Snoop response channel* on page F1-423
- *Snoop data channel* on page F1-424

Parity check signals are not included in the tables in this section.

Write address channel

Table F1-2 shows the additional write address channel signals.

Table F1-2 Write address channel signals

Signal	Source	Property	Description
AWTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
AWLOOP	Master	Loopback_Signals	Loopback value for a write transaction. Reflected back on **BLOOP**. See *User Loopback signaling* on page E1-359.
AWMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSID	Master	Untranslated_Transactions	Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSSIDV	Master	Untranslated_Transactions	Indicates if the AWMMUSSID signal is valid. See *Untranslated transactions* on page E1-369.
AWMMUSSID	Master	Untranslated_Transactions	Substream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUATST	Master	Untranslated_Transactions	Indicates whether a write transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.

Table F1-2 Write address channel signals (continued)

Signal	Source	Property	Description
AWNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a write transaction. See *Non-secure access identifiers* on page E1-376.
AWMPAM	Master	MPAM_Support	Write address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385.
AWIDUNQ	Master	Unique_ID_Support	Write address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.

Write data channel

Table F1-3 shows the additional write data channel signals.

Table F1-3 Write data channel signals

Signal	Source	Property	Description
WPOISON	Master	Poison	Indicates that the write data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
WTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.

Write response channel

Table F1-4 shows the additional write response channel signals.

Table F1-4 Write response channel signals

Signal	Source	Property	Description
BTRACE	Interconnect	Trace_signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
BLOOP	Interconnect	Loopback_Signals	Loopback value for a write response. See *User Loopback signaling* on page E1-359.
BIDUNQ	Slave	Unique_ID_Support	Write response channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.

Read address channel

Table F1-5 shows the additional read address channel signals.

Table F1-5 Read address channel signals

Signal	Source	Property	Description
ARVMIDEXT	Master	DVM_v8.1	VMID extension for a read address. See *DVM message versions* on page D13-323.
ARTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
ARLOOP	Master	Loopback_Signals	Loopback value for a read transaction. See *User Loopback signaling* on page E1-359.
ARMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSID	Master	Untranslated_Transactions	Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSSIDV	Master	Untranslated_Transactions	Substream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSSID	Master	Untranslated_Transactions	Indicates whether the ARMMUSSID signal is valid. See *Untranslated transactions* on page E1-369.
ARMMUATST	Master	Untranslated_Transactions	Indicates whether a read transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.
ARNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a read transaction. See *Non-secure access identifiers* on page E1-376.
ARMPAM	Master	MPAM_Support	Read address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
ARIDUNQ	Master	Unique_ID_Support	Read address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.

Read data channel

Table F1-6 shows the additional read data channel signals.

Table F1-6 Read data channel signals

Signal	Source	Property	Description
RPOISON	Interconnect	Poison	Indicates that the read data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
RTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
RLOOP	Interconnect	Loopback_Signals	Loopback value for a read response. See *User Loopback signaling* on page E1-359.
RIDUNQ	Slave	Unique_ID_Support	Read data channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.

Snoop address channel

Table F1-7 shows the additional snoop address channel signals.

Table F1-7 Snoop address channel signals

Signal	Source	Property	Description
ACVMIDEXT	Interconnect	DVM_v8.1	VMID extension for a snoop address. See *DVM message versions* on page D13-323.
ACTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.

Snoop response channel

Table F1-8 shows the additional snoop response channel signals.

Table F1-8 Snoop response channel signals

Signal	Source	Property	Description
CRTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system See *Trace signals* on page E1-357.
CRNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a snoop response. See *Non-secure access identifiers* on page E1-376.

Snoop data channel

Table F1-9 shows the additional snoop data channel signals.

Table F1-9 Snoop data channel signals

Signal	Source	Property	Description
CDPOISON	Master	Poison	Indicates that the snoop data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
CDTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.

F1.2.2 Additional signaling

Table F1-10 shows the additional signaling required on the ACE5 interface to support the new capabilities.

Table F1-10 Additional signals

Signal	Source	Property	Description
VAWQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for write transactions. See *QoS Accept signaling* on page E1-360.
VARQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for read transactions. See *QoS Accept signaling* on page E1-360.
AWAKEUP	Master	Wakeup_Signals	Indicates that activity is initiated on the write or read address channels. See *Wake-up Signaling* on page E1-362.
ACWAKEUP	Interconnect	Wakeup_Signals	Indicates that activity is initiated on the snoop address channels. See *Wake-up Signaling* on page E1-362.
SYSCOREQ	Master	Coherency_Connection_Signals	Coherency connect request. See *Coherency Connection signaling* on page E1-364.
SYSCOACK	Interconnect	Coherency_Connection_Signals	Coherency connect acknowledge. See *Coherency Connection signaling* on page E1-364.

Chapter F2
AMBA ACE5-Lite

This chapter specifies the new capabilities in the ACE5-Lite protocol specification. It contains the following sections:

F2.1 About the ACE5-Lite protocol

ACE5-Lite extends the capabilities of the ACE-Lite protocol that is specified in Chapter D11 *AMBA ACE-Lite*. To maintain compatibility, a property is used to declare a new capability. Table F2-1 summarizes the properties.

Table F2-1 Properties that can be set on an ACE5-Lite interface

Property	Description
Atomic_Transactions	Adds Atomic transactions that perform more than just a single access, and have some form of operation that is associated with the transaction. See *Atomic transactions* on page E1-342.
Cache_Stash_Transactions	Adds Cache Stashing transactions that enable one component to indicate that a particular cache line should be placed in the cache of another component in the system. See *Cache stashing* on page E1-351.
DeAllocation_Transactions	Adds Deallocation transactions that permit an IO coherent master to influence the allocation of cache lines in the system. See *Deallocating transactions* on page E1-355.
Persist_CMO	Adds an additional cache maintenance operation that is used to provide a cache clean to the point of persistence operation. See *CMO transactions* on page D7-267.
Check_Type	Adds data or interface level parity signals for error detection. See Chapter E2 *Interface and data protection*.
Poison	Adds Poison signaling, which is used to indicate that a set of data bytes have been previously corrupted. See Chapter E2 *Interface and data protection*.
QoS_Accept	Adds two additional QoS interface signals that enable a slave to indicate the QoS value of transactions that it will accept. See *QoS Accept signaling* on page E1-360.
Trace_Signals	Adds a Trace signal that is associated with each channel to support the debugging, tracing, and performance measurement of systems. See *Trace signals* on page E1-357.
Loopback_Signals	Adds loopback signaling, which permits an agent that is issuing transactions to store information relating to the transaction in an indexed table. See *User Loopback signaling* on page E1-359
Wakeup_Signaling	Adds wakeup signaling, which is used to indicate that there is activity that is associated with the interface. See *Wake-up Signaling* on page E1-362.
Untranslated_Transactions	Adds untranslated transaction support and permits different transactions on the same interface to use different translation schemes. See *Untranslated transactions* on page E1-369.
NSAccess_Identifiers	Adds Non-secure access identifiers that support the storage and processing of protected data. See *Non-secure access identifiers* on page E1-376.
MPAM_Support	Used to indicate whether an interface supports MPAM. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385.
CMO_On_Read	Indicates whether an interface supports cache maintenance operations on the read channels. See *CMO signaling* on page D7-271.

Table F2-1 Properties that can be set on an ACE5-Lite interface (continued)

Property	Description
CMO_On_Write	Indicates whether a component supports cache maintenance operations on write channels. See *CMO signaling* on page D7-271.
Read_Interleaving_Disabled	Indicates whether an interface supports the interleaving of read data beats from different transactions. See *Read interleaving property* on page E1-382.
Read_Data_Chunking	Indicates whether an interface supports the return of read data in reorderable chunks. See *Read data chunking* on page E1-378.
Unique_ID_Support	Indicates whether an interface supports the Unique ID Indicator. See *Unique ID indicator* on page E1-383.
Write_Plus_CMO	Used to indicate whether a component supports combined write and CMOs on the write channels. See *Write with CMO configuration* on page D7-277.
MTE_Support	Used to indicate that a component supports the Memory Tagging Extension. See *Memory tagging* on page E1-387.
Prefetch_Transaction	Used to indicate whether a component supports prefetching. See *Prefetch request and response* on page E1-396
WriteZero_Transaction	Used to indicate whether an interface supports the WriteZero transaction. See *Write zero with no data* on page E1-398
Consistent_DECERR	Used to define whether a slave signals DECERR consistently across all beats of read and write response. See *Consistent DECERR response* on page E1-401.
Exclusive_Accesses	Used to define whether a master issues exclusive accesses or whether a slave supports them. See *Exclusive accesses* on page E1-399.
Max_Transaction_Bytes	Defines the maximum size of a transaction in bytes. See *Maximum transaction size and boundary* on page E1-400.
Regular_Transactions_Only	Used to define whether a master issues only Regular type transactions and if a slave only supports Regular transactions. See *Regular transactions property* on page A3-53.
Shareable_Transactions	Used to define whether a master issues Inner Shareable or Outer Shareable transactions and whether a slave supports them. See *Shareable transactions* on page E1-399.

F2.2 ACE5-Lite signal descriptions

This section introduces the additional ACE5-Lite interface signals that support the new capabilities. It contains the following subsections:

- *Changes to existing ACE-Lite channels*
- *Additional signaling* on page F2-433

See Chapter D11 *AMBA ACE-Lite* for details of the ACE-Lite interface signals.

F2.2.1 Changes to existing ACE-Lite channels

Additional signals are required on the following ACE-Lite channels:

- *Write address channel*
- *Write data channel* on page F2-430
- *Write response channel* on page F2-430
- *Read address channel* on page F2-431
- *Read data channel* on page F2-432

Parity check signals are not included in the tables in this section.

Write address channel

Table F2-2 shows the additional write address channel signals.

Table F2-2 Write address channel signals

Signal	Source	Property	Description
AWATOP	Master	Atomic_Transactions	Indicates the type and endianness of atomic transactions. See *Atomic transactions* on page E1-342.
AWSTASHNID	Master	Cache_Stash_Transactions	Node Identifier of the target for a stash operation. See *Cache stashing* on page E1-351.
AWSTASHNIDEN	Master	Cache_Stash_Transactions	Write address Stash Node ID Enable. See *Cache stashing* on page E1-351.
AWSTASHLPID	Master	Cache_Stash_Transactions	Logical Processor Identifier within the target for a stash operation. See *Cache stashing* on page E1-351.
AWSTASHLPIDEN	Master	Cache_Stash_Transactions	Indicates whether the **AWSTASHLPID** signal is valid. See *Cache stashing* on page E1-351.
AWTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
AWLOOP	Master	Loopback_Signals	Loopback value for a write transaction. Reflected back on **BLOOP**. See *User Loopback signaling* on page E1-359.

Table F2-2 Write address channel signals (continued)

Signal	Source	Property	Description
AWMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSID	Master	Untranslated_Transactions	Stream Identifier for a write transaction. See *Untranslated transactions* on page E1-369.
AWMMUSSIDV	Master	Untranslated_Transactions	Indicates if the AWMMUSSID signal is valid. See *Untranslated transactions* on page E1-369.
AWMMUSSID	Master	Untranslated_Transactions	Substream Identifier for a write transaction. This signal is only valid if **AWMMUSSIDV** is asserted. See *Untranslated transactions* on page E1-369.
AWMMUATST	Master	Untranslated_Transactions	Indicates whether a write transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.
AWMMUFLOW	Master	Untranslated_Transactions	Indicates the SMMU flow for managing translation faults. See *Untranslated transactions* on page E1-369.
AWNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a write transaction. See *Non-secure access identifiers* on page E1-376.
AWIDUNQ	Master	Unique_ID_Support	Write address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
AWCMO	Master	CMO_On_Write	Write address channel CMO indicator. See *CMO signaling* on page D7-271.
AWMPAM	Master	MPAM_Support	Write address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
AWTAGOP	Master	MTE_Support	Write request tag operation. See *MTE signaling* on page E1-388

Write data channel

Table F2-3 shows the additional write data channel signals.

Table F2-3 Write data channel signals

Signal	Source	Property	Description
WPOISON	Master	Poison	Indicates that the write data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
WTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
WTAG	Master	MTE_Support	Indicates the tag that is associated with write data. See *MTE signaling* on page E1-388
WTAGUPDATE	Master	MTE_Support	Indicates which tags must be written to memory in an Update operation. See *MTE signaling* on page E1-388

Write response channel

Table F2-4 shows the additional write response channel signals.

Table F2-4 Write response channel signals

Signal	Source	Property	Description
BTRACE	Interconnect	Trace_signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
BLOOP	Interconnect	Loopback_Signals	Loopback value for a write response. See *User Loopback signaling* on page E1-359.
BIDUNQ	Slave	Unique_ID_Support	Write response channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
BTAGMATCH	Slave	MTE_Support	Indicates the result of a tag comparison on a write transaction. See *MTE signaling* on page E1-388
BCOMP	Slave	CMO_On_Write, Persist_CMO, MTE_Support	Response flag that indicates that a write is observable. See *PCMO response on the B channel* on page D7-275 and *MTE signaling* on page E1-388.
BPERSIST	Slave	CMO_On_Write, Persist_CMO	Response flag that indicates that write data is updated in persistent memory. See *PCMO response on the B channel* on page D7-275

Read address channel

Table F2-5 shows the additional read address channel signals.

Table F2-5 Read address channel signals

Signal	Source	Property	Description
ARTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
ARLOOP	Master	Loopback_Signals	Loopback value for a read transaction. Reflected back on **RLOOP**. See *User Loopback signaling* on page E1-359.
ARMMUSECSID	Master	Untranslated_Transactions	Secure Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSID	Master	Untranslated_Transactions	Stream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUSSIDV	Master	Untranslated_Transactions	Indicates whether the **ARMMUSSID** signal is valid. See *Untranslated transactions* on page E1-369.
ARMMUSSID	Master	Untranslated_Transactions	Substream Identifier for a read transaction. See *Untranslated transactions* on page E1-369.
ARMMUATST	Master	Untranslated_Transactions	Indicates whether a read transaction has undergone PCIe ATS translation. See *Untranslated transactions* on page E1-369.
ARMMUFLOW	Master	Untranslated_Transactions	Indicates the SMMU flow for managing translation faults. See *Untranslated transactions* on page E1-369.
ARNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a read transaction. See *Non-secure access identifiers* on page E1-376.
ARMPAM	Master	MPAM_Support	Read address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
ARCHUNKEN	Master	Read_Data_Chunking	Read data chunking enable. See *Read data chunking* on page E1-378.
ARIDUNQ	Master	Unique_ID_Support	Read address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
ARTAGOP	Master	MTE_Support	Read request tag operation. See *MTE signaling* on page E1-388

Read data channel

Table F2-6 shows the additional read data channel signals. Parity check signals are not included in this table.

Table F2-6 Read data channel signals

Signal	Source	Property	Description
RPOISON	Interconnect	Poison	Indicates that the read data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
RTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
RLOOP	Interconnect	Loopback_Signals	Loopback value for a read response. See *User Loopback signaling* on page E1-359.
RIDUNQ	Slave	Unique_ID_Support	Read data channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
RCHUNKV	Slave	Read_Data_Chunking	Valid signal of **RCHUNKNUM** and **RCHUNKSTRB**. See *Read data chunking* on page E1-378.
RCHUNKNUM	Slave	Read_Data_Chunking	Read data chunk number. See *Read data chunking* on page E1-378.
RCHUNKSTRB	Slave	Read_Data_Chunking	Read data chunk strobe. See *Read data chunking* on page E1-378.
RTAG	Slave	MTE_Support	The tag that is associated with read data. See *MTE signaling* on page E1-388

F2.2.2 Additional signaling

The following ancillary signaling is required on the ACE5-Lite interface to support the new capabilities. Table F2-7 shows the additional QoS accept and Wakeup signaling.

Table F2-7 QoS accept and Wakeup signals

Signal	Source	Property	Description
VAWQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for write transactions. See *QoS Accept signaling* on page E1-360.
VARQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for read transactions. See *QoS Accept signaling* on page E1-360.
AWAKEUP	Master	Wakeup_Signals	Indicates that activity is initiated on the write or read address channels. See *Wake-up Signaling* on page E1-362.

Chapter F3
AMBA ACE5-LiteDVM

This chapter describes the new ACE5-LiteDVM protocol specification that is introduced in AMBA 5. It contains the following sections:

F3.1 About the ACE5-LiteDVM protocol

Issue F of the AMBA AXI and ACE protocol specification introduced the AMBA protocol ACE5-LiteDVM.

ACE5-LiteDVM extends the capabilities of the ACE5-Lite protocol that is specified in Chapter F2 *AMBA ACE5-Lite*.

ACE5-LiteDVM is identical to ACE5-Lite, with the addition of support for IO coherent components that include SMMU functionality and therefore receive DVM transactions.

Untranslated transactions are not supported on ACE5-LiteDVM interfaces, since this interface is intended to be used after translation has occurred.

An ACE5-LiteDVM master must be able to receive DVM messages on the AC channel. For DVM Syncronization messages, the master must also be able to send DVM Complete messages on the AR channel.

An interconnect with an ACE5-LiteDVM interface can issue DVM messages on the AC channel, and must be able to receive DVM Complete messages on the AR channel.

To maintain compatibility, a property is used to declare a new capability. Table F3-1 summarizes the properties.

Table F3-1 Properties that can be set on an ACE5-LiteDVM interface

Property	Description
Atomic_Transactions	Adds Atomic transactions that perform more than just a single access, and have some form of operation that is associated with the transaction. See *Atomic transactions* on page E1-342.
DVM_v8	Specifies that a component supports DVMv8 and DVMv7 message protocols. See *DVM message versions* on page D13-323.
DVM_v8.1	Specifies that a component supports DVMv8.1, DVMv8, and DVMv7 message protocols. See *DVM message versions* on page D13-323.
DVM_v8.4	Specifies that a component supports DVMv8.4, DVMv8.1, DVMv8, and DVMv7 message protocols. See *DVM message versions* on page D13-323.
Cache_Stash_Transactions	Adds Cache Stashing transactions that enable one component to indicate that a particular cache line should be placed in the cache of another component in the system. See *Cache stashing* on page E1-351.
DeAllocation_Transactions	Adds Deallocation transactions that permit an IO coherent master to influence the allocation of cache lines in the system. See *Deallocating transactions* on page E1-355.
Persist_CMO	Adds an additional cache maintenance operation that is used to provide a cache clean to the point of persistence operation. See *CMO transactions* on page D7-267.
Check_Type	Adds data checking signaling, which is used to detect, and potentially correct, data bytes that might have been corrupted. See Chapter E2 *Interface and data protection*.
Poison	Adds Poison signaling, which is used to indicate that a set of data bytes have been previously corrupted. See Chapter E2 *Interface and data protection*.

Table F3-1 Properties that can be set on an ACE5-LiteDVM interface (continued)

Property	Description
QoS_Accept	Adds two additional QoS interface signals that enable a slave to indicate the QoS value of transactions that it will accept. See *QoS Accept signaling* on page E1-360.
Trace_Signals	Adds a Trace signal, which is associated with each channel, to support the debugging, tracing, and performance measurement of systems. See *Trace signals* on page E1-357.
Loopback_Signals	Adds loopback signaling that permits an agent that is issuing transactions to store information relating to the transaction in an indexed table. See *User Loopback signaling* on page E1-359.
Wakeup_Signals	Adds two wakeup signals, which are used to indicate that there is activity that is associated with the interface. See *Wake-up Signaling* on page E1-362.
Coherency_Connection_Signals	Adds signaling to connect or disconnect this interface from the coherency system. See *Coherency Connection signaling* on page E1-364.
NSAccess_Identifiers	Adds Non-secure access identifiers that support the storage and processing of protected data. See *Non-secure access identifiers* on page E1-376.
MPAM_Support	Used to indicate whether an interface supports MPAM. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385.
Unique_ID_Support	Used to indicate if an interface supports the Unique ID Indicator: See *Unique ID indicator* on page E1-383
CMO_On_Write	Indicates whether a component supports cache maintenance operations on write channels. See *CMO signaling* on page D7-271.
CMO_On_Read	Indicates whether an interface supports cache maintenance operations on the read channels. See *CMO signaling* on page D7-271.
Read_Interleaving_Disabled	Used to indicate whether an interface supports the interleaving of read data beats from different transactions. See *Read interleaving property* on page E1-382
Read_Data_Chunking	Used to indicate whether an interface supports the return of read data in reorderable chunks. See *Read data chunking* on page E1-378
Write_Plus_CMO	Used to indicate whether a component supports combined write and CMOs on the write channels. See *Write with CMO configuration* on page D7-277
DVM_Message_Support	Used to describe the types of DVM messages that are supported by an interface. See *About DVM transactions* on page D13-318

Table F3-1 Properties that can be set on an ACE5-LiteDVM interface (continued)

Property	Description
MTE_Support	Used to indicate that a component supports the Memory Tagging Extension. See *Memory tagging* on page E1-387
Prefetch_Transaction	Used to indicate whether a component supports prefetching. See *Prefetch request and response* on page E1-396
WriteZero_Transaction	Used to indicate whether an interface supports the WriteZero transaction. See *Write zero with no data* on page E1-398
Consistent_DECERR	Used to define whether a slave signals DECERR consistently across all beats of read and write response. See *Consistent DECERR response* on page E1-401.
Exclusive_Accesses	Used to define whether a master issues exclusive accesses or whether a slave supports them. See *Exclusive accesses* on page E1-399.
Max_Transaction_Bytes	Defines the maximum size of a transaction in bytes. See *Maximum transaction size and boundary* on page E1-400.
Regular_Transactions_Only	Used to define whether a master issues only Regular type transactions and if a slave only supports Regular transactions. See *Regular transactions property* on page A3-53.
Shareable_Transactions	Used to define whether a master issues Inner Shareable or Outer Shareable transactions and whether a slave supports them. See *Shareable transactions* on page E1-399.

F3.2 ACE5-LiteDVM signal descriptions

This section introduces the additional ACE5-LiteDVM interface signals that support the new capabilities. It contains the following subsections:

- *Changes to existing ACE-Lite channels*
- *Additional channels* on page F3-442
- *Additional signaling* on page F3-443

See Chapter D11 *AMBA ACE-Lite* for details of the ACE-Lite interface signals.

F3.2.1 Changes to existing ACE-Lite channels

Additional signals are required on the following ACE-Lite channels:

- *Write address channel*
- *Write data channel* on page F3-440
- *Write response channel* on page F3-441
- *Read address channel* on page F3-441
- *Read data channel* on page F3-442

Parity check signals are not included in the following tables in this section.

Write address channel

Table F3-2 shows the additional write address channel signals.

Table F3-2 Write address channel signals

Signal	Source	Property	Description
AWATOP	Master	Atomic_Transactions	Indicates the type and endianness of atomic transactions. See *Atomic transactions* on page E1-342.
AWSTASHNID	Master	Cache_Stash_Transactions	Node Identifier of the target for a stash operation. See *Cache stashing* on page E1-351.
AWSTASHNIDEN	Master	Cache_Stash_Transactions	Indicates whether the **AWSTASHNID** signal is valid. See *Cache stashing* on page E1-351.
AWSTASHLPID	Master	Cache_Stash_Transactions	Logical Processor Identifier within the target for a stash operation. See *Cache stashing* on page E1-351.
AWSTASHLPIDEN	Master	Cache_Stash_Transactions	Indicates whether a write transaction has undergone PCIe ATS translation See *Cache stashing* on page E1-351.
AWTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
AWLOOP	Master	Loopback_Signals	Loopback value for a write transaction. See *User Loopback signaling* on page E1-359.

Table F3-2 Write address channel signals (continued)

Signal	Source	Property	Description
AWMPAM	Master	MPAM_Support	Write address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385
AWNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a write transaction. See *Non-secure access identifiers* on page E1-376.
AWIDUNQ	Master	Unique_ID_Support	Write address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
AWCMO	Master	CMO_On_Write	Write address channel CMO indicator. See *CMO signaling* on page D7-271.
AWTAGOP	Master	MTE_Support	Write request tag operation. See *MTE signaling* on page E1-388

Write data channel

Table F3-3 shows the additional write data channel signals.

Table F3-3 Write data channel signals

Signal	Source	Property	Description
WPOISON	Master	Poison	Indicates that the write data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
WTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
WTAG	Master	MTE_Support	Indicates the tag that is associated with write data. See *MTE signaling* on page E1-388.
WTAGUPDATE	Master	MTE_Support	Indicates which tags must be written to memory in an Update operation. See *MTE signaling* on page E1-388

Write response channel

Table F3-4 shows the additional write response channel signals.

Table F3-4 Write response channel signals

Signal	Source	Property	Description
BTRACE	Interconnect	Trace_signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
BLOOP	Interconnect	Loopback_Signals	Loopback value for a write response. See *User Loopback signaling* on page E1-359.
BIDUNQ	Slave	Unique_ID_Support	Write response channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
BTAGMATCH	Slave	MTE_Support	Indicates the result of a tag comparison on a write transaction. See *MTE signaling* on page E1-388
BCOMP	Slave	CMO_On_Write, Persist_CMO, MTE_Support	Response flag that indicates that a write is observable. See *PCMO response on the B channel* on page D7-275 and *MTE signaling* on page E1-388.
BPERSIST	Slave	CMO_On_Write, Persist_CMO	Response flag that indicates that write data is updated in persistent memory. See *PCMO response on the B channel* on page D7-275.

Read address channel

Table F3-5 shows the additional read address channel signals.

Table F3-5 Read address channel signals

Signal	Source	Property	Description
ARVMIDEXT	Master	DVM_v8.1	VMID extension for a read address. See *DVM message versions* on page D13-323.
ARTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
ARLOOP	Master	Loopback_Signals	Loopback value for a read transaction. See *User Loopback signaling* on page E1-359.
ARNSAID	Master	NSAccess_Identifiers	Non-secure Access Identifier for a read transaction. See *Non-secure access identifiers* on page E1-376.
ARMPAM	Master	MPAM_Support	Read address channel MPAM information. See *Memory Partitioning and Monitoring (MPAM)* on page E1-385

Table F3-5 Read address channel signals (continued)

Signal	Source	Property	Description
ARCHUNKEN	Master	Read_Data_Chunking	Read data chunking enable. See *Read data chunking* on page E1-378.
ARIDUNQ	Master	Unique_ID_Support	Read address channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
ARTAGOP	Master	MTE_Support	Read request tag operation. See *MTE signaling* on page E1-388

Read data channel

Table F3-6 shows the additional read data channel signals. Parity check signals are not included in this table.

Table F3-6 Read data channel signals

Signal	Source	Property	Description
RPOISON	Interconnect	Poison	Indicates that the read data in this transfer has been corrupted. See Chapter E2 *Interface and data protection*.
RTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.
RLOOP	Interconnect	Loopback_Signals	Loopback value for a read response. See *User Loopback signaling* on page E1-359.
RIDUNQ	Slave	Unique_ID_Support	Read data channel unique ID indicator, active HIGH. See *Unique ID indicator* on page E1-383.
RCHUNKV	Slave	Read_Data_Chunking	Valid signal of **RCHUNKNUM** and **RCHUNKSTRB**. See *Read data chunking* on page E1-378.
RCHUNKNUM	Slave	Read_Data_Chunking	Read data chunk number. See *Read data chunking* on page E1-378.
RCHUNKSTRB	Slave	Read_Data_Chunking	Read data chunk strobe. See *Read data chunking* on page E1-378.
RTAG	Slave	MTE_Support	The tag that is associated with read data. See *MTE signaling* on page E1-388

F3.2.2 Additional channels

Two additional snoop channels are required on the ACE5-LiteDVM interface to support DVM message transfers. See Chapter D13 *Distributed Virtual Memory Transactions*.

Snoop address channel

Table F3-7 shows the additional signals on the snoop address channel.

Table F3-7 Snoop address channel signals

Signal	Source	Property	Description
ACVMIDEXT	Interconnect	DVM_v8.1	VMID extension for a snoop address. See *DVM message versions* on page D13-323.
ACTRACE	Interconnect	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.

Snoop response channel

Table F3-8 shows the additional signal on the snoop response channel.

Table F3-8 Snoop response channel signal

Signal	Source	Property	Description
CRTRACE	Master	Trace_Signals	Supports the tracing of specific transactions through the system. See *Trace signals* on page E1-357.

F3.2.3 Additional signaling

Table F3-9 shows ancillary signaling required on the ACE5-LiteDVM interface to support the new capabilities.

Table F3-9 QoS accept signals

Signal	Source	Property	Description
AWAKEUP	Master	Wakeup_Signals	Indicates that activity is initiated on the write or read address channels. See *Wake-up Signaling* on page E1-362.
ACWAKEUP	Interconnect	Wakeup_Signals	Indicates that activity is initiated on the snoop address channels. See *Wake-up Signaling* on page E1-362.
VAWQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for write transactions. See *QoS Accept signaling* on page E1-360.
VARQOSACCEPT	Slave	QoS_Accept	QoS acceptance level for read transactions. See *QoS Accept signaling* on page E1-360.
SYSCOREQ	Master	Coherency_Connection_Signals	Coherency connect request. See *Coherency Connection signaling* on page E1-364
SYSCOACK	Interconnect	Coherency_Connection_Signals	Coherency connect acknowledge. See *Coherency Connection signaling* on page E1-364

Chapter F4
ACE5-LiteACP

This chapter describes ACE5-LiteACP interface and associated protocol. It contains the following sections:

F4.1 Definition of ACE5-LiteACP

ACE5-LiteACP, which is a subset of ACE5-Lite, is intended for tightly coupling accelerator components to a processor cluster. The interface is optimized for coherent cache line accesses and is less complex than an ACE5-Lite interface. This simpler protocol enables high frequency, low latency implementations in this performance critical application.

The ACE5-LiteACP interface supersedes the AMBA4 Accelerator Coherency Port (ACP) defined in ARM IHI 0022E.

Table F4-1 shows the differences between ACE5-Lite and ACE5-LiteACP interfaces. Any ACE5-Lite features which are not mentioned in this specification are unchanged in ACE5-LiteACP. Table G2-1 on page G2-460 shows the required and optional signals for an ACE-LiteACP interface.

Table F4-1 Differences between ACE5-Lite and ACE5-LiteACP

	ACE5-Lite	ACE5-LiteACP
Data width	Up to 1024 bits	128 bits
Transaction length	Up to 256 beats	1 or 4 beats
Transaction size	Up to data bus width	128 bits
Write strobes	Any	Any
AxBURST	Any	INCR only
AxCACHE	Any	Write-Back only: • **AxCACHE**[1:0] is `0b11` • **AxCACHE**[3:2] must not be `0b00`
AxDOMAIN	Any	`0b00` Non-shareable
		`0b10` Outer Shareable
ARSNOOP	`0b0000` ReadNoSnoop / ReadOnce	`0b0000` ReadNoSnoop / ReadOnce
	`0b1000` CleanShared	
	`0b1001` CleanInvalid	
	`0b1101` MakeInvalid	
AWSNOOP	`0b0000` WriteNoSnoop / WriteUniquePtl	`0b0000` WriteNoSnoop / WriteUniquePtl
	`0b0001` WriteUniqueFull	`0b0001` WriteUniqueFull
	`0b1000` WriteUniquePtlStash	`0b1000` WriteUniquePtlStash
	`0b1001` WriteUniqueFullStash	`0b1001` WriteUniqueFullStash
	`0b1100` StashOnceShared	`0b1100` StashOnceShared
	`0b1101` StashOnceUnique	`0b1101` StashOnceUnique
	`0b1110` StashTranslation	
AxQOS	Supported	Not supported
AxREGION	Supported	Not supported
Exclusive accesses	Supported	Not supported

F4.2 Optional Extensions

ACE5-LiteACP has a restricted number of AMBA 5 optional extensions, Table 2 shows which properties are permitted to be True for ACE5-Lite and ACE5-LiteACP.

Table F4-2 Property options for ACE5-Lite and ACE5-LiteACP

Property	ACE5-Lite	ACE5-LiteACP
Cache_Stash_Transactions	Y	Y
Wakeup_Signals	Y	Y
Check_Type	Y	Y
Poison	Y	Y
Trace_Signals	Y	Y
QoS_Accept	Y	-
Loopback_Signals	Y	-
Untranslated_Transactions	Y	-
NSAccess_Identifiers	Y	-
Persist_CMO	Y	-
Atomic_Transactions	Y	-
DeAllocation_Transactions	Y	-
Unique_ID_Support	Y	Y
Ordered_Write_Observation	Y	Y
CMO_On_Read	Y	-
CMO_On_Write	Y	-
MPAM_Support	Y	Y
Read_Interleaving_Disabled	Y	Y
Read_Data_Chunking	Y	Y
Multi-Copy_Atomicity	Y	Y
Write_Plus_CMO	Y	-
MTE_Support	Y	-
Prefetch_Transaction	Y	-
WriteZero_Transaction	Y	-
Regular_Transactions_Only	Y	-
Exclusive_Accesses	Y	-
Sharable_Transactions	Y	Y
Consistent_DECERR	Y	Y
Max_Transaction_Bytes	Y	Y

F4.3 Interoperability

This section describes the interoperability of ACE5-Lite, ACE5-LiteACP, and ACP masters and slaves.

Note

The ACP interface was defined in ARM IHI 0022E and is superseded with ACE5-LiteACP in this specification.

Table F4-3 Interoperability of ACE5-Lite, ACE5-LiteACP, and ACP masters and slaves

Master	Slave	Interoperability
ACE5-Lite	ACE5-LiteACP	Can connect directly if master uses ACE5-LiteACP subset of transactions and optional features.
ACE5-LiteACP	ACE5-Lite	Fully operational. Tie off unused inputs according to Table 2.
ACP	ACE5-LiteACP	Fully operational. Set optional properties on ACE5-LiteACP to False.
ACE5-LiteACP	ACP	Fully operational if master does not issue 64-byte write bursts with sparse strobes. Set optional properties on ACE5-LiteACP to False.

When connecting an ACE5-LiteACP master directly to an ACE5-Lite slave interface, undriven inputs on the ACE5-Lite slave interface must be tied according to Table F4-4.

Table F4-4 Tie-offs for undriven signals in ACE5-LiteACP

Signal	Tie-off	Meaning
AxSIZE	`0b100`	128 bit
AxBURST	`0b00`	INCR
AxLOCK	`0b0`	Normal access
AxQOS	`0b0000`	-
AxREGION	`0b0000`	-

F4.4 ACE5-LiteACP signal list

Table F4-5 lists the signals available on each channel with ACE5-LiteACP. Check signals are not included in this table.

Table F4-5 ACE5-LiteACP signals

Global	Write address channel	Write data channel	Write response channel	Read address channel	Read data channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
AWAKEUP[a]	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWID	WSTRB	BID	ARID	RID
-	AWLEN	WLAST	-	ARLEN	RLAST
-	AWCACHE	-	-	ARCACHE	RRESP
-	AWPROT	-	-	ARPROT	-
-	AWDOMAIN	-	-	AWDOMAIN	-
-	AWSNOOP	-	-	ARSNOOP	-
-	AWTRACE[a]	WTRACE[a]	BTRACE[a]	ARTRACE[a]	RTRACE[a]
-	-	WPOISON[a]	-	-	RPOISON[a]
-	AWSTAHNID[a]	-	-	-	-
-	AWSTASHIDEN[a]	-	-	-	-
-	AWSTASHLPID[a]	-	-	-	-
-	AWSTASHLPIDEN[a]	-	-	-	-
-	AWMPAM[a]	-	-	ARMPAM[a]	-
-	AWIDUNQ[a]	-	BIDUNQ[a]	ARIDUNQ[a]	RIDUNQ[a]
-	-	-	-	ARCHUNKEN[a]	RCHUNKV[a]
-	-	-	-	-	RCHUNKNUM[a]
-	-	-	-	-	RCHUNKSTRB[a]

a. These signals are optional. See Table G2-1 on page G2-460

Chapter F5
Changes in ACE5 and ACE5-Lite

This chapter describes the changes in AMBA 5 to the ACE and ACE-Lite channel signaling requirements. It contains the following sections:

F5.1 Shareability domain support

To simplify the specification and clarify the expected use of domains, from Issue F onward, the use of the Inner Shareable domain is deprecated in the following interfaces:

- ACE5
- ACE5-Lite
- ACE5-LiteDVM
- ACE5-LiteACP

This specification recommends that new designs use the Outer Shareable domain for all transactions that would previously have been indicated as Inner Shareable.

Table F5-1 shows the updated definitions.

Table F5-1 ACE5 and ACE5-Lite shareability domain encoding

AxDOMAIN	Description	Note
0b00	Non-shareable	No change
0b01	Inner Shareable	Deprecated, use 0b10
0b10	Outer Shareable	No change
0b11	System Shareable	No change

F5.2 Barrier transaction support

From Issue F onward, barrier transactions are not supported in ACE5 and ACE5-Lite variant interfaces. ACE5 and ACE5-Lite masters that require specific ordering or observability must delay the issue of dependent requests until earlier transactions are complete.

The interface property Barrier_Transactions is used to indicate whether a component supports barrier transactions:

True The interface has the **AxBAR** signals and barrier transactions are supported. If Barrier_Transactions is not declared, it is considered True.

False The interface does not have **AxBAR** signals and barrier transactions are not supported.

The default for the Barrier_Transactions property is True, because legacy ACE and ACE-Lite components that support barrier transactions might not have the Barrier_Transactions property defined.

Table F5-2 shows interoperability and indicates that special consideration must be given when connecting an ACE or ACE-Lite master to an ACE5 or ACE5-Lite slave.

Table F5-2 Barrier Compatibility

Master	Slave	Barrier Compatibility
ACE5 or ACE5-Lite	ACE5 or ACE5-Lite	Fully compatible.
ACE5 or ACE5-Lite	ACE or ACE-Lite	Fully compatible, tie off **AxBAR** slave inputs to `0b00`.
ACE or ACE-Lite	ACE5 or ACE5-Lite	Compatible if Barrier_Transactions property is False or can be configured to be False. **AxBAR** master outputs can be left unconnected. Needs a bridging component if Barrier_Transactions is True for master.

F5.3 AWSNOOP signal width

From Issue G onward for ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACP interfaces, **AWSNOOP** is extended to be 4 bits wide, to accommodate additional operations that are permitted on those interfaces.

When connecting a master with a 3-bit **AWSNOOP** output to a slave interface with a 4-bit **AWSNOOP** input, **AWSNOOP[3]** must be tied LOW.

When connecting a master with a 4-bit **AWSNOOP** output to a slave interface with a 3-bit **AWSNOOP** input, **AWSNOOP[3]** can be left unconnected. The master must not use any cache stash transactions.

Part G

Appendices

Appendix G1
Transaction Naming

This appendix defines the naming scheme that this specification recommends for full cache line and partial cache line write transactions. It contains the following section:

G1.1 Full and partial cache line write transaction naming

A more consistent naming terminology for write transactions is introduced, to differentiate between full cache line and partial cache line transactions:

- Any transaction that is a full cache line write with all byte strobes asserted is identified by the name suffix *Full*.
- Any transaction that is a partial cache line write that is not guaranteed to have all byte strobes asserted is identified by the name suffix *Ptl*.

It is permitted for a transaction that is indicated as being a partial cache line write to be a full cache line write.

The name without a suffix, or using a * suffix, is used in any description that covers both the full and partial line variant of the transaction.

Table G1-1 shows the augmented naming.

Table G1-1 Augmented naming for write transactions

Generic name	Full cache line variant	Partial cache line variant	Notes
WriteUnique	WriteUniqueFull	WriteUniquePtl	-
WriteBack	WriteBackFull	WriteBackPtl	-
WriteClean	WriteCleanFull	WriteCleanPtl	-
WriteEvict	WriteEvictFull	-	There is no partial line variant for WriteEvict

Adoption of the new naming scheme is optional and context always permits the naming scheme in use to be determined.

Note

ACE does not provide an address phase indication that a WriteBack or WriteClean transaction is a full or partial line write.

Appendix G2
Signal Lists

Signals for each of the AMBA 5 interfaces are defined in a table with an indication of whether they are mandatory, optional or configurable.

This Appendix contains:

G2.1 Signal matrix

AMBA 5 does not require a component to use the full set of signals available on an interface. To assist in the connection of components that do not use every signal, Table G2-2 on page G2-461 defines which signals are required, and which signals are optional in AMBA 5.

If an interface does not include a **VALID** and **READY** signal for a particular channel, then no other signals on that channel can be present.

Table G2-1 lists the codes that are used in Table G2-2 on page G2-461.

Table G2-1 Key for Signal Matrix

Code	Meaning
M	Source is the Master
S	Source is the Slave
I	Source is Interconnect
V	Value is configurable
Y	Mandatory for inputs and outputs
N	Must not be present
O	Optional for inputs and outputs
OO	Optional for output ports, mandatory for inputs
OI	Optional for input ports, mandatory for outputs
C	Conditional, must be present if property is True
OC	Optional conditional, optional but can only be present if property is True
OM	Optional for master interfaces, not present on slave interfaces

Table G2-2 Signal matrix

Signal	Width	Source	Default	Property	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
ACLK	1	-	-	-	Y	Y	Y	Y	Y	Y
ARESETn	1	-	-	-	Y	Y	Y	Y	Y	Y
AWVALID	1	M	-	-	Y	Y	Y	Y	Y	Y
AWREADY	1	S	-	-	Y	Y	Y	Y	Y	Y
AWID	V	M	All zeros	-	OO	OO	OO	OO	OO	OO
AWADDR	V	M	-	-	Y	Y	Y	Y	Y	Y
AWREGION	4	M	`0b0000`	-	O	O	O	N	O	N
AWLEN	8	M	`0x00`	-	OO	OO	OO	OO	OO	N
AWSIZE	3	M	Data bus width	-	OO	OO	OO	N	OO	O
AWBURST	2	M	`0b01`, INCR	-	OO	OO	OO	N	OO	N
AWLOCK	1	M	`0b0`, normal access	Exclusive_Accesses	O	O	O	N	O	N
AWCACHE	4	M	`0b0000`	-	O	O	O	O	O	N
AWPROT	3	M	-	-	OI	OI	OI	OI	OI	OI
AWQOS	4	M	`0b0000`	-	O	O	O	N	O	N
AWUSER	V	M	All zeros	-	O	O	O	O	O	O
AWDOMAIN	2	M	-	-	Y	Y	Y	Y	N	N
AWSNOOP	4	M	`0b0000`	-	OO	OO	OO	OO	N	N
AWBAR	2	M	`0b00`	Barrier_Transactions	N	N	N	N	N	N
AWUNIQUE	1	M	`0b0`	WriteEvict_Transaction	C	N	N	N	N	N
AWSTASHNID	11	M	`0x0000`	Cache_Stash_Transactions	N	OC	OC	OC	N	N
AWSTASHNIDEN	1	M	`0b0`	Cache_Stash_Transactions	N	OC	OC	OC	N	N
AWSTASHLPID	5	M	`0b00000`	Cache_Stash_Transactions	N	OC	OC	OC	N	N
AWSTASHLPIDEN	1	M	`0b0`	Cache_Stash_Transactions	N	OC	OC	OC	N	N
AWTRACE	1	M	`0b0`	Trace_Signals	C	C	C	C	C	C
AWLOOP	V	M	All zeros	Loopback_Signals	C	C	C	N	C	N
AWMMUSECSID	1	M	`0b0`	Untranslated_Transactions	OC	N	OC	N	OC	N
AWMMUSID	V	M	All zeros	Untranslated_Transactions	OC	N	OC	N	OC	N

Table G2-2 Signal matrix (continued)

Signal	Width	Source	Default	Property	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
AWMMUSSIDV	1	M	`0b0`	Untranslated_Transactions	OC	N	OC	N	OC	N
AWMMUSSID	V	M	All zeros	Untranslated_Transactions	OC	N	OC	N	OC	N
AWMMUATST	1	M	`0b0`	Untranslated_Transactions	OC	N	OC	N	OC	N
AWMMUFLOW	2	M	`0b00`	Untranslated_Transactions	N	N	OC	N	OC	N
AWNSAID	4	M	`0x0`	NSAccess_Identifiers	C	C	C	N	C	N
AWATOP	6	M	`0b000000`	Atomic_Transactions	N	C	C	N	C	N
AWMPAM	11	M	**AWPROT**[1]	MPAM_Support	C	C	C	C	C	N
AWIDUNQ	1	M	`0b0`	Unique_ID_Support	C	C	C	C	C	C
AWCMO	2	M	`0b0`	CMO_On_Write	N	C	C	N	N	N
AWTAGOP	2	M	-	MTE_Support	N	C	C	N	C	N
WVALID	1	M	-	-	Y	Y	Y	Y	Y	Y
WREADY	1	S	-	-	Y	Y	Y	Y	Y	Y
WDATA	V	M	-	-	Y	Y	Y	Y	Y	Y
WSTRB	V	M	All ones	-	OO	OO	OO	OO	OO	OO
WLAST	1	M	-	-	OI	OI	OI	OI	OI	N
WUSER	V	M	All zeros	-	O	O	O	O	O	O
WPOISON	V	M	-	Poison	C	C	C	C	C	C
WTRACE	1	M	`0b0`	Trace_Signals	C	C	C	C	C	C
WTAG	V	M	-	MTE_Support	N	C	C	N	C	N
WTAGUPDATE	V	M	-	MTE_Support	N	C	C	N	C	N
BVALID	1	S	-	-	Y	Y	Y	Y	Y	Y
BREADY	1	M	-	-	Y	Y	Y	Y	Y	Y
BID	V	S	-	-	OI	OI	OI	OI	OI	OI
BRESP	V	S	OKAY	Untranslated _Transactions	O	O	O	O	O	O
BUSER	V	S	All zeros	-	O	O	O	O	O	O
BTRACE	1	S	`0b0`	Trace_Signals	C	C	C	C	C	C
BLOOP	V	S	All zeros	Loopback_Signals	C	C	C	N	C	N
BIDUNQ	1	S	`0b0`	Unique_ID_Support	C	C	C	C	C	C
BCOMP	1	S	`0b0`	CMO_On_Write, Persist_CMO, MTE_Support	N	C	C	N	C	N

Table G2-2 Signal matrix (continued)

Signal	Width	Source	Default	Property	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
BPERSIST	1	S	0b0	CMO_On_Write, Persist_CMO	N	C	C	N	N	N
BTAGMATCH	2	S	-	MTE_Support	N	N	C	N	C	N
ARVALID	1	M	-	-	Y	Y	Y	Y	Y	Y
ARREADY	1	S	-	-	Y	Y	Y	Y	Y	Y
ARID	V	M	All zeros	-	OO	OO	OO	OO	OO	OO
ARADDR	V	M	-	-	Y	Y	Y	Y	Y	Y
ARREGION	4	M	0b0000	-	O	O	O	N	O	N
ARLEN	8	M	0x00	-	OO	OO	OO	OO	OO	N
ARSIZE	3	M	Data bus width	-	OO	OO	OO	N	OO	O
ARBURST	2	M	0b01, INCR	-	OO	OO	OO	N	OO	N
ARLOCK	1	M	0b0, normal access	Exclusive_Accesses	O	O	O	N	O	N
ARCACHE	4	M	0b0000	-	O	O	O	O	O	N
ARPROT	3	M	-	-	OI	OI	OI	OI	OI	OI
ARQOS	4	M	0b0000	-	O	O	O	N	O	N
ARUSER	V	M	All zeros	-	O	O	O	O	O	O
ARDOMAIN	2	M	-	-	Y	Y	Y	Y	N	N
ARSNOOP	4	M	0x0	-	OO	OO	OO	O	N	N
ARBAR	2	M	0b00	Barrier_Transactions	N	N	N	N	N	N
ARVMIDEXT	4	M	0b0000	DVM_Message_Support, DVM_v8.1	OC	OC	N	N	N	N
ARTRACE	1	M	-	Trace_Signals	C	C	C	C	C	C
ARLOOP	V	M	All zeros	Loopback_Signals	C	C	C	N	C	N
ARMMUSECSID	1	M	0b0	Untranslated_Transactions	OC	N	OC	N	OC	N
ARMMUSID	V	M	All zeros	Untranslated_Transactions	OC	N	OC	N	OC	N
ARMMUSSIDV	1	M	0b0	Untranslated_Transactions	OC	N	OC	N	OC	N
ARMMUSSID	V	M	All zeros	Untranslated_Transactions	OC	N	OC	N	OC	N
ARMMUATST	1	M	0b0	Untranslated_Transactions	OC	N	OC	N	OC	N
ARMMUFLOW	2	M	0b00	Untranslated_Transactions	N	N	OC	N	OC	N
ARNSAID	4	M	0x0	NSAccess_Identifiers	C	C	C	N	C	N

Table G2-2 Signal matrix (continued)

Signal	Width	Source	Default	Property	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
ARMPAM	11	M	**ARPROT**[1]	MPAM_Support	C	C	C	C	C	N
ARCHUNKEN	1	M	`0b0`	Read_Data_Chunking	N	C	C	C	C	N
ARIDUNQ	1	M	`0b0`	Unique_ID_Support	C	C	C	C	C	C
ARTAGOP	2	M	-	MTE_Support	N	C	C	N	C	N
RVALID	1	S	-	-	Y	Y	Y	Y	Y	Y
RREADY	1	M	-	-	Y	Y	Y	Y	Y	Y
RID	V	S	-	-	OI	OI	OI	OI	OI	OI
RDATA	V	S	-	-	Y	Y	Y	Y	Y	Y
RRESP	V	S	OKAY	Prefetch_Transaction, Untranslated_Transactions	Y	O	O	O	O	O
RLAST	1	S	-	-	OI	OI	OI	OI	OI	N
RUSER	V	S	All zeros	-	O	O	O	O	O	O
RPOISON	V	S	-	Poison	C	C	C	C	C	C
RTRACE	1	S	`0b0`	Trace_Signals	C	C	C	C	C	C
RLOOP	V	S	All zeros	Loopback_Signals	C	C	C	N	C	N
RIDUNQ	1	S	`0b0`	Unique_ID_Support	C	C	C	C	C	C
RCHUNKV	1	S	`0b0`	Read_Data_Chunking	N	C	C	C	C	N
RCHUNKNUM	V	S	All zeros	Read_Data_Chunking	N	C	C	C	C	N
RCHUNKSTRB	V	S	All zeros	Read_Data_Chunking	N	C	C	C	C	N
RTAG	V	S	-	MTE_Support	N	C	C	N	C	N
ACVALID	1	I	-	-	Y	Y	N	N	N	N
ACREADY	1	M	-	-	Y	Y	N	N	N	N
ACADDR	V	I	-	-	Y	Y	N	N	N	N
ACSNOOP	4	I	`0b1111`, DVM message	-	Y	OO	N	N	N	N
ACPROT	3	I	`0b000`	-	Y	O	N	N	N	N
ACVMIDEXT	4	I	-	DVM_Message_Support, DVM_v8.1	C	C	N	N	N	N
ACTRACE	1	I	-	Trace_Signals	C	C	N	N	N	N
CRVALID	1	M	-	-	Y	Y	N	N	N	N
CRREADY	1	I	-	-	Y	Y	N	N	N	N

Table G2-2 Signal matrix (continued)

Signal	Width	Source	Default	Property	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
CRRESP	5	M	0b00000	-	Y	O	N	N	N	N
CRTRACE	1	M	-	Trace_Signals	C	C	N	N	N	N
CRNSAID	4	M	-	NSAccess_Identifiers	OC	N	N	N	N	N
CDVALID	1	M	0b0	-	O	N	N	N	N	N
CDREADY	1	I	0b1	-	O	N	N	N	N	N
CDDATA	V	M	All zeros	-	O	N	N	N	N	N
CDLAST	1	M	0b0	-	O	N	N	N	N	N
CDPOISON	V	M	-	Poison	OC	N	N	N	N	N
CDTRACE	1	M	-	Trace_Signals	OC	N	N	N	N	N
RACK	1	M	-	-	Y	N	N	N	N	N
WACK	1	M	-	-	Y	N	N	N	N	N
VAWQOSACCEPT	4	S	-	QoS_Accept	C	C	C	N	C	N
VARQOSACCEPT	4	S	-	QoS_Accept	C	C	C	N	C	N
AWAKEUP	1	M	-	Wakeup_Signals	C	C	C	C	C	C
ACWAKEUP	1	I	-	Wakeup_Signals	C	C	N	N	N	N
SYSCOREQ	1	M	-	Coherency_Connection_Signals	C	C	N	N	N	N
SYSCOACK	1	I	-	Coherency_Connection_Signals	C	C	N	N	N	N
BROADCASTATOMIC	1	-	-	-	N	OM	OM	N	OM	N
BROADCASTINNER	1	-	-	-	OM	N	N	N	N	N
BROADCASTOUTER	1	-	-	-	OM	N	N	N	N	N
BROADCASTCACHEMAINT	1	-	-	-	OM	N	N	N	N	N

G2.2 Check signal matrix

Table G2-3 shows the protection signals that can be present on an interface, based on the value of the property Check_Type.

Table G2-3 Signal matrix

Signal	Width	Source	Default	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
AWVALIDCHK	1	M	-	C	C	C	C	C	C
AWREADYCHK	1	S	-	C	C	C	C	C	C
AWIDCHK	V	M	`0b1`	OC	OC	OC	OC	OC	OC
AWADDRCHK	V	M	-	C	C	C	C	C	C
AWLENCHK	1	M	`0b1`	OC	OC	OC	OC	OC	N
AWCTLCHK0	1	M	`0b0`	OC	OC	OC	OC	OC	OC
AWCTLCHK1	1	M	`0b1`	OC	OC	OC	OC	OC	N
AWCTLCHK2	1	M	`0b0`	C	C	C	C	N	N
AWCTLCHK3	1	M	`0b0`	N	OC	OC	N	OC	N
AWUSERCHK	V	M	All ones	OC	OC	OC	OC	OC	OC
AWSTASHNIDCHK	1	M	`0b1`	N	OC	OC	OC	N	N
AWSTASHLPIDCHK	1	M	`0b1`	N	OC	OC	OC	N	N
AWTRACECHK	1	M	`0b1`	C	C	C	C	C	C
AWLOOPCHK	1	M	`0b1`	C	C	C	N	C	N
AWMMUCHK	1	M	`0b1`	OC	N	OC	N	OC	N
AWMMUSIDCHK	V	M	All ones	OC	N	OC	N	OC	N
AWMMUSSIDCHK	V	M	All ones	OC	N	OC	N	OC	N
AWNSAIDCHK	1	M	`0b1`	C	C	C	N	C	N
AWMPAMCHK	1	M	-	C	C	C	C	C	N
WVALIDCHK	1	M	-	C	C	C	C	C	C
WREADYCHK	1	S	-	C	C	C	C	C	C
WDATACHK	V	M	-	C	C	C	C	C	C
WSTRBCHK	V	M	All zeros	OC	OC	OC	OC	OC	OC
WLASTCHK	1	M	-	OC	OC	OC	OC	OC	N
WUSERCHK	V	M	All ones	C	C	C	C	C	C
WPOISONCHK	V	M	-	C	C	C	C	C	C
WTRACECHK	1	M	`0b1`	C	C	C	C	C	C

Table G2-3 Signal matrix (continued)

Signal	Width	Source	Default	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
WTAGCHK	V	M	-	N	C	C	N	C	N
BVALIDCHK	1	S	-	C	C	C	C	C	C
BREADYCHK	1	M	-	C	C	C	C	C	C
BIDCHK	V	S	`0b1`	OC	OC	OC	OC	OC	OC
BRESPCHK	1	S	`0b1`	C	C	C	C	C	C
BUSERCHK	V	S	All ones	C	C	C	C	C	C
BTRACECHK	1	S	`0b1`	C	C	C	C	C	C
BLOOPCHK	1	S	`0b1`	C	C	C	N	C	N
ARVALIDCHK	1	M	-	C	C	C	C	C	C
ARREADYCHK	1	S	-	C	C	C	C	C	C
ARIDCHK	V	M	`0b1`	OC	OC	OC	OC	OC	OC
ARADDRCHK	V	M	-	C	C	C	C	C	C
ARLENCHK	1	M	`0b1`	OC	OC	OC	OC	OC	N
ARCTLCHK0	1	M	-	OC	OC	OC	OC	OC	OC
ARCTLCHK1	1	M	`0b1`	OC	OC	OC	OC	OC	N
ARCTLCHK2	1	M	-	C	C	C	C	N	N
ARCTLCHK3	1	M	-	C	C	C	C	C	N
ARUSERCHK	V	M	All ones	OC	OC	OC	OC	OC	OC
ARTRACECHK	1	M	`0b1`	C	C	C	C	C	C
ARLOOPCHK	1	M	`0b1`	C	C	C	N	C	N
ARMMUCHK	1	M	`0b1`	OC	N	OC	N	OC	N
ARMMUSIDCHK	V	M	All ones	OC	N	OC	N	OC	N
ARMMUSSIDCHK	V	M	All ones	OC	N	OC	N	OC	N
ARNSAIDCHK	1	M	`0b1`	C	C	C	N	C	N
ARMPAMCHK	1	M	-	C	C	C	C	C	N
RVALIDCHK	1	S	-	C	C	C	C	C	C
RREADYCHK	1	M	-	C	C	C	C	C	C
RIDCHK	V	S	`0b1`	C	C	C	C	C	C
RDATACHK	V	S	-	C	C	C	C	C	C
RRESPCHK	1	S	`0b1`	C	OC	OC	OC	OC	OC

Table G2-3 Signal matrix (continued)

Signal	Width	Source	Default	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
RLASTCHK	1	S	-	C	C	C	C	C	N
RCHUNKCHK	1	S	`0b1`	N	C	C	C	C	N
RUSERCHK	V	S	All ones	OC	OC	OC	OC	OC	OC
RPOISONCHK	V	S	-	C	C	C	C	C	C
RTRACECHK	1	S	`0b1`	C	C	C	C	C	C
RLOOPCHK	1	S	`0b1`	C	C	C	N	C	N
RTAGCHK	V	S	-	N	C	C	N	C	N
ACVALIDCHK	1	I	-	C	C	N	N	N	N
ACREADYCHK	1	M	-	C	C	N	N	N	N
ACADDRCHK	V	I	-	C	C	N	N	N	N
ACCTLCHK	1	I	`0b1`	C	OC	N	N	N	N
ACVMIDEXTCHK	1	I	-	C	C	N	N	N	N
ACTRACECHK	1	I	-	C	C	N	N	N	N
CRVALIDCHK	1	M	-	C	C	N	N	N	N
CRREADYCHK	1	I	-	C	C	N	N	N	N
CRRESPCHK	1	M	`0b1`	C	OC	N	N	N	N
CRTRACECHK	1	M	-	C	C	N	N	N	N
CRNSAIDCHK	1	M	-	OC	N	N	N	N	N
CDVALIDCHK	1	M	`0b1`	OC	N	N	N	N	N
CDREADYCHK	1	I	`0b0`	OC	N	N	N	N	N
CDDATACHK	V	M	All ones	OC	N	N	N	N	N
CDLASTCHK	1	M	`0b1`	OC	N	N	N	N	N
CDPOISONCHK	V	M	-	OC	N	N	N	N	N
CDTRACECHK	1	M	-	OC	N	N	N	N	N
RACKCHK	1	M	-	C	N	N	N	N	N
WACKCHK	1	M	-	C	N	N	N	N	N
VAWQOSACCEPTCHK	1	S	-	C	C	C	N	C	N
VARQOSACCEPTCHK	1	S	-	C	C	C	N	C	N
AWAKEUPCHK	1	M	-	C	C	C	C	C	C

Table G2-3 Signal matrix (continued)

Signal	Width	Source	Default	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
ACWAKEUPCHK	1	S	-	C	C	N	N	N	N
SYSCOREQCHK	1	M	-	C	C	N	N	N	N
SYSCOACKCHK	1	I	-	C	C	N	N	N	N

Appendix G3
Interface Property Summary

G3.1 Summary of interface properties

If an entry is not "Y", then the property must be False or undeclared for that interface type.

Table G3-1 Summary of interface properties

Property	Issue introduced	ACE5	ACE5- LiteDVM	ACE5- Lite	ACE5- LiteACP	AXI5	AXI5- Lite	ACE	ACE-Lite	AXI4	AXI4-Lite	AXI3
Continuous_Cache_Line_Read_Data	E	Y	-	-	-	-	-	Y	-	-	-	-
DVM_v8	E	Y	Y	-	-	-	-	Y	-	-	-	-
Multi_Copy_Atomicity	E	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Ordered_Write_Observation	E	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteEvict_Transaction	E	Y	-	-	-	-	-	Y	-	-	-	-
Atomic_Transactions	F	-	Y	Y	-	Y	-	-	-	-	-	-
Barrier_Transactions	F	-	-	-	-	-	-	Y	Y	-	-	-
Cache_Stash_Transactions	F	-	Y	Y	Y	-	-	-	-	-	-	-
Check_Type	F	Y	Y	Y	Y	Y	Y	-	-	-	-	-
Coherency_Connection_Signals	F	Y	Y	-	-	-	-	-	-	-	-	-
DeAllocation_Transactions	F	-	Y	Y	-	-	-	-	-	-	-	-
DVM_v8.1	F	Y	Y	-	-	-	-	Y	-	-	-	-
Loopback_Signals	F	Y	Y	Y	-	Y	-	-	-	-	-	-
NSAccess_Identifiers	F	Y	Y	Y	-	Y	-	-	-	-	-	-
Persist_CMO	F	Y	Y	Y	-	-	-	-	-	-	-	-
Poison	F	Y	Y	Y	Y	Y	Y	-	-	-	-	-
QoS_Accept	F	Y	Y	Y	-	Y	-	-	-	-	-	-
Trace_Signals	F	Y	Y	Y	Y	Y	Y	-	-	-	-	-
Untranslated_Transactions	F	Y[a]	-	Y	-	Y	-	-	-	-	-	-
Wakeup_Signals	F	Y	Y	Y	Y	Y	Y	-	-	-	-	-
CMO_On_Read	G	Y	Y	Y	-	-	-	-	-	-	-	-
CMO_On_Write	G	-	Y	Y	-	-	-	-	-	-	-	-
MPAM_Support	G	Y	Y	Y	Y	Y	-	-	-	-	-	-
Read_Data_Chunking	G	-	Y	Y	Y	Y	-	-	-	-	-	-
Read_Interleaving_Disabled	G	-	Y	Y	Y	Y	-	-	-	-	-	-
Unique_ID_Support	G	Y	Y	Y	Y	Y	Y	-	-	-	-	-

Table G3-1 Summary of interface properties (continued)

Property	Issue introduced	ACE5	ACE5- LiteDVM	ACE5- Lite	ACE5- LiteACP	AXI5	AXI5- Lite	ACE	ACE-Lite	AXI4	AXI4-Lite	AXI3
ADDR_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Consistent_DECERR	H	Y	Y	Y	Y	Y	-	Y	Y	Y	-	Y
DATA_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
DVM_Message_Support	H	Y	Y	-	-	-	-	Y	-	-	-	-
DVM_v8.4	H	-	Y	-	-	-	-	-	-	-	-	-
Exclusive_Accesses	H	Y	Y	Y	-	Y	-	Y	Y	Y	-	Y
ID_R_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ID_W_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
LOOP_R_WIDTH	H	Y	Y	Y	-	Y	-	-	-	-	-	-
LOOP_W_WIDTH	H	Y	Y	Y	-	Y	-	-	-	-	-	-
Max_Transaction_Bytes	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MTE_Support	H	-	Y	Y	-	Y	-	-	-	-	-	-
Prefetch_Transaction	H	-	Y	Y	-	-	-	-	-	-	-	-
Regular_Transactions_Only	H	Y	Y	Y	-	Y	-	-	-	-	-	-
Shareable_Transactions	H	Y	Y	Y	Y	-	-	Y	Y	-	-	-
SID_WIDTH	H	Y	-	Y	-	Y	-	-	-	-	-	-
SSID_WIDTH	H	Y	-	Y	-	Y	-	-	-	-	-	-
USER_DATA_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
USER_REQ_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
USER_RESP_WIDTH	H	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Write_Plus_CMO	H	-	Y	Y	-	-	-	-	-	-	-	-
WriteZero_Transaction	H	-	Y	Y	-	-	-	-	-	-	-	-

a. v2 option is not supported on ACE5 interfaces.

Appendix G4
Summary of AxSNOOP encodings

This appendix shows all possible **ARSNOOP** and **AWSNOOP** encodings and the property that is used to determine if a particular value is supported for a given interface. It contains the following sections:

- *ARSNOOP encodings* on page G4-476
- *AWSNOOP encodings* on page G4-477

Encodings for **ACSNOOP** can be found in Table D3-19 on page D3-193.

G4.1 ARSNOOP encodings

Table G4-1 shows the properties and interface types that are supported for all possible **ARSNOOP** encodings.

Table G4-1 Summary of ARSNOOP encodings

Code	Transaction type	Domain options[a]	Property	Regular transaction	Cache line sized	Cache line or smaller	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
0b0000	ReadNoSnoop	NSH, SY	-	-	-	-	Y	Y	Y	Y	Y	Y
	ReadOnce	ISH, OSH	Shareable_Transactions	-	-	-	Y	Y	Y	Y	-	-
0b0001	ReadShared	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b0010	ReadClean	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b0011	ReadNotSharedDirty	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b0100	ReadOnceCleanInvalid	ISH, OSH	Shareable_Transactions, DeAllocation_Transactions	-	-	Y	-	Y	Y	-	-	-
0b0101	ReadOnceMakeInvalid	ISH, OSH	Shareable_Transactions, DeAllocation_Transactions	-	-	Y	-	Y	Y	-	-	-
0b0110	Reserved	-	-	-	-	-	-	-	-	-	-	-
0b0111	ReadUnique	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b1000	CleanShared	NSH, ISH, OSH	CMO_On_Read	Y	Y	-	Y	Y	Y	-	-	-
0b1001	CleanInvalid	NSH, ISH, OSH	CMO_On_Read	Y	Y	-	Y	Y	Y	-	-	-
0b1010	CleanSharedPersist	NSH, ISH, OSH	CMO_On_Read, Persist_CMO	Y	Y	-	Y	Y	Y	-	-	-
0b1011	CleanUnique	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b1100	MakeUnique	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b1101	MakeInvalid	NSH, ISH, OSH	CMO_On_Read	Y	Y	-	Y	Y	Y	-	-	-
0b1110	DVM Complete	ISH, OSH	DVM_Message_Support	-	-	-	Y	Y	-	-	-	-
0b1111	DVM Message	ISH, OSH	DVM_Message_Support	-	-	-	Y	Y	-	-	-	-

a. NSH is Non-shareable, ISH is Inner Shareable, OSH is Outer Shareable, SY is System Shareable

G4.2 AWSNOOP encodings

Table G4-2 shows the properties and interface types that are supported for all possible **AWSNOOP** encodings.

Table G4-2 Summary of AWSNOOP encodings

Code	Transaction type	Domain options[a]	Property	Regular transaction	Cache line sized	Cache line or smaller	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
0b0000	WriteNoSnoop	NSH, SY	-	-	-	-	Y	Y	Y	Y	Y	Y
	WriteUniquePtl	ISH, OSH	Shareable_Transactions	-	-	-	Y	Y	Y	Y	-	-
	Atomic Transactions	NSH, ISH, OSH, SY	Atomic_Transactions	-	-	-	-	Y	Y	-	Y	-
0b0001	WriteUniqueFull	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	Y	Y	Y	-	-
0b0010	WriteClean	NSH, ISH, OSH	-	-	-	Y	Y	-	-	-	-	-
0b0011	WriteBack	NSH, ISH, OSH	-	-	-	Y	Y	-	-	-	-	-
0b0100	Evict	ISH, OSH	Shareable_Transactions	Y	Y	-	Y	-	-	-	-	-
0b0101	WriteEvict	NSH, ISH, OSH	WriteEvict_Transaction	Y	Y	-	Y	-	-	-	-	-
0b0110	CMO	NSH, ISH, OSH	CMO_On_Write	Y	Y	-	-	Y	Y	-	-	-
0b0111	WriteZero	NSH, ISH, OSH, SY	WriteZero_Transaction	Y	Y	-	-	Y	Y	-	-	-
0b1000	WriteUniquePtlStash	ISH, OSH	Shareable_Transactions, Cache_Stash_Transactions	-	-	Y	-	Y	Y	Y	-	-
0b1001	WriteUniqueFullStash	ISH, OSH	Shareable_Transactions, Cache_Stash_Transactions	Y	Y	-	-	Y	Y	Y	-	-
0b1010	WritePtlCMO	NSH, ISH, OSH	Write_Plus_CMO	-	-	Y	-	Y	Y	-	-	-
0b1011	WriteFullCMO	NSH, ISH, OSH	Write_Plus_CMO	Y	Y	-	-	Y	Y	-	-	-
0b1100	StashOnceShared	NSH, ISH, OSH	Cache_Stash_Transactions	Y	Y	-	-	Y	Y	Y	-	-

Table G4-2 Summary of AWSNOOP encodings (continued)

Code	Transaction type	Domain options[a]	Property	Regular transaction	Cache line sized	Cache line or smaller	ACE5	ACE5-LiteDVM	ACE5-Lite	ACE5-LiteACP	AXI5	AXI5-Lite
0b1101	StashOnceUnique	NSH, ISH, OSH	Cache_Stash_Transactions	Y	Y	-	-	Y	Y	Y	-	-
0b1110	StashTranslation	NSH, ISH, OSH, SY	Untranslated_Transactions, Cache_Stash_Transactions	-	-	-	-	-	Y	-	-	-
0b1111	Prefetch	NSH, ISH, OSH	Prefetch_Transaction	Y	Y	-	-	Y	Y	-	-	-

a. NSH is Non-shareable, ISH is Inner Shareable, OSH is Outer Shareable, SY is System Shareable

Appendix G5
Summary of ID constraints

The following restrictions on ID usage are specified in this document; they are summarized here for reference.

G5.1 ID constraints

Must not use the same ID for in-flight transactions:

- Barrier and non-barrier transactions
- DVM transactions and non-DVM transactions
- Atomic and Non-atomic transactions
- Multiple Atomic transactions
- StashOnce and non-StashOnce transactions
- StashTranslation and non-StashTranslation transactions
- Prefetch transactions
- WriteZero transactions
- Read transactions with data chunking enabled
- Transactions which transport MTE tags

It is recommended that the following transactions use an ID that is unique in-flight:

- WriteBack
- WriteClean
- WriteEvict
- Evict

Must use the same ID:

- Transactions in an exclusive access pair
- Transactions in a locked sequence (AXI3 only)
- Transactions in a barrier pair (ACE and ACE-Lite only)

Appendix G6
Summary of response codes

This appendix contains a summary of read and write response codes:

G6.1 Read response codes

For interfaces where the Untranslated_Transactions property is v2 or the Prefetch_Transaction property is True, RRESP is extended from 2-bits to 3-bits to enable the signaling of additional responses.

The full list of possible read response encodings is shown in Table G6-1.

Table G6-1 Read responses

RRESP[2:0]	Response	Indication
0b000	OKAY	Transaction has completed successfully. Read data is valid. Read tags are valid if they were requested.
		Exclusive read to a slave which does not support exclusive accesses.
0b001	EXOKAY	Exclusive read succeeded.
0b010	SLVERR	Slave error. The access has reached the slave successfully, but the slave wishes to return an error condition to the originating master.
0b011	DECERR	Decode error. Generated, typically by an interconnect component, to indicate that there is no slave at the transaction address.
0b100	PREFETCHED	Read data is valid and has been sourced from a prefetched value.
0b101	TRANSFAULT	Transaction was terminated because of a translation fault which might be resolved by a PRI request. Read data is not valid.
0b110	Reserved	-
0b111	Reserved	-

G6.2 Write response codes

For interfaces where the Untranslated_Transactions property is v2, **BRESP** is extended from 2-bits to 3-bits to enable the signaling of a TRANSFAULT response. The full list of possible write response encodings is shown in Table G6-2.

Table G6-2 Write responses

BRESP[2:0]	Response	Indication
0b000	OKAY	Non-exclusive access: write data can be seen by all observers.
		Exclusive access: write failed to update the location.
0b001	EXOKAY	Exclusive write succeeded.
0b010	SLVERR	Slave error. The access has reached the slave successfully, but the slave wishes to return an error condition to the originating master.
0b011	DECERR	Decode error. Generated, typically by an interconnect component, to indicate that there is no slave at the transaction address.
0b100	Reserved	-
0b101	TRANSFAULT	Transaction was terminated because of a translation fault which might be resolved by a PRI request.
0b110	Reserved	-
0b111	Reserved	-

G6.3 Write response combinations

Table G6-2 on page G6-483 shows the legal write response combinations for interfaces which include the **BCOMP** signal. Transaction types in Table G6-2 on page G6-483 are defined as:

- Tag Match is any transaction where **AWTAGOP** is `0b11` (Match).
- Persist is any transaction where **AWCMO** is `0b10` (CSP) or `0b11` (CSDP).

Table G6-3 Write response combinations

Transaction type	BCOMP	BPERSIST	BTAGMATCH	OKAY	EXOKAY	SLVERR	DECERR	TRANSFAULT	Note
Not Persist, not Tag Match	1	0	`0b00`	Y	Y	Y	Y	Y	Single beat response.
Persist	0	1	`0b00`	Y	-	Y	Y	-	Persist response. Part of 2-part response.
	1	0	`0b00`	Y	-	Y	Y	-	Completion response. Part of 2-part response.
	1	1	`0b00`	Y	-	Y	Y	Y	Combined response. Single beat response.
Tag Match	1	0	`0b01`	Y	Y	Y	Y	-	Completion response. Part of 2-part response.
	0	0	`0b10` / `0b11`	Y	-	Y	Y	-	Match response. Part of 2-part response.
	1	0	`0b10` / `0b11`	Y	Y	Y	Y	Y	Combined response. Single beat response.

Appendix G7
Revisions

This appendix describes the technical changes between released issues of this specification.

Table G7-1 Issue B

Change	Location
First release of Version 1.0	–

Table G7-2 Differences between issue B and issue C

Change	Location
Additional section describing the chapter layout of Version 2.0 of the document	AXI revisions on page A1-21
Additional details on the constraints for the **VALID** and **READY** handshake	Handshake process on page 3-2
Additional equation for wrapping bursts	Burst address on page 4-7
Additional chapter describing the AXI4 update to the AXI3 protocol	Chapter 13 AXI4
Additional chapter describing the AXI4-Lite subset of the AXI4 protocol	Chapter 14 AXI4-Lite

Table G7-3 Differences between issue C and issue D

Change	Location
Full integration of the AXI3 and AXI4 content	Part A AXI3 and AXI4 Protocol Specification
Additional Part added describing the ACE update to the AXI protocol	Part C AXI Coherency Extensions (ACE) Protocol Specification

Table G7-4 Differences between issue D and issue E

Change	Location
Clarification of the byte lane strobes' requirement for FIXED bursts	Burst type in *Address structure* on page A3-48
Correction to pseudo code routine: `//Increment address if necessary`	*Pseudocode description of the transfers* on page A3-52
Additional section describing the Ordered_Write_Observation property	*Ordered write observation* on page A6-91
Additional section describing the Multi_Copy_Atomocity property	*Multi-copy write atomicity* on page A7-95
Clarification of the peripheral slave transaction subset	*Memory slaves and peripheral slaves* on page A9-109
Additional section describing the **AWUNIQUE** signal	*AWUNIQUE signal* on page D3-181
Clarification of WriteUnique Propagation to Main Memory	*WriteUnique* on page D4-223 and *WriteLineUnique* on page D4-223
Additional section describing the WriteEvict transaction	*WriteEvict* on page D4-225

Table G7-4 Differences between issue D and issue E (continued)

Change	Location
Clarification of WriteNoSnoop blocked by WriteUnique and WriteLineUnique	*Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226
Clarification of sequencing Coherent and Cache Maintenance transactions to a cache line	*Sequencing transactions* on page D6-249
Additional section describing the Continuous_Cache_Line_Read_Data property	*Continuous read data return* on page D6-250
Clarification of Exclusive Accesses and naturally evicted cache lines	*Exclusive Store* on page D9-298
Clarification of the Shareable terminology in Exclusive Accesses	*About Exclusive accesses from ACE masters* on page D9-296 and *Transaction requirements* on page D9-304
Additional section describing optional DVM message support for Armv8	*DVM message versions* on page D13-323
Additional section describing DVMv7 and DVMv8 address spaces	*Addresses in DVM messages* on page D13-324
Additional format definitions for DVMv8 messages	*Addresses in DVM messages* on page D13-324
Additional format definitions for the TLB Invalidate message to support DVMv8	*TLB Invalidate* on page D13-330
Additional section describing DVMv7 and DVMv8 conversion	*DVM message versions* on page D13-323
Additional chapter providing a set of recommendations for the design of master components	Chapter D14 *Master Design Recommendations*
Additional appendix describing full cache line and partial cache line write transaction naming	Appendix G1 *Transaction Naming*
Additional appendix describing the ACP interface requirements	Appendix F4 *Accelerator Coherency Port Interface Restrictions*

Table G7-5 Differences between issue E and issue F

Change	Location
Removed Low Power Interface chapter, this content has been superseded by a separate specification (ARM IHI 0068C)	Was Chapter A9
New interfaces defined: AXI5, AXI5-Lite	Part C *AMBA AXI5 and AXI5-Lite Interface Specification*
Rule that a snoop response must give IsShared asserted while a Write is in progress with **AWUNIQUE** asserted is clarified to only apply to WriteBack and WriteEvict	*AWUNIQUE signal* on page D3-181
Clarification that a line might become dirty when a CleanShared is in progress	*CleanShared* on page D4-218

Table G7-5 Differences between issue E and issue F (continued)

Change	Location
Change which adds restrictions when using Evict transactions with WriteUnique and WriteLineUnique	*Restrictions on WriteUnique and WriteLineUnique usage* on page D4-226
Added a missing row to the table of Alternative Snoop Transactions to cover MakeInvalid	Table D5-2 on page D5-233
Change to allow an Evict and WriteEvict to be issued while a CleanShared is in progress	*Cache maintenance propagation* on page D7-270
Change to the mismatched shareability and cacheability rules to allow for cache maintenance transactions	Chapter D7 *Cache Maintenance*
Clarification that a component is not permitted to wait for a DVM Complete relating to a DVM Sync it has issued, before it provides DVM Complete for a DVM Sync it has received	*DVM Synchronization and DVM Complete transactions* on page D13-320
Clarification that an ACE/ACE-Lite master is permitted to wait for both parts of a 2-part DVM message before responding on CR channel	*DVM message transactions* on page D13-319
New interfaces and features defined: ACE5, ACE5-Lite, ACE5-LiteDVM, ACE5-LiteACP	Part E *AMBA 5 Protocol Features*
Deprecation of Inner Shareable domain for AMBA5 interfaces	*Shareability domain support* on page F5-452
Deprecation of Barrier transaction support for AMBA5 interfaces	*Barrier transaction support* on page F5-453
Removed Accelerator Coherency Port Interface Restrictions appendix, this content has been superseded by ACE5-LiteACP	Was Appendix B

Table G7-6 Differences between issue F and issue G

Change	Location
Clarified that an EXOKAY response can only be given for exclusive accesses.	*Read and write response structure* on page A3-59
Rewrite of AXI ordering chapter to improve clarity.	Chapter A6 *AXI Ordering Model*
Change that Multi_Copy_Atomicity property must be True for interfaces compliant with Issue G and later.	*Multi-copy write atomicity* on page A7-95
Clarification of permitted responses to exclusive accesses.	*Responses to exclusive access* on page A7-98
Removed table D3-10 as it does not add value to the text.	*Cache line size restrictions* on page D3-182
Rewrite of chapter on Cache Maintenance Operations, adding Clean to Deep Persistence and option for CMOs on write channels.	Chapter D7 *Cache Maintenance*
Clarification of rules for Atomic transactions.	*Atomic transactions attributes* on page E1-343
Clarification that WriteUniqueStash and StashOnce transactions must not cross a cache line boundary.	*Stash transaction signaling* on page E1-352
Change that Stash transactions are permitted to be Inner Shareable.	*Stash transaction signaling* on page E1-352
Change that Deallocating transactions are permitted to be Inner Shareable.	*Deallocating transaction signaling* on page E1-356
Clarified that for Atomic transactions with a read response, **RTRACE** should be asserted if **AWTRACE** is asserted.	*Trace signals* on page E1-357
Clarified that for Atomic transactions with a read response, **RLOOP** must be identical to **AWLOOP**.	*User Loopback signaling* on page E1-359
Clarified rules for **AWAKEUP** assertion.	*Wake-up Signaling* on page E1-362
Clarified that **SYSCOREQ** and **SYSCOACK** are synchronous signals.	*Untranslated transactions* on page E1-369
Change that the Untranslated_Transactions property must not be True for ACE5-LiteDVM interfaces.	*Untranslated transactions* on page E1-369
Added section on read data chunking.	*Read data chunking* on page E1-378
Added section on the read interleaving property.	*Read interleaving property* on page E1-382
Added section on unique ID indication.	*Unique ID indicator* on page E1-383
Added section on MPAM support.	*Memory Partitioning and Monitoring (MPAM)* on page E1-385

Table G7-6 Differences between issue F and issue G (continued)

Change	Location
Added support for interface parity, combined with existing section on data parity and poison.	Chapter E2 *Interface and data protection*
Change that **AWSNOOP** is extended to 4 bits for ACE5-Lite, ACE5-LiteDVM and ACE5-LiteACP interfaces, irrespective of property settings.	*AWSNOOP signal width* on page F5-454
Updated signal matrix with new signals.	*Signal matrix* on page G2-460
Change that **ARVMIDEXT** is optional for interfaces with the DVM_v8.1 property set, where the master can receive but not issue DVM messages.	*Signal matrix* on page G2-460
Updated property table with new properties.	*Summary of interface properties* on page G3-472
Updated tables showing **AxSNOOP** encodings.	Appendix G4 *Summary of AxSNOOP encodings*
Added summary of ID constraints.	Appendix G5 *Summary of ID constraints*
Added interface property timeline.	*Summary of interface properties* on page G3-472

Table G7-7 Differences between issue G and issue H

Change	Location
Clarified rules regarding inter-transaction dependencies for a master.	*Relationships between the channels* on page A3-44
Added recommendation that WDATA and RDATA are zero for inactive byte lanes.	*Channel signaling requirements* on page A3-42
Added section on Regular Transactions	*Regular transactions* on page A3-53
Updated section on User Defined Signaling	*User-defined signaling* on page A8-104
Consolidated all CMO content into a single chapter	Chapter D7 *Cache Maintenance*
Added Write with CMO transaction	*Write with cache maintenance* on page D7-277
Rewrite of DVM message chapter	Chapter D13 *Distributed Virtual Memory Transactions*
Added support for DVM v8.4 messages	*DVM message versions* on page D13-323
Corrected transaction sizes for AtomicStore, AtomicLoad and AtomicSwap	*Atomic transactions attributes* on page E1-343

Table G7-7 Differences between issue G and issue H (continued)

Change	Location
Clarified that for an AtomicCompare transaction, a burst length of 1 is permitted and AWADDR is not required to be aligned.	*Atomic transactions attributes* on page E1-343
Clarified that master might ignore the write response to an Atomic Transaction	*Response signaling* on page E1-347
Renamed width parameters for Loopback signaling	*User Loopback signaling* on page E1-359
Added v2 functionality to Untranslated Transactions	*SMMU flows* on page E1-373
Moved StashTranslation description from Cache Stashing to Untranslated Transactions chapter	*StashTranslation* on page E1-374
Relaxed rule for RCHUNKNUM and RCHUNKSTRB value when RCHUNKV is deasserted	*Read data chunking protocol rules* on page E1-378
Added section on Memory Tagging	*Memory tagging* on page E1-387
Added Prefetch request and response	*Prefetch request and response* on page E1-396
Added Write zero with no data	*Write zero with no data* on page E1-398
Added new interface properties	*Additional interface properties* on page E1-399
Added new signal width properties	*Signal width properties* on page E1-403
Added parity check signals to cover signals added in Issue H	*Parity check signals* on page E2-411
Updated signal matrix with new signals	*Signal matrix* on page G2-460
Removed erroneous signals AWIDUNQCHK and ARIDUNQCHK from check signal matrix	*Check signal matrix* on page G2-466
Updated summary of interface properties, now includes property timeline	Appendix G3 *Interface Property Summary*
Enhanced and updated summary of AxSNOOP encodings	Appendix G4 *Summary of AxSNOOP encodings*
Added summary of response codes	Appendix G6 *Summary of response codes*

Glossary

Aligned
A data item stored at an address that is divisible by the highest power of 2 that divides into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

At approximately the same time
Two events occur at approximately the same time if a remote observer might not be able to determine the order in which they occurred.

AXI beat
See Beat.

AXI burst
See Burst.

AXI transaction
See Transaction.

Barrier
An operation that forces a defined ordering of other actions.

Beat
An individual data transfer within an AXI burst.

See also Burst, Transaction.

Big-endian memory
Means that *the most significant byte* (MSB) of the data is stored in the memory location with the lowest address.

See also Endianness, Little-endian memory,

Blocking
Describes an operation that prevents following actions from continuing until the operation completes.

A non-blocking operation can permit following operations to continue before it completes.

Branch prediction
Is where a processor selects a future execution path to fetch along. For example, after a branch instruction, the processor can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also Prefetching.

Burst In an AXI transaction, the payload data is transferred in a single burst, that can comprise multiple beats, or individual data transfers.

See also Beat, Transaction.

Byte An 8-bit data item.

Cache Any cache, buffer, or other storage structure in a caching master that can hold a copy of the data value for a particular address location.

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

Cache line The basic unit of storage in a cache. Its size in words is always a power of two. A cache line must be aligned to the size of the cache line.

The size of the cache line is equivalent to the coherency granule.

See also Coherency granule.

Cache miss A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Caching master A master component that has a hardware-coherent cache. A caching master has a snoop address and snoop response channel, and optionally, a snoop data channel.

A master component might have only non-coherent caches. These caches can be for private data or they can be software-managed to ensure coherency. A master with a non-coherent cache is not a caching master. That is, the term *caching master* refers to a master with a cache that the ACE protocol must keep coherent.

See also *Initiating master*, *Master component*, *Snooped master*.

ceil() A function that returns the lowest integer value that is equal to or greater than the input to the function.

Coherent Data accesses from a set of observers to a memory location are coherent accesses to that memory location by the members of the set of observers are consistent with there being a single total order of all writes to that memory location by all members of the set of observers.

Coherency granule
The minimum size of the block of memory affected by any coherency consideration. For example, an operation to make two copies of an address coherent makes the two copies of a block of memory coherent, where that block of memory is:

- at least the size of the coherency granule
- aligned to the size of the coherency granule.

In the ACE specification, the coherency granule is the cache line size.

See also Cache line.

Component A distinct functional unit that has at least one AMBA interface. Component can be used as a general term for master, slave, peripheral, and interconnect components.

See also Interconnect component, Master component, Memory slave component, Peripheral slave component, Slave component.

Device *See* Peripheral slave component.

Downstream An AXI transaction operates between a master component and one or more slave components, and can pass through one or more intermediate components. At any intermediate component, for a given transaction, *downstream* means between that component and a destination slave component, and includes the destination slave component.

Downstream and upstream are defined relative to the transaction as a whole, not relative to individual data flows within the transaction.

See also Master component, Slave component, Upstream.

Downstream cache

A downstream cache is defined from the perspective of an initiating master. A downstream cache for a master is one that it accesses using the fundamental AXI transaction channels. An initiating master can allocate cache lines into a downstream cache.

See also Downstream, Initiating master.

Deprecated

Something that is present in the specification for backwards compatibility. Whenever possible you must avoid using deprecated features. These features might not be present in future versions of the specification.

Endianness

An aspect of the system memory mapping.

See also Big-endian memory and Little-endian memory.

Full coherency

A fully coherent master can share data with other masters and allocate that data in its local caches; it can snoop and be snooped. Masters with an ACE interface are fully coherent whereas masters with an ACE-Lite interface are IO coherent.

See also IO coherency

Hit

See Cache hit.

IO coherency

An IO coherent master can share data with other masters but cannot allocate that data in its local caches; it can snoop but no be snooped. Masters with an ACE interface are fully coherent whereas masters with an ACE-Lite interface are IO coherent.

See also Full coherency

IMPLEMENTATION DEFINED

Means that the behavior is not defined by this specification, but must be defined and documented by individual implementations.

in a timely manner

The protocol cannot define an absolute time within which something must occur. However, in a sufficiently idle system, it will make progress and complete without requiring any explicit action.

Initiating master

A master that issues a transaction that starts a sequence of events. When describing a sequence of transactions, the term initiating master distinguishes the master that triggers the sequence of events from any snooped master that is accessed as a result of the action of the initiating master.

Initiating master is a temporal definition, meaning it applies at particular points in time, and typically is used when describing sequences of events. A master that is an initiating master for one sequence of events can be a snooped master for another sequence of events.

See also Caching master, Downstream cache, Local cache, Peer cache, Snooped master.

Interconnect component

A component with more than one AMBA interface that connects one or more master components to one or more slave components

An interconnect component can be used to group together either:

- a set of masters so that they appear as a single master interface
- a set of slaves so that they appear as a single slave interface.

See also Component, Master component, Slave component.

Line

See Cache line.

Little-endian memory

Means that the *least significant byte* (LSB) of the data is stored in the memory location with the lowest address.

See also Big-endian memory, Endianness.

Load
The action of a master component reading the value held at a particular address location. For a processor, a load occurs as the result of executing a particular instruction. Whether the load results in the master issuing a read transaction depends on whether the accessed cache line is held in the local cache.

See also Caching master, Speculative read, Store.

Local cache
A local cache is defined from the perspective of an initiating master. A local cache is one that is internal to the master. Any access to the local cache is performed within the master.

See also Initiating master, Peer cache.

Main memory
The memory that holds the data value of an address location when no cached copies of that location exist. For any location, main memory can be out of date with respect to the cached copies of the location, but main memory is updated with the most recent data value when no cached copies exist.

Main memory can be referred to as memory when the context makes the intended meaning clear.

Master component
A component that initiates transactions.

It is possible that a single component can act as both a master component and as a slave component. For example, a *Direct Memory Access* (DMA) component can be a master component when it is initiating transactions to move data, and a slave component when it is being programmed.

See also Component, Interconnect component, Slave component.

Memory barrier
See Barrier.

Memory Management Unit (MMU)
Provides detailed control of the part of a memory system that provides address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

See also System Memory Management Unit (SMMU).

Memory slave component
A memory slave component, or *memory slave*, is a slave component with the following properties:

- a read of a byte from a memory slave returns the last value written to that byte location.
- a write to a byte location in a memory slave updates the value at that location to a new value that is obtained by subsequent reads.
- reading a location multiple times has no side-effects on any other byte location.
- reading or writing one byte location has no side effects on any other byte location.

See also Component, Master component, Peripheral slave component.

Miss
See Cache miss.

MMU
See Memory Management Unit (MMU).

Observer
A processor or other master component, such as a peripheral device, that can generate reads from or writes to memory.

Peer cache
A peer cache is defined from the perspective of an initiating master. A peer cache for that master is one that is accessed using the snoop channels. An initiating master cannot allocate cache lines into a peer cache.

See also Initiating master, Local cache.

Peripheral slave component
A peripheral slave component is also described as a *peripheral slave*. This specification recommends that a peripheral slave has an IMPLEMENTATION DEFINED method of access that is typically described in the data sheet for the component. Any access that is not defined as permitted might cause the peripheral slave to fail, but must complete in a protocol-correct manner to prevent system deadlock. The protocol does not require continued correct operation of the peripheral.

In the context of the descriptions in this specification, peripheral slave is synonymous with *peripheral*, *peripheral component*, *peripheral device*, and *device*.

See also Memory slave component, Slave component.

Permission to store

A master component has permission to store if it can perform a store to the associated cache line without informing any other caching master or the interconnect.

See also Caching master, Master component, Permission to update main memory, Store.

Permission to update main memory

A master component has permission to update main memory if the master can perform a write transaction to main memory. The ACE protocol ensures that no other master performs a write transaction to the same cache location at the same time.

See also Caching master, Master component, Main memory, Permission to store, Store.

PoS

Point of Serialization. The point through which all transactions to a given address must pass and the order in which the transactions are processed is determined.

Prefetching

Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Slave component

A component that receives transactions and responds to them.

It is possible that a single component can act as both a slave component and as a master component. For example, a *Direct Memory Access* (DMA) component can be a slave component when it is being programmed and a master component when it is initiating transactions to move data.

See also Master component, Memory slave component, Peripheral slave component.

Snooped cache

A hardware-coherent cache on a snooped master. That is, it is a hardware-coherent cache that receives snoop transactions.

The term snooped cache is used in preference to the term snooped master when the sequence of events being described only involves the cache and does not involve any actions or events on the associated master.

See also Snooped master,

Snoop filter

A precise snoop filter that is able to track precisely the cache lines that might be allocated within a master.

Snooped master

A caching master that receives snoop transactions.

Snooped master is a temporal definition, meaning it applies at particular points in time, and typically is used when describing sequences of events. A master that is a snooped master for one sequence of events can be an initiating master for another sequence of events.

See also Caching master, Initiating master, Snooped cache.

Speculative read

A transaction that a master issues when it might not need the transaction to be performed because it already has a copy of the accessed cache line in its local cache. Typically, a master issues a speculative read in parallel with a local cache lookup. This gives lower latency than looking in the local cache first, and then issuing a read transaction only if the required cache line is not found in the local cache.

See also Caching master, Load.

Store
The action of a master component changing the value held at a particular address location. For a processor, a store occurs as the result of executing a particular instruction. Whether the store results in the master issuing a read or write transaction depends on whether the accessed cache line is held in the local cache, and if it is in the local cache, the state it is in.

See also Caching master, Load, Permission to update main memory, Permission to store.

Synchronization barrier
See Barrier.

System Memory Management Unit (SMMU)
A system-level MMU. That is, a system component that provides address translation from a one address space to another. An SMMU provides one or more of:

- *virtual address* (VA) to *physical address* (PA) translation
- VA to *intermediate physical address* (IPA) translation
- IPA to PA translation.

TLB
See Translation Lookaside Buffer (TLB).

Transaction
An AXI master initiates an AXI transaction to communicate with an AXI slave. Typically, the transaction requires information to be exchanged between the master and slave on multiple channels. The complete set of required information exchanges form the AXI transaction.

See also Beat, Burst.

Translation Lookaside Buffer (TLB)
A memory structure containing the results of translation table walks. TLBs help to reduce the average cost of a memory access.

See also System Memory Management Unit (SMMU), Translation table, Translation table walk.

Translation table
A table held in memory that defines the properties of memory areas of various sizes from 1KB.

See also Translation Lookaside Buffer (TLB), Translation table walk.

Translation table walk
The process of doing a full translation table lookup.

See also Translation Lookaside Buffer (TLB), Translation table.

Unaligned
An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

See also *Aligned* on page Glossary-493

UNPREDICTABLE
In the AMBA AXI and ACE Architecture means that the behavior cannot be relied upon.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

Upstream
An AXI transaction operates between a master component and one or more slave components, and can pass through one or more intermediate components. At any intermediate component, for a given transaction, *upstream* means between that component and the originating master component, and includes the originating master component.

Downstream and upstream are defined relative to the transaction as a whole, not relative to individual data flows within the transaction.

See also Downstream, Master component, Slave component.

Write-Back cache
A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or re-allocated. Another common term for a Write-Back cache is a *copy-back cache*.

Write-Through cache

A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the processor.