AMBA® 5 CHI

Architecture Specification

ARM IHI 0050E.a (ID081920)

AMBA 5 CHI

Release Information

The *Change history* lists the changes made to this specification.

Change history

Date	Issue	Confidentiality	Change
12 June 2014	A	Confidential	First limited release
04 August 2017	B	Non-Confidential	First public release
08 May 2018	C	Non-Confidential	Second public release
28 August 2019	D	Non-Confidential	Third public release
19 August 2020	E.a	Non-Confidential	Fourth public release

Proprietary Notice

This document is **NON-CONFIDENTIAL** and any use by you is subject to the terms of this notice and the Arm AMBA Specification Licence set about below.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at http://www.arm.com/company/policies/trademarks.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
LES-PRE-21451 version 2.2

AMBA SPECIFICATION LICENCE

THIS END USER LICENCE AGREEMENT ("LICENCE") IS A LEGAL AGREEMENT BETWEEN YOU (EITHER A SINGLE INDIVIDUAL, OR SINGLE LEGAL ENTITY) AND ARM LIMITED ("ARM") FOR THE USE OF ARM'S INTELLECTUAL PROPERTY (INCLUDING, WITHOUT LIMITATION, ANY COPYRIGHT) IN THE RELEVANT AMBA SPECIFICATION ACCOMPANYING THIS LICENCE. ARM LICENSES 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.

"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 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 5 CHI Architecture Specification

Preface

Chapter 1 Introduction

Chapter 2 Transactions

Chapter 3 Network Layer

Appendix B Communicating Nodes

Appendix C Revisions

Glossary

Preface

This preface introduces the *AMBA 5 CHI Architecture Specification*. It contains the following sections:

About this specification

This specification describes the AMBA 5 CHI architecture.

Intended audience

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

Using this specification

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to the CHI architecture, and the terminology used in this specification.

Chapter 2 *Transactions*

Read this for an overview of the communication channels between nodes, the associated packet fields, transaction structures, transaction ID flows, and the supported transaction ordering.

Chapter 3 *Network Layer*

Read this for a description of the Network layer that is responsible for determining the node ID of a destination node.

Chapter 4 *Coherence Protocol*

Read this for an introduction to the coherence protocol.

Chapter 5 *Interconnect Protocol Flows*

Read this for examples of protocol flows for different transaction types.

Chapter 6 *Exclusive Accesses*

Read this for a description of the mechanisms that the architecture includes to support Exclusive accesses.

Chapter 7 *Cache Stashing*

Read this for a description of the cache stashing mechanism whereby data can be installed in a cache.

Chapter 8 *DVM Operations*

Read this for a description of DVM operations that the protocol uses to manage virtual memory.

Chapter 9 *Error Handling*

Read this for a description of the error response requirements.

Chapter 10 *Quality of Service*

Read this for a description of the mechanisms that the protocol includes to support *Quality of Service* (QoS).

Chapter 11 *System Debug, Trace, and Monitoring*

Read this for a description of the mechanisms that provide additional support for the debugging, tracing, and performance measurement of systems.

Chapter 12 *Memory Tagging*

Read this for a description of the *Memory Tagging Extension* (MTE) that provides a mechanism to check the correct usage of data held in memory.

Chapter 13 *Link Layer*

Read this for a description of the Link layer that provides a mechanism for packet based communication between protocol nodes and the interconnect.

Chapter 14 *Link Handshake*

Read this for a description of the Link layer handshake requirements.

Chapter 15 *System Coherency Interface*

Read this for a description of the interface signals that support connecting and disconnecting components from both the Coherency and DVM domains.

Chapter 16 *Properties, Parameters, and Broadcast Signals*

Read this for a description of the optional signals that provide flexibility in configuring optional interface properties.

Appendix A *Message Field Mappings*

Read this for the field mappings for messages.

Appendix B *Communicating Nodes*

Read this for the node pairs that can legally communicate within the protocol.

Appendix C *Revisions*

Read this for a description of the technical changes between released issues of this specification.

Glossary Read this for definitions of terms used in this specification.

Conventions

The following sections describe conventions that this specification can use:

- *Typographical conventions*.
- *Timing diagrams*.
- *Signals* on page xvi.
- *Numbers* on page xvi.

Typographical conventions

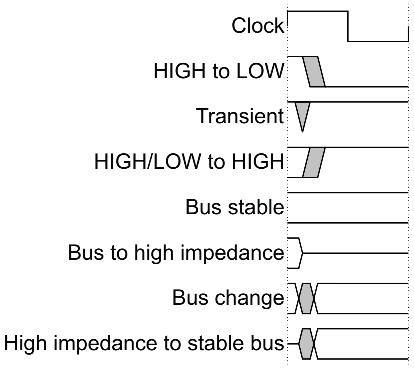
The typographical conventions are:

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 *Key to timing diagram conventions* figure 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.

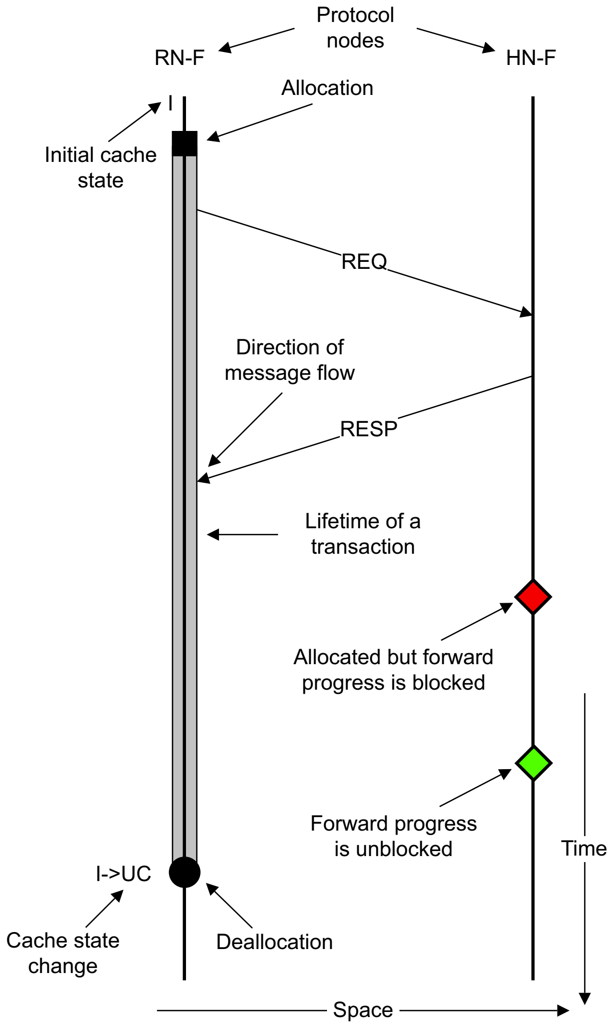


Key to timing diagram conventions

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

Time-Space diagrams

The *Key to Time-Space diagram conventions* figure explains the format used to illustrate protocol flow.



Key to Time-Space diagram conventions

In the Time-Space diagram:

- The protocol nodes are positioned along the horizontal axis and time is indicated vertically, top to bottom.
- The lifetime of a transaction at a protocol node is shown by an elongated shaded rectangle along the time axis from allocation to the deallocation time.
- The initial cache state at the node is shown at the top.
- The diamond shape on the timeline indicates arrival of a request and whether its processing is blocked waiting for another event to complete.
- The cache state transition, upon the occurrence of an event, is indicated by I->UC.

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.
Lowercase n	At the start or end of a signal name denotes an active-LOW signal.

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® AXI and ACE Protocol Specification* (ARM IHI 0022).

Feedback

Arm welcomes feedback on its documentation.

Feedback on this specification

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

- The title, *AMBA 5 CHI Architecture Specification*.
- The number, ARM IHI 0050E.a.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the CHI architecture and the terminology used throughout this specification. It contains the following sections:

1.1 Architecture overview

The CHI architecture provides a comprehensive layered specification to build small, medium, and large systems comprising of multiple components using a scalable coherent hub interface and on-chip interconnect. The CHI architecture permits flexibility on the topology of the component connections, which can be driven from the system performance, power, and area requirements.

The components of CHI based systems can comprise of standalone processors, processor clusters, graphic processors, memory controllers, I/O bridges, PCIe subsystems and the interconnect itself.

The key features of the architecture are:

- Scalable architecture, enabling modular designs that scale from small to large systems.
- Independent layered approach, comprising of Protocol, Network, and Link layer, with distinct functionalities.
- Packet-based communication.
- All transactions handled by an interconnect-based Home Node that co-ordinates required snoops, cache, and memory accesses.
- The CHI coherence protocol supports:
    - Coherency granule of 64-byte cache line.
    - Snoop filter and directory based systems for snoop scaling.
    - Both MESI and MOESI cache models with forwarding of data from any cache state.
    - Additional partial and empty cache line states.
- The CHI transaction set includes:
    - Enriched transaction types that permit performance, area, and power efficient system cache implementation.
    - Support for atomic operations and synchronization within the interconnect.
    - Support for the efficient execution of Exclusive accesses.
    - Transactions for the efficient movement and placement of data, to move data in a timely manner closer to the point of anticipated use.
    - Virtual memory management through *Distributed Virtual Memory* (DVM) operations.
- Request retry to manage protocol resources.
- Support for end-to-end *Quality of Service* (QoS).
- Support for the Arm *Memory Tagging Extension* (MTE).
- Configurable data width to meet the requirements of the system.
- ARM TrustZone™ support on a transaction-by-transaction basis.
- Optimized transaction flow for coherent writes with a producer-consumer ordering model.
- Error reporting and propagation across components and interconnect for system reliability and integrity.
- Handling sub cache line data errors using Data Poisoning and per byte error indication.
- Power-aware signaling on the component interface:
    - Enabling flit-level clock gating.
    - Component activation and deactivation sequence for clock-gate and power-gate control.
    - Protocol activity indication for power and clock control.

1.1.1 Architecture layers

Functionality is grouped into the following layers:

- Protocol.
- Network.
- Link.

Table 1-1 describes the primary function of each layer.

Table 1-1 Layers of the CHI architecture

Layer	Communication granularity	Primary function
Protocol	Transaction	The Protocol layer is the top-most layer in the CHI architecture. The function of the Protocol layer is to: • Generate and process requests and responses at the protocol nodes. • Define the permitted cache state transitions at the protocol nodes that include caches. • Define the transaction flows for each request type. • Manage the protocol level flow control.
Network	Packet	The function of the Network layer is to: • Packetize the protocol message. • Determine, and add to the packet, the source and target node IDs required to route the packet over the interconnect to the required destination.
Link	Flit	The function of the Link layer is to: • Provide flow control between network devices. • Manage link channels to provide deadlock-free switching across the network.

1.2 Topology

The CHI architecture is primarily topology-independent. However, certain topology-dependent optimizations are included in this specification to make implementation more efficient. Figure 1-1 shows three examples of topologies selected to show the range of interconnect bandwidth and scalability options that are available.

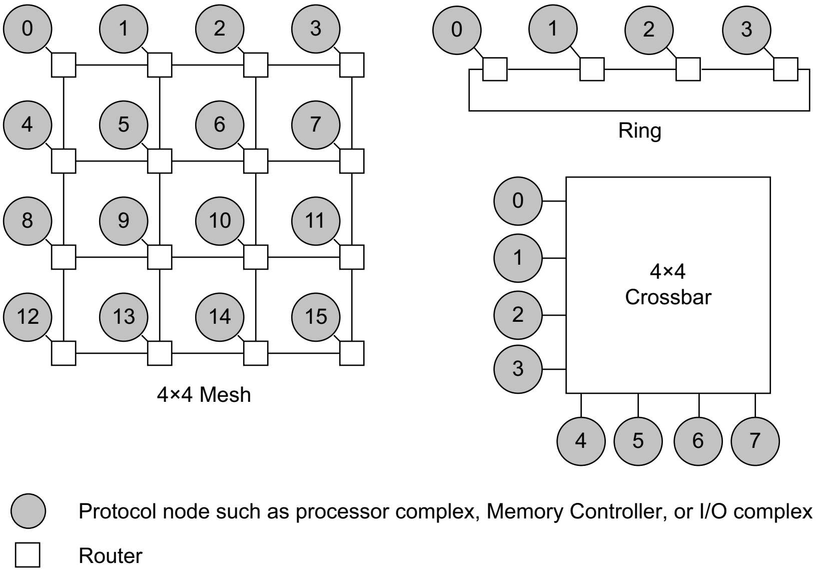


Figure 1-1 Example interconnect topologies

Crossbar This topology is simple to build, and naturally provides an ordered network with low latency. It is suitable where the wire counts are still relatively small. This topology is suitable for an interconnect with a small number of nodes.

Ring This topology provides a good trade-off between interconnect wiring efficiency and latency. The latency increases linearly with the number of nodes on the ring. This topology is suitable for a medium size interconnect.

Mesh This topology provides greater bandwidth at the cost of more wires. It is very modular and can be easily scaled to larger systems by adding more rows and columns of switches. This topology is suitable for a larger scale interconnect.

1.3 Terminology

The following terms have a specific meaning in this specification:

Transaction	A transaction carries out a single operation. Typically, a transaction either reads from memory or writes to memory.
Message	A Protocol layer term that defines the granule-of-exchange between two components. Examples are: • Request. • Data response. • Snoop request. A single Data response message might be made up of a number of packets.
Packet	The granule-of-transfer over the interconnect between endpoints. A message might be made up of one or more packets. For example, a single Data response message can be made up of 1 to 4 packets. Each packet contains routing information, such as destination ID and source ID that enables it to be routed independently over the interconnect.
Flit	The smallest flow control unit. A packet can be made up of one or more flits. All the flits of a given packet follow the same path through the interconnect. **Note** For CHI, all packets consist of a single flit.
Phit	The physical layer transfer unit. A flit can be made up of one or more phits. A phit is defined as one transfer between two adjacent network devices. **Note** For CHI, all flits consist of a single phit.
PoS	Point of Serialization. A point within the interconnect where the ordering between Requests from different agents is determined.
PoC	Point of Coherence. A point at which all agents that can access memory are guaranteed to see the same copy of a memory location. In a typical CHI based system it is the HN-F in the interconnect.
PoP	Point of Persistence. The point in a memory system, if it exists, at or beyond the Point of Coherency, where a write to memory is maintained when system power is removed, and reliably recovered when power is restored to the affected locations in memory.
Downstream cache	A downstream cache is defined from the perspective of a Request Node. A downstream cache for a Request, is a cache that the Request accesses using CHI Request transactions. A Request Node can send a Request with data to allocate data into a downstream cache.
Requester	A component that starts a transaction by issuing a Request message. The term Requester can be used for a component that independently initiates transactions and such a component is also referred to as a master. The term Requester can also be used for an interconnect component that issues a downstream Request message independently or as a side-effect of other transactions that are occurring in the system.
Completer	Any component that responds to a transaction it receives from another component. A Completer can either be an interconnect component, such as Home Node or a Misc Node, or a component, such as a slave, that is outside of the interconnect.
Master	An agent that independently issues transactions. Typically a master is the most upstream agent in a system. A master can also be referred to as a Requester.

Slave	An agent that receives transactions and completes them appropriately. Typically, a slave is the most downstream agent in a system. A slave can also be referred to as a Completer or Endpoint.
Endpoint	Another name for a slave component. As the name implies, an endpoint is the final destination for a transaction.
Protocol Credit	A credit, or guarantee, from a Completer that it will accept a transaction.
Link layer Credit	A credit, or guarantee, that a flit will be accepted on the other side of the link. A *Link layer Credit* (L-Credit) is a credit for a single hop at the Link layer.
ICN	A short form of interconnect, which is the CHI transport mechanism that is used for communication between protocol nodes. An ICN might include a fabric of switches connected in a ring, mesh, crossbar, or some other topology. The ICN might include protocol nodes such as Home Node and Misc Node. The topology of the ICN is IMPLEMENTATION DEFINED.
IPA	Intermediate Physical Address. In two stage address translation: • Stage one results in an Intermediate Physical Address. • Stage two provides the Physical Address.
RN	Request Node. Generates protocol transactions, including reads and writes, to the interconnect.
HN	Home Node. Node located within the interconnect that receives protocol transactions from Request Nodes, completes the required Coherency action, and returns a Response.
SN	Slave Node. Node that receives a Request from a Home Node, completes the required action, and returns a Response.
MN	Misc or Miscellaneous Node. Node located within the interconnect that receives DVM messages from Request Nodes, completes the required action, and returns a Response.
IO Coherent node	An RN that generates a subset of Snoopable requests in addition to Non-snoopable requests. The Snoopable requests that an IO Coherent node generates do not result in the caching of the received data in a coherent state. Therefore, an IO Coherent node does not receive any Snoop requests.
Snoopee	A Request Node that is receiving a snoop.
Write-Invalidate protocol	A protocol in which an RN writing to a cache line that is shared in the system must invalidate all the shared copies before proceeding with the write. The CHI protocol is a Write-Invalidate protocol.
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.
Don't Care	A field value that indicates that the field can be set to any value, including reserved or illegal values. Any component receiving a packet with a field value set to Don't Care must ignore the value set for that field.
Inapplicable	A field value that indicates that the field is not used in the processing of the message.

1.4 Transaction classification

The protocol transactions that this specification supports, and their major classification, are as follows:

Read

- ReadNoSnp, ReadNoSnpSep.
- ReadOnce.
- ReadOnceCleanInvalid.
- ReadOnceMakeInvalid.
- ReadClean.
- ReadNotSharedDirty.
- ReadShared.
- ReadUnique.
- ReadPreferUnique.
- MakeReadUnique.

Dataless

- CleanUnique.
- MakeUnique.
- Evict.
- StashOnceUnique, StashOnceSepUnique.
- StashOnceShared, StashOnceSepShared.
- CleanShared.
- CleanSharedPersist.
- CleanSharedPersistSep.
- CleanInvalid.
- MakeInvalid.

Write

- WriteNoSnpPtl, WriteNoSnpFull, WriteNoSnpZero.
- WriteUniquePtl, WriteUniqueFull, WriteUniqueZero.
- WriteUniquePtlStash, WriteUniqueFullStash.
- WriteBackPtl, WriteBackFull.
- WriteCleanFull.
- WriteEvictFull, WriteEvictOrEvict.

Combined Write

- WriteNoSnpPtlCleanSh, WriteNoSnpPtlCleanInv, WriteNoSnpPtlCleanShPerSep.
- WriteNoSnpFullCleanSh, WriteNoSnpFullCleanInv, WriteNoSnpFullCleanShPerSep.
- WriteUniquePtlCleanSh, WriteUniquePtlCleanShPerSep.
- WriteUniqueFullCleanSh, WriteUniqueFullCleanShPerSep.
- WriteBackFullCleanSh, WriteBackFullCleanInv, WriteBackFullCleanShPerSep.
- WriteCleanFullCleanSh, WriteCleanFullCleanShPerSep.

Atomic

- AtomicStore.
- AtomicLoad.
- AtomicSwap.
- AtomicCompare.

Other

- DVMOp.
- PrefetchTgt.
- PCrdReturn.

Snoop

- SnpOnceFwd.
- SnpOnce.
- SnpStashUnique.
- SnpStashShared.
- SnpCleanFwd.
- SnpClean.
- SnpNotSharedDirtyFwd.
- SnpNotSharedDirty.
- SnpSharedFwd.
- SnpShared.
- SnpUniqueFwd.
- SnpUnique.
- SnpPreferUniqueFwd.
- SnpPreferUnique.
- SnpUniqueStash.
- SnpCleanShared.
- SnpCleanInvalid.
- SnpMakeInvalid.
- SnpMakeInvalidStash.
- SnpQuery.
- SnpDVMOp.

In this specification, unless specifically stated otherwise:

- ReadOnce∗ represents ReadOnce, ReadOnceCleanInvalid and ReadOnceMakeInvalid.
- WriteNoSnp represents both WriteNoSnpPtl and WriteNoSnpFull.
- WriteUnique represents WriteUniquePtl, WriteUniqueFull, WriteUniquePtlStash and WriteUniqueFullStash.
- WriteBack represents both WriteBackPtl and WriteBackFull.
- StashOnce represents both StashOnceUnique and StashOnceShared.
- StashOnceSep represents both StashOnceSepUnique and StashOnceSepShared.
- StashOnce∗ represents both StashOnce and StashOnceSep.
- StashOnce∗Unique represents both StashOnceUnique and StashOnceSepUnique.
- StashOnce∗Shared represents both StashOnceShared and StashOnceSepShared.
- CleanSharedPersist* represents both CleanSharedPersist and CleanSharedPersistSep.
- SnpStash represents both SnpStashUnique and SnpStashShared.

1.5 Coherence overview

Hardware coherency enables the sharing of memory by system components without the 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.

1.5.1 Coherency model

Figure 1-2 shows an example coherent system that includes three master components, each with a local cache and coherent protocol node. The protocol permits cached copies of the same memory location to reside in the local cache of one or more master components.

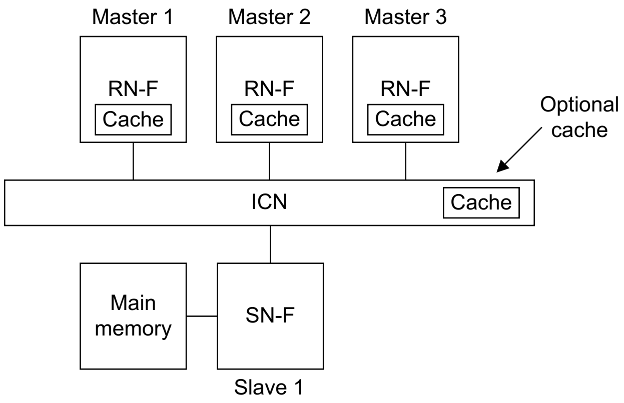


Figure 1-2 Example coherency model

The coherence protocol ensures that all masters observe the correct data value at any given address location by enforcing that no more than 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, to permit multiple cached copies to exist.

All coherency is maintained at cache line granularity. A cache line is defined as a 64-byte aligned memory region that is 64-bytes in size.

The protocol does not require main memory to be 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 protocol enables master components to determine whether 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.

1.5.2 Cache state model

To determine whether an action is required when a component accesses a cache line, the protocol defines cache states. Each cache state is based on the following cache line characteristics:

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 this 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.
Full, Partial, Empty	A Full cache line has all bytes valid. A Partial cache line might have some bytes valid, but not all bytes valid. An Empty cache line has no bytes valid.

Figure 1-3 shows the seven state cache model. *Cache line states* on page 4-162 gives further information about each cache state.

A valid cache state name that is not Partial or Empty is considered to be Full. In Figure 1-3 UC, UD, SC, and SD are all Full cache line states.

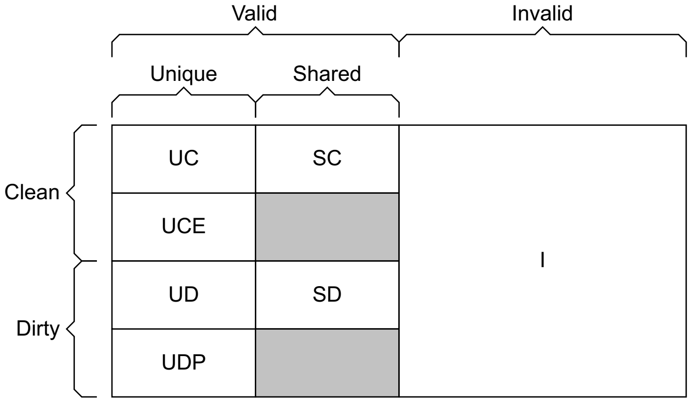


Figure 1-3 Cache state model

1.6 Component naming

Components are classified by CHI protocol node type:

RN Request Node. Generates protocol transactions, including reads and writes to the interconnect.

An RN is further categorized as:

RN-F Fully coherent Request Node:

- Includes a hardware-coherent cache.
- Permitted to generate all transactions defined by the protocol.
- Supports all Snoop transactions.

RN-D IO coherent Request Node with DVM support:

- Does not include a hardware-coherent cache.
- Receives DVM transactions.
- Generates a subset of transactions defined by the protocol.

RN-I IO coherent Request Node:

- Does not include a hardware-coherent cache.
- Does not receive DVM transactions.
- Generates a subset of transactions defined by the protocol.
- Does not require snoop functionality.

HN Home Node. Node located within the interconnect that receives protocol transactions from RNs.

An HN is further categorized as:

HN-F Fully coherent Home Node:

- Is expected to receive all Request types except DVMOp.
- Includes a *Point of Coherence* (PoC) that manages coherency by snooping the required RN-Fs, consolidating the snoop responses for a transaction, and sending a single response to the requesting RN.
- Is expected to be the *Point of Serialization* (PoS) that manages order between memory requests.
- Might include a directory or snoop filter to reduce redundant snoops.

Note

IMPLEMENTATION SPECIFIC, can include an integrated ICN cache.

HN-I Non-coherent Home Node:

- Processes a limited subset of Request types defined by the protocol.
- Does not include a PoC and is not capable of processing a Snoopable request. On receipt of a Snoopable request must respond with a protocol compliant message.
- Is expected to be the PoS that manages order between IO requests targeting the IO subsystem.

MN Miscellaneous Node:

- Receives a DVM transaction from an RN, completes the required action, and returns a response.

SN	Slave Node. An SN receives a request from an HN, completes the required action and returns a response.

An SN is further categorized as:

SN-F	A Slave Node type used for Normal memory. It can process Non-snoopable read write, and atomic requests, including exclusive variants of them, and *Cache Maintenance Operation* (CMO) requests.
SN-I	A Slave Node type used for peripherals or Normal memory. It can process Non-snoopable read, write and atomic requests, including exclusive variants of them, and CMO requests.

Figure 1-4 shows various protocol node types connected through an interconnect.

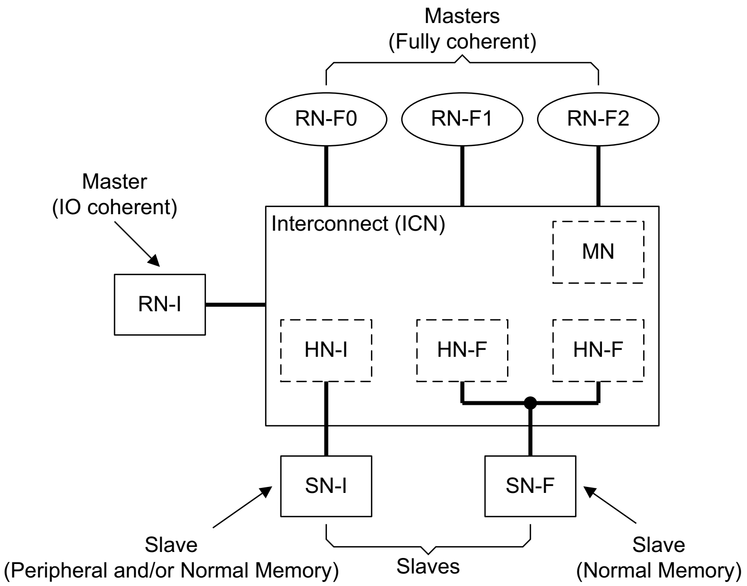


Figure 1-4 Protocol node examples

1.7 Read data source

In a CHI-based system, a Read request can obtain data from different sources. Figure 1-5 shows that these sources are:

- Cache within ICN.
- Slave Node.
- Peer RN-F.

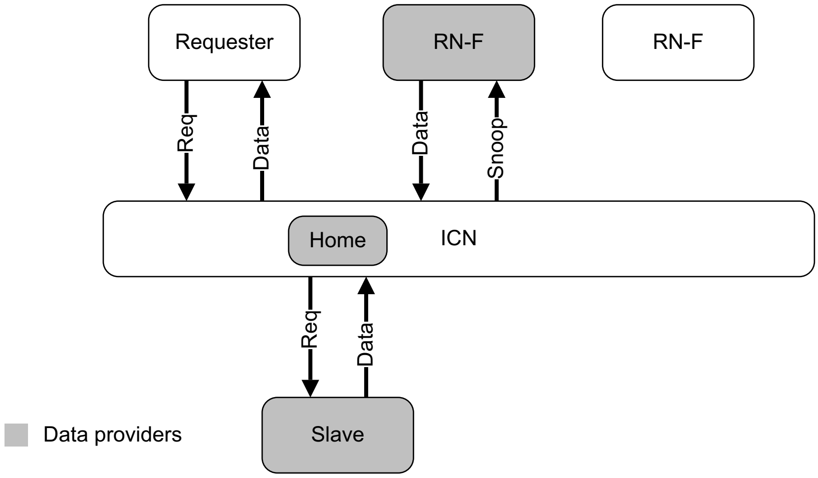


Figure 1-5 Possible Data providers for a Read request

One option for Home is to request that the RN-F or Slave Node returns data only to Home. The Home in turn forwards a copy of the received data to the Requester. A hop in obtaining Data in this Read transaction flow can be removed if the Data provider is enabled to forward the Data response directly to the Requester instead of via the Home.

The Read and Write latency saving and the interconnect bandwidth utilization reduction techniques supported in this specification, which use the reduction of the number of hops, can be categorized as:

***Direct Memory Transfer* (DMT)**

Defines the feature that permits the Slave Node to send Data directly to the Requester.

***Direct Cache Transfer* (DCT)**

Defines the feature which permits a peer RN-F to send Data directly to the Requester.

The Data provider in the DCT Read transaction flows has to inform the Home that it has sent Data to the Requester and, in some cases, it also has to send a copy of Data to the Home.

***Direct Write-data Transfer* (DWT)**

Defines the feature which permits the requesting RN to send Write data directly to the Slave Node.

Chapter 2
Transactions

This chapter gives an overview of the communication channels between nodes, the associated packet fields, and the transaction structure. It contains the following sections:

2.1 Channels overview

Communication between nodes is channel based. Table 2-1 shows the channel naming and the channel designations at the RN and SN nodes.

This section uses shorthand naming for the channels to describe the transaction structure. Table 2-1 shows the shorthand name and the physical channel name that exists on the RN or SN component.

See *Channel* on page 13-385 for the mapping of physical channels on the RN and SN components.

Table 2-1 Channel naming and designation at the RN and SN nodes

Channel	RN channel designation	SN channel designation
REQ	TXREQ. Outbound Request.	RXREQ. Inbound Request.
WDAT	TXDAT. Outbound Data. Used for write data, atomic data, snoop data, forward data.	RXDAT. Inbound Data. Used for write data, atomic data.
SRSP	TXRSP. Outbound Response. Used for Snoop response and Completion Acknowledge.	-
CRSP	RXRSP. Inbound Response. Used for responses from the Completer.	TXRSP. Outbound Response. Used for responses from the Completer.
RDAT	RXDAT. Inbound Data. Used for read data, atomic data.	TXDAT. Outbound Data. Used for read data, atomic data.
SNP	RXSNP. Inbound Snoop request.	-

2.2 Channel fields

This section gives a brief overview of the channel fields and indicates which fields will affect the transaction structure. The fields associated with each channel are described in the following sections:

- *Transaction request fields*.
- *Snoop request fields* on page 2-38.
- *Data fields* on page 2-39.
- *Response fields* on page 2-41.

2.2.1 Transaction request fields

Table 2-2 shows the fields associated with a Request packet.

The term transaction structure is used to describe the different packets that build a transaction and the transaction structure can vary depending on a number of factors. Table 2-2 shows which Request fields can affect the transaction structure. More information on the different transaction structures can be found in *Transaction structure* on page 2-42 and *Flit packet definitions* on page 13-398.

Table 2-2 Request channel fields

Field	Affects structure	Description
QoS	No	Quality of Service priority. Specifies 1 of 16 possible priority levels for the transaction with ascending values of QoS indicating higher priority levels. See Chapter 10 *Quality of Service*.
TgtID	No	Target ID. The node ID of the port on the component to which the packet is targeted. See *Details of transaction identifier fields* on page 2-89 and *System address map* on page 3-152.
SrcID	No	Source ID. The node ID of the port on the component from which the packet was sent. See *Details of transaction identifier fields* on page 2-89.
TxnID	No	Transaction ID. A transaction has a unique transaction ID per source node. See *Details of transaction identifier fields* on page 2-89.
LPID	No	Logical Processor ID. Used in conjunction with the SrcID field to uniquely identify the logical processor that generated the request. See *Logical Processor Identifier* on page 2-115.
PGroupID	No	Persistence Group ID. Indicates the set of CleanSharedPersistSep transactions to which the request applies. See *PGroupID* on page 13-406.
GroupIDExt	No	Group ID Extend. Extends TagGroupID, StashGroupID and PGroupID fields as applicable. See *GroupIDExt* on page 13-406.
Deep	No	Deep persistence. Indicates that the Persist response must not be sent until all earlier writes are written to the final destination. See *Use of the Deep attribute in Persistent CMO* on page 4-172.
ReturnNID	No	Return Node ID. The node ID that the response with Data is to be sent to. See *Details of transaction identifier fields* on page 2-89.
ReturnTxnID	No	Return Transaction ID. The unique transaction ID that conveys the value of TxnID in the data response from the Slave. See *Details of transaction identifier fields* on page 2-89.
StashNID	No	Stash Node ID. The node ID of the Stash target. See *StashNID* on page 13-406.
StashNIDValid	Yes	Stash Node ID Valid. Indicates that the StashNID field has a valid Stash target value. See *StashNIDValid* on page 13-407.

Table 2-2 Request channel fields (continued)

Field	Affects structure	Description
StashLPID	No	Stash Logical Processor ID. The ID of the logical processor at the Stash target. See *StashLPID* on page 13-407.
StashLPIDValid	No	Stash Logical Processor ID Valid. Indicates that the StashLPID field value must be considered as the Stash target. See *StashLPIDValid* on page 13-407.
StashGroupID	No	Stash Group ID. Indicates the set of StashOnceSep transactions to which the request applies. See *StashGroupID* on page 13-408.
Opcode	Yes	Request opcode. Specifies the transaction type and is the primary field that determines the transaction structure. See *Request types* on page 4-164 and *REQ channel opcodes* on page 13-409.
Addr	No	Address. The address of the memory location being accessed for Read and Write requests. See *Address* on page 2-127 and *Addr* on page 13-415.
NS	No	Non-secure. Determines if the transaction is Non-secure or Secure. See *Non-secure bit* on page 2-128 and *NS* on page 13-415.
Size	Yes	Data size. Specifies the size of the data associated with the transaction. This determines the number of data packets within the transaction. See *Data transfer* on page 2-136.
AllowRetry	Yes	Allow Retry. Determines if the target is permitted to give a Retry response. See *Request Retry* on page 2-147.
PCrdType	No	Protocol Credit Type. Indicates the type of Protocol Credit being used by a request that has the AllowRetry field deasserted. See *Request Retry* on page 2-147.
ExpCompAck	Yes	Expect CompAck. Indicates that the transaction will include a Completion Acknowledge message. See *Transaction structure* on page 2-42 and *Ordering* on page 2-116.
MemAttr	No	Memory attribute. Determines the memory attributes associated with the transaction. See *Memory Attributes* on page 2-128.
SnpAttr	No	Snoop attribute. Specifies the snoop attributes associated with the transaction. See *Likely Shared* on page 2-133.
DoDWT	Yes	Do Direct Write Transfer. Supports Direct Write-data Transfer and the handling of Combined Writes. See *DoDWT* on page 13-417.
SnoopMe	No	Snoop Me. Indicates that Home must determine whether to send a snoop to the Requester. See *Atomics* on page 2-78.
LikelyShared	No	Likely Shared. Provides an allocation hint for downstream caches. See *Likely Shared* on page 2-133.
SLCRepHint	No	System Level Cache Replacement Hint. Forwards cache replacement hints from the Requesters to the caches in the interconnect. See *SLC replacement hint* on page 11-353.
Excl	No	Exclusive access. Indicates that the corresponding transaction is an Exclusive access transaction. See Chapter 6 *Exclusive Accesses*.
Order	Yes	Order requirement. Determines the ordering requirement for this request with respect to other requests from the same agent. See *Ordering* on page 2-116.

Table 2-2 Request channel fields (continued)

Field	Affects structure	Description
Endian	No	Endianness. Indicates the endianness of Data in the Data packet for Atomic transactions. See *Endianness* on page 2-140.
TagOp	Yes	Tag Operation. Indicates the operation to be performed on the tags present in the corresponding DAT channel. See *TagOp* on page 13-421.
TagGroupID	No	TagGroupID. Precise contents are IMPLEMENTATION DEFINED. Typically expected to contain Exception Level, TTBR value, and CPU identifier. See *TagGroupID* on page 13-421
TraceTag	No	Trace Tag. Provides additional support for the debugging, tracing, and performance measurement of systems. See Chapter 11 *System Debug, Trace, and Monitoring*.
MPAM	No	Memory System Performance Resource Partitioning and Monitoring. Efficiently utilizes the memory resources among users and monitors their use. See *MPAM* on page 11-355.
RSVDC	No	User defined. See *RSVDC* on page 13-427.

2.2.2 Snoop request fields

Table 2-3 shows the Snoop Request fields. Many of the Snoop Request fields are the same as fields defined for the Request channel.

Table 2-3 Snoop request fields

Field	Affects structure	Description
QoS	No	Quality of Service priority. As defined in *Request channel fields* on page 2-35. See Chapter 10 *Quality of Service*.
TxnID	No	Transaction ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
FwdNID	No	Forward Node ID. Node ID of the original Requester. See *Details of transaction identifier fields* on page 2-89.
FwdTxnID	No	Forward Transaction ID. The transaction ID used in the Request by the original Requester. See *Details of transaction identifier fields* on page 2-89.
StashLPID	No	Stash Logical Processor ID. As defined in *Request channel fields* on page 2-35. See *Stash messages* on page 7-292.
StashLPIDValid	No	Stash Logical Processor ID Valid. As defined in *Request channel fields* on page 2-35. See *Stash messages* on page 7-292.
VMIDExt	No	Virtual Machine ID Extension. See *DVM Operation types* on page 8-305.
SrcID	No	Source ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
Opcode	Yes	Snoop opcode. See *Snoop request fields* and *SNP channel opcodes* on page 13-413.
Addr	No	Address. The address of the memory location being accessed for Snoop requests. See *Address* on page 2-127 and *Addr* on page 13-415.
NS	No	Non-secure or Secure access. As defined in *Request channel fields* on page 2-35. See *Non-secure bit* on page 2-128 and *NS* on page 13-415.
DoNotGoToSD	No	Do Not Go To SD state. Controls Snoopee use of SD state. See *Do not transition to SD* on page 4-233.
RetToSrc	Yes	Return to Source. Instructs the receiver of the snoop to return Data with the Snoop response. See *Returning Data with Snoop response* on page 4-232.
TraceTag	No	Trace Tag. As defined in *Request channel fields* on page 2-35. See Chapter 11 *System Debug, Trace, and Monitoring*.
MPAM	No	Memory System Performance Resource Partitioning and Monitoring. As defined in *Request channel fields* on page 2-35. See *MPAM* on page 11-355.

Note

This specification does not define a TgtID field for the Snoop request. See *Target ID determination for Snoop Request messages* on page 3-156.

2.2.3 Data fields

Table 2-4 describes the fields associated with a Data packet. Data packets can be sent on the RDAT or WDAT channels. The fields in a Data packet do not affect the transaction structure.

Table 2-4 Data packet fields

Field	Description
QoS	Quality of Service priority. As defined in *Request channel fields* on page 2-35. See Chapter 10 *Quality of Service*.
TgtID	Target ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
SrcID	Source ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
TxnID	Transaction ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
HomeNID	Home Node ID. The Node ID of the target of the CompAck response to be sent from the Requester. See *Details of transaction identifier fields* on page 2-89.
CBusy	Completer Busy. Indicates the current level of activity at the Completer. See *Completer Busy* on page 11-357.
DBID	Data Buffer ID. The ID provided to be used as the TxnID in the response to this message. See *Details of transaction identifier fields* on page 2-89 and *Ordering* on page 2-116.
Opcode	Data opcode. Indicates, for example, if the data packet is related to a Read transaction, a Write transaction, or a Snoop transaction. See *DAT channel opcodes* on page 13-414.
RespErr	Response Error status. Indicates the error status associated with a data transfer. See Chapter 6 *Exclusive Accesses* and *Error response fields* on page 9-325.
Resp	Response status. Indicates the cache line state associated with a data transfer. See *Response types* on page 4-190.
FwdState	Forward State. Indicates the cache line state associated with a data transfer to the Requester from the receiver of the snoop. See *FwdState* on page 13-424.
DataPull	Data Pull. Indicates the inclusion of an implied Read request in the Data response. See *Snoop requests and Data Pull* on page 7-286.
DataSource	Data Source. The value indicates the source of the data in a read Data response. See *Data Source indication* on page 11-350.
CCID	Critical Chunk Identifier. Replicates the address offset of the original transaction request. See *Data transfer* on page 2-136.
DataID	Data Identifier. Provides the address offset of the data provided in the packet. See *Data transfer* on page 2-136.
BE	Byte Enable. For a data write, or data provided in response to a snoop, indicates which bytes are valid. See *Data transfer* on page 2-136.
Data	Data payload. See *Data transfer* on page 2-136.

Table 2-4 Data packet fields (continued)

Field	Description
DataCheck	Data Check. Detects data errors in the DAT packet. See *Data Check* on page 9-338.
Poison	Poison. Indicates that a set of data bytes has previously been corrupted. See *Poison* on page 9-337.
TagOp	Tag Operation. As defined in *Request channel fields* on page 2-35. See *TagOp* on page 13-421.
Tag	Memory Tag. Provides sets of 4-bit tags, each associated with an aligned 16-byte of data. See *Tag* on page 13-421.
TU	Tag Update. Indicates which of the Allocation Tags must be updated. See *TU* on page 13-421.
TraceTag	Trace Tag. As defined in *Request channel fields* on page 2-35. See Chapter 11 *System Debug, Trace, and Monitoring*.
RSVDC	User defined. See *RSVDC* on page 13-427.

2.2.4 Response fields

Table 2-5 describes the fields associated with a Response packet. The fields in a Response packet do not affect the transaction structure.

Table 2-5 Response packet fields

Field	Description
QoS	Quality of Service priority. As defined in *Request channel fields* on page 2-35. See Chapter 10 *Quality of Service*.
TgtID	Target ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
SrcID	Source ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
TxnID	Transaction ID. As defined in *Request channel fields* on page 2-35. See *Details of transaction identifier fields* on page 2-89.
CBusy	Completer Busy. As defined in *Data fields* on page 2-39. See *Completer Busy* on page 11-357.
DBID	Data Buffer ID. As defined in *Data packet fields* on page 2-39. See *Details of transaction identifier fields* on page 2-89 and *Ordering* on page 2-116.
PGroupID	Persistence Group ID. As defined in *Request channel fields* on page 2-35. See *PGroupID* on page 13-406.
StashGroupID	Stash Group ID. As defined in *Request channel fields* on page 2-35. See *StashGroupID* on page 13-408.
GroupIDExt	GroupID Extension. As defined in *Request channel fields* on page 2-35. See *GroupIDExt* on page 13-406.
PCrdType	Protocol Credit Type. See *PCrdType* on page 2-149.
Opcode	Response opcode. Specifies the response type. See *RSP channel opcodes* on page 13-412.
RespErr	Response Error status. As defined in *Data packet fields* on page 2-39. See Chapter 6 *Exclusive Accesses* and *Error response fields* on page 9-325.
Resp	Response status. As defined in *Data packet fields* on page 2-39. See *Response types* on page 4-190.
FwdState	Forward State. As defined in *Data packet fields* on page 2-39. See *FwdState* on page 13-424.
DataPull	Data Pull. As defined in *Data packet fields* on page 2-39. See *Snoop requests and Data Pull* on page 7-286.
TagOp	Tag Operation. As defined in *Request channel fields* on page 2-35. See *TagOp* on page 13-421.
TagGroupID	TagGroupID. As defined in *Request channel fields* on page 2-35. See *TagGroupID* on page 13-421.
TraceTag	Trace Tag. As defined in *Request channel fields* on page 2-35. See Chapter 11 *System Debug, Trace, and Monitoring*.

2.3 Transaction structure

This section describes the structure of the transactions described in *Transaction classification* on page 1-25 together with the channel usage.

Where sufficient, the structure of a transaction is described as seen at a single interface.

The transaction types presented in this section are:

- Request transactions without a Retry.
- Request transactions with a Retry.
- Snoop transactions.

For a Request transaction to complete, a Snoop transaction might be required. However, such dependencies are not visible at the Requester, so these two transaction types are generally presented separately. See Chapter 5 *Interconnect Protocol Flows* for examples of how Request and Snoop flows are related.

All transaction types, except PCrdReturn and PrefetchTgt can have a Retry sequence at the start of the transaction. For ease of presentation, the Retry sequence is described separately. See *Transaction Retry sequence* on page 2-82.

2.3.1 Request transaction structure

The Request transaction structure is described in the following groupings:

- *Snoopable Reads excluding ReadOnce**.
- *ReadNoSnp, ReadOnce, ReadOnceCleanInvalid, ReadOnceMakeInvalid* on page 2-47.
- *Reads with separate Non-data and Data-only responses* on page 2-53.
- *Dataless* on page 2-58.
- *Writes* on page 2-64.
- *Atomics* on page 2-78.
- *DVM* on page 2-80.
- *PrefetchTgt* on page 2-81.

Snoopable Reads excluding ReadOnce*

The Snoopable Read transactions excluding ReadOnce* are:

- ReadClean.
- ReadNotSharedDirty.
- ReadShared.
- ReadUnique.
- ReadPreferUnique.
- MakeReadUnique.

The Snoopable Read transactions described in this section are used by a fully coherent Requester (RN-F) to carry out a read when the snooping of other Snoopable Requesters (RN-Fs) is required.

The transaction structures for ReadOnce* transactions, which are also Snoopable transactions, are described along with ReadNoSnp in subsequent sections of this chapter.

Transaction structure with DMT

A Snoopable Read transaction with DMT is used when the data can be sent directly from the Slave to the original Requester. The progress of the Snoopable Read transaction with DMT is as follows:

1. The Requester sends a Snoopable Read request on the REQ channel:
    - ReadClean.
    - ReadNotSharedDirty.
    - ReadShared.
    - ReadUnique.
    - ReadPreferUnique.
    - MakeReadUnique.
2. The ICN sends a ReadNoSnp request to SN on the REQ channel.
3. The SN, as Completer, forwards the read data and any associated transaction response with the CompData opcode directly to the Requester on the RDAT channel.

    The read data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. Because the transaction request ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

Figure 2-1 shows the transaction structure.

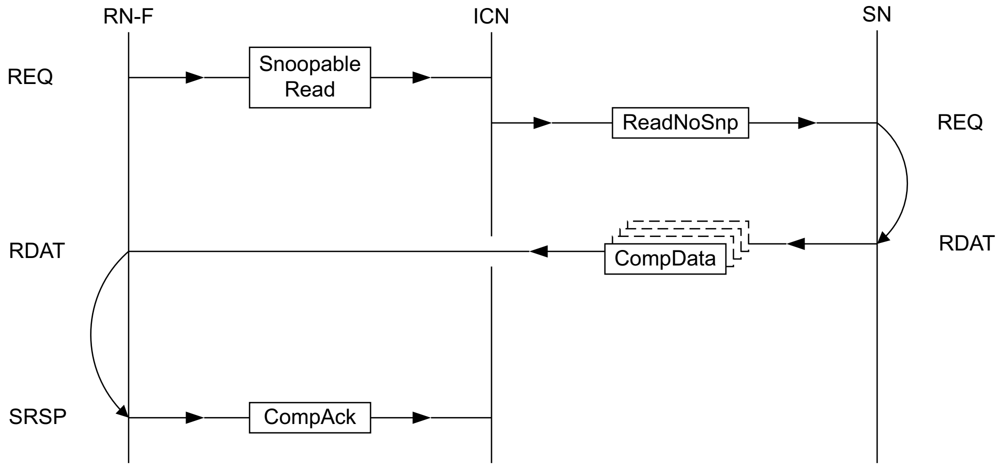


Figure 2-1 Snoopable Read DMT structure

The Request/Response rules are:

- CompData must only be sent by the Completer after the associated request is received.
- The Requester must only send CompAck after at least one CompData packet is received.

The Snoopable Read transaction that Figure 2-1 shows can include a separate Comp and Data response instead of the CompData response. The transaction structure for reads with separate Comp and Data response is described in *Reads with separate Non-data and Data-only responses* on page 2-53.

DMT restrictions

The following restrictions apply to DMT transactions:

- A Requester can reuse the TxnID only after all the responses that could use the TxnID have been returned.
- Home must wait to send a DMT request to SN-F until it is guaranteed that all the following applicable conditions are true:
  - A Snoop request does not need to be sent.
  - If a Snoop request is sent, then the Snoop response is received without a Dirty copy of the cache line being returned.
  - If the Snoop response returns a partial Dirty copy of the cache line, then the DMT can only be sent if the partial data is written to SN-F and a completion for the write is received.
  - If the snoop is a Forwarding type snoop, then it does not result in the cache line being forwarded to the Requester.
- This flow is conditional in MakeReadUnique, it is only applicable when the Home determines that the Requester has lost the cache line.

Note

Home can enable DMT in combination with DCT but must wait for the DCT response to be received before sending the DMT request to SN-F.

Transaction structure with DCT

A Snoopable Read transaction with DCT is used when the data is to be sent directly from the Snooped RN-F to the original Requester. The progress of the Snoopable read transaction with DCT is as follows:

1. The Requester sends a Snoopable Read request on the REQ channel:
   - ReadClean.
   - ReadNotSharedDirty.
   - ReadShared.
   - ReadUnique.
   - ReadPreferUnique.
   - MakeReadUnique.
2. The ICN sends a Snp[*]Fwd request to the RN-F on the SNP channel.
3. The RN-F as the Completer forwards the read data and any associated transaction response to RN with the CompData opcode on the DAT channel.

   The data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. The RN-F also forwards a SnpRespFwded response to the ICN on the SRSP channel to indicate that read data was forwarded to the Requester.
5. Because the transaction request ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

Figure 2-2 shows the transactions structure.

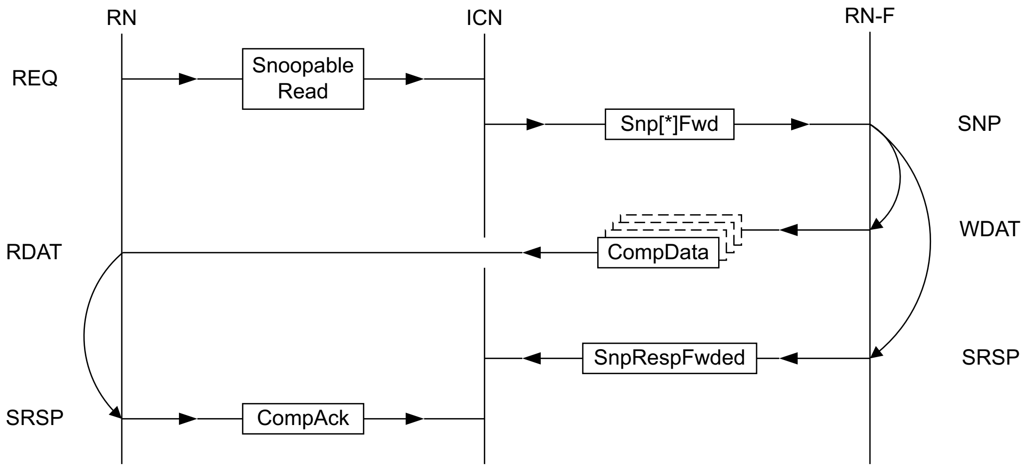


Figure 2-2 Snoopable Read DCT structure

The Request/Response rules are:

- CompData must only be sent by the Completer after the associated request is received.
- CompAck can be sent as soon as the first Data packet of read data has been received.

Transaction structure without Direct Data Transfer

This section shows the Read transaction structure without DMT or DCT.

The progress of a Snoopable Read transaction without Direct Data Transfer, from the Requester perspective, is identical to a Snoopable Read transaction with Direct Data Transfer and is as follows:

1. The Requester sends a Snoopable Read request on the REQ channel:
    - ReadClean.
    - ReadNotSharedDirty.
    - ReadShared.
    - ReadUnique.
    - ReadPreferUnique.
    - MakeReadUnique.
2. The Completer returns the read data and any associated transaction response with the CompData opcode on the RDAT channel.

    The read data can be sent using multiple transfers. See *Data transfer* on page 2-136.
3. Because the ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

    CompAck can be sent as soon as the first Data packet of read data has been received.

    Separate Comp and Data responses are possible. See *Reads with separate Non-data and Data-only responses* on page 2-53.

Figure 2-3 shows the transaction structure.

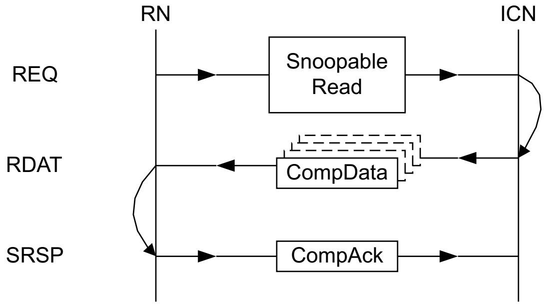


Figure 2-3 Snoopable Read structure without Direct Data Transfer

The Request/Response rules are:

- CompData must only be sent by the Completer after the associated request is received.
- If the data being sent is received from either a Slave or snooped Agent then the ICN is permitted to forward the CompData packets as soon as it receives the first packet from the Slave or snooped Agent.
- CompAck can be sent as soon as the first Data packet of read data has been received.

ReadNoSnp, ReadOnce, ReadOnceCleanInvalid, ReadOnceMakeInvalid

A ReadNoSnp transaction is used to carry out a read when the snooping of other masters is not required.

Data obtained by ReadNoSnp either comes directly from the Slave Node or via the interconnect.

A ReadOnce, ReadOnceCleanInvalid, and ReadOnceMakeInvalid transaction is used to carry out a read when the snooping of other masters is required but the Requester is not going to allocate the cache line in its own cache.

Note

ReadOnce, ReadOnceCleanInvalid, and ReadOnceMakeInvalid obtain a snapshot of the coherent data value. If a Requester holds this value in a local buffer or cache, the data value will no longer be coherent.

In the remainder of this section ReadOnce* represents the three transaction types, ReadOnce, ReadOnceCleanInvalid, and ReadOnceMakeInvalid.

Data obtained by ReadOnce* either comes directly from the Slave Node or a peer Request Node, or via the interconnect.

ReadNoSnp and ReadOnce* transactions can optionally have an ordering requirement. For transactions that require ordering, the Home must ensure that a transaction is observable before taking any action that could make a later ordered transaction observable. Such a transaction can include a ReadReceipt response from the Home to the Requester.

ReadNoSnp and ReadOnce* transactions can optionally set the ExpCompAck field, indicating that the transaction will include a CompAck response. The use of a CompAck response is not functionally required for ReadNoSnp and ReadOnce* transactions, as the RN issuing the transaction will not hold a copy of the cache line. However, use of CompAck permits the use of DMT and separate Comp and Data responses in some cases.

ReadNoSnp and ReadOnce* structure with DMT

The progress of a ReadNoSnp and a ReadOnce* transaction with DMT is as follows:

1. Requester sends a request with the ReadNoSnp or ReadOnce* opcode on the REQ channel.
2. If the request Order field indicates that ordering is required, then a ReadReceipt response must be returned on the CRSP channel when order has been established.
3. ICN sends a ReadNoSnp request to SN on the REQ channel.
4. SN returns a ReadReceipt response to ICN on the CRSP channel if the ReadNoSnp request has Order[1:0] set to `0b01`.
5. SN, as Completer, returns the read data and any associated transaction response with the CompData opcode directly to the Requester on the RDAT channel.

   The read data can be sent using multiple transfers. See *Data transfer* on page 2-136.
6. If the transaction request ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

   CompAck can be sent as soon as the first Data packet of read data has been received.

Figure 2-4 shows the transaction structure.

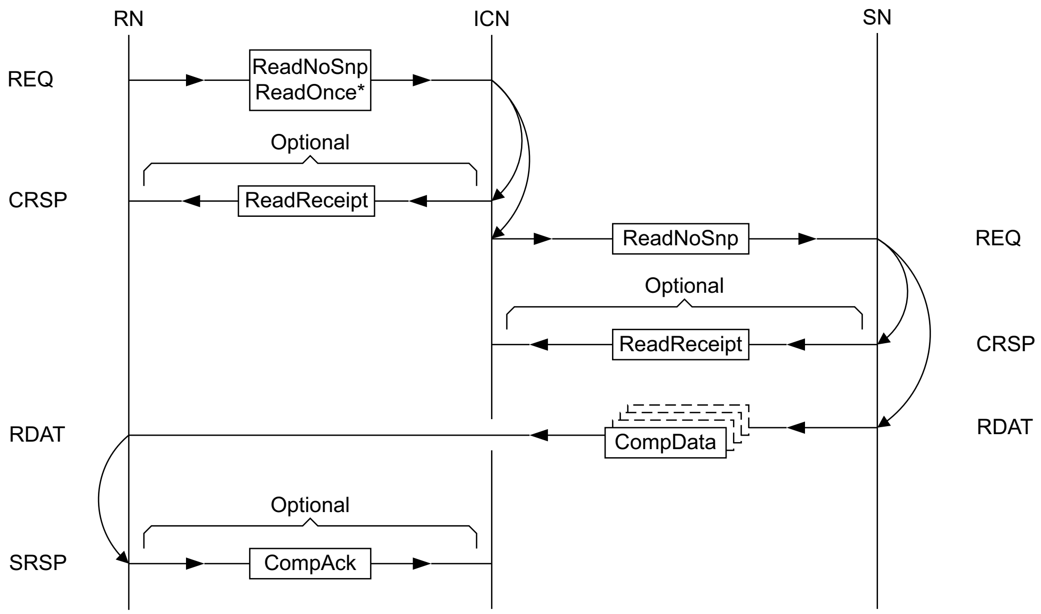


Figure 2-4 ReadNoSnp and ReadOnce* DMT structure

The life time at the Home of ReadNoSnp and ReadOnce* transactions that use DMT can be reduced by using the ReadReceipt response from the Slave as a read received acknowledgment to deallocate the transaction at Home. The ReadReceipt from SN, marked as optional in Figure 2-4, becomes required when used to early deallocate the request at the Home when used instead of CompAck from the Requester, which typically comes later, to deallocate the request. See *ReadOnce* and ReadNoSnp with early Home deallocation* on page 5-246 for an example of this type of flow. See Table 2-6 on page 2-52 for the relationship between DMT and the use of Read Received Ack, in the form of a ReadReceipt, and CompAck and Ordering requirements in ReadOnce* and ReadNoSnp transactions.

The following requirements apply to this type of lifetime reduction transaction:

- Home must set Order[1:0] to the value `0b01` in the Read request to the Slave Node.
- For a Request with Order[1:0] set to the value `0b01` the Slave must send a ReadReceipt to acknowledge the Read request when it can guarantee that the request is accepted and that it will not send a RetryAck response.
- Home is permitted to deallocate the request after receiving the ReadReceipt without waiting for a CompAck if the requests from RN do not have an ordering requirement.
- Home is permitted to receive CompAck even after the request is deallocated.

ReadOnce* structure with DCT

The progress of a ReadOnce* transaction with DCT is as follows:

1. Requester sends a request with the ReadOnce* opcode on the REQ channel.
2. ICN sends a Snp[*]Fwd request to RN-F on the SNP channel.
3. RN-F, as the Completer, forwards the read data and any associated transaction response to RN with the CompData opcode on the DAT channel.

   The read data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. RN-F also forwards a SnpRespFwded response to ICN on the SRSP channel to indicate that read data was forwarded to the Requester. Alternatively, the response to the ICN can include data and will be a SnpRespDataFwded response.
5. If the transaction request ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

   CompAck can be sent as soon as the first Data packet of read data has been received.

Figure 2-5 shows the transaction structure.

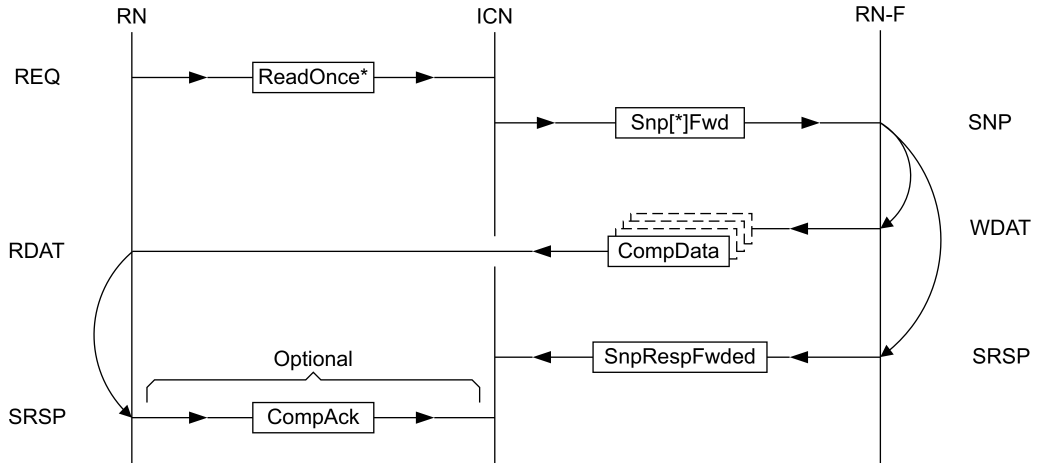


Figure 2-5 ReadOnce* DCT structure

ReadNoSnp and ReadOnce structure without Direct Data Transfer*

This section shows a Read transaction structure without DMT or DCT.

The progress of a ReadNoSnp and a ReadOnce* transaction without Direct Data Transfer, from the Requester perspective, is identical to a ReadNoSnp and ReadOnce* transaction with Direct Data Transfer and is as follows:

1. Requester sends a request with the ReadNoSnp or ReadOnce* opcode on the REQ channel.
2. If the request Order field indicates that ordering is required then a ReadReceipt response must be returned on the CRSP channel when order has been established.
3. Completer returns the read data and any associated transaction response with the CompData opcode on the RDAT channel.

   The read data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. If the transaction request ExpCompAck bit is set, the Requester must return an acknowledgement, using the CompAck opcode on the SRSP channel to indicate that the transaction has completed.

   CompAck can be sent as soon as the first Data packet of read data has been received.

Figure 2-6 shows the transaction structure.

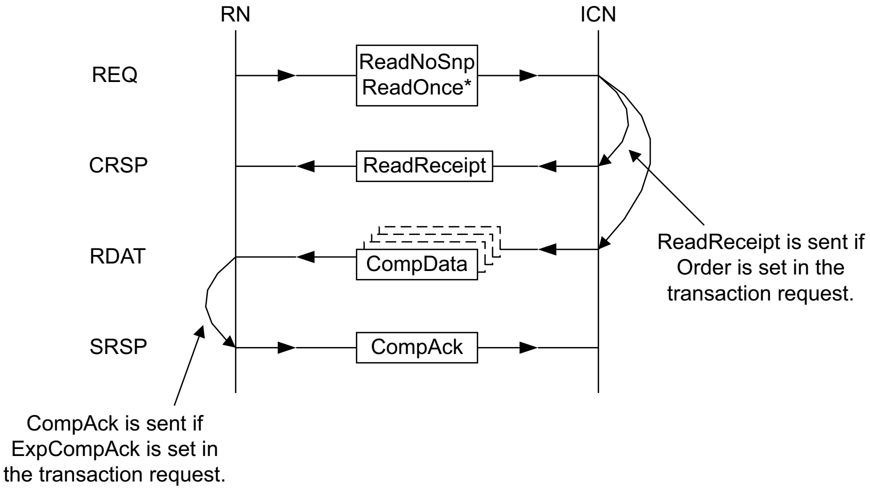


Figure 2-6 ReadNoSnp and ReadOnce* structure without Direct Data Transfer

The Request/Response rules are:

- ReadReceipt, if part of the transaction:
  - Must only be sent by the Completer after the associated request is received.
  - Typically is sent by the Completer before CompData. However it is permitted to be sent after CompData.
  - Typically is received by the Requester before CompData. However it is permitted to be received after CompData.
- CompData must only be sent by the Completer after the associated request is received.
- If the data being sent is received from either a Slave or snooped Agent then the ICN is permitted to forward the CompData packets as soon as it receives the first packet from the Slave or snooped Agent.
- If CompAck is part of the transaction:
  - The Requester must only send CompAck after at least one CompData packet is received.
  - The Requester is permitted, but not required, to wait for the ReadReceipt before sending CompAck.
  - The Completer is not permitted to wait for the CompAck before sending the ReadReceipt.

Summary of impact of ordering and CompAck rules

Table 2-6 shows the use of DMT and DCT, with different combinations of ordering and CompAck for ReadNoSnp and ReadOnce* from an RN.

Table 2-6 Permitted DMT and DCT for ReadNoSnp and ReadOnce* from an RN

Order[1:0]	ExpCompAck	DMT	DCT	Notes
00	0	Y	Y	Home does not need to be notified of transaction completion. For DMT, Home must obtain the Request Accepted response from SN to ensure the request to SN is not given a RetryAck response.
	1	Y	Y	Home does not need to be notified of transaction completion. For DMT, Home can ensure the request to SN is not given a RetryAck response by either obtaining the Request Accepted response from SN or waiting for the CompAck response.
01	-	-	-	Not permitted.
10 11	0	N	Y	For DCT, Home uses the SnpRespFwded or SnpRespDataFwded snoop response to determine transaction completion.
	1	Y	Y	For DMT, Home uses the CompAck response to determine transaction completion. For DCT, Home uses the SnpRespFwd or SnpRespDataFwd snoop response to determine transaction completion.

Reads with separate Non-data and Data-only responses

This specification supports a separate Non-data response from Home and a Data-only response from either Home or Slave for all Read* transactions.

This feature does not apply to:

- Atomic transactions.
- Exclusive transactions.
- Ordered ReadNoSnp and ReadOnce* transactions without CompAck.

The progress of a Read* with separate Comp and Data responses is as follows:

- Comp from Home and Data from Slave:
    1. Requester sends a request with the Read* opcode on the REQ channel.
    2. Home sends a RespSepData response to the Requester on the CRSP channel.
        - This tells the Requester that the transaction has been ordered at Home with respect to requests to the same address, and that Data for the transaction will be provided in a separate Data response.
    3. Requester sends CompAck acknowledgement to ICN on the SRSP channel.

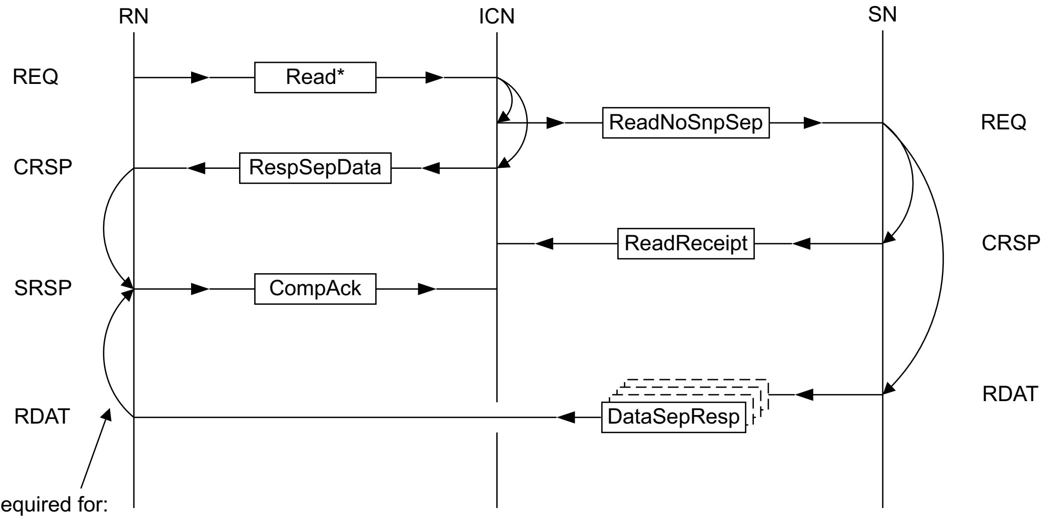
       For ordered transactions, CompAck must be sent only after the first Data response packet has been received.
    4. ICN sends ReadNoSnpSep request to the SN on the REQ channel.
        - This tells the Slave, as Completer, to send a DataSepResp response.
    5. SN sends ReadReceipt to Home on the CSRP channel.
        - This tells the Home that the request to the Slave will be completed and ensures that the SN will not respond with RetryAck for that request.
    6. SN, as Completer, sends DataSepResp to the Requester returning the read data.

       For ReadOnce and ReadNoSnp requests with an Order requirement, the Home determines that the request has been completed by receiving a CompAck response.

- Comp and Data from Home:
    1. Requester sends a request with the Read* opcode on the REQ channel.
    2. Home sends a RespSepData response to the Requester on the CRSP channel.
        - This tells the Requester that the transaction has been ordered at Home with respect to the same address, and that Data for the transaction will be provided in a separate Data response.
    3. Home, as Completer, sends DataSepResp to the Requester returning the read data
    4. Requester sends CompAck acknowledgement to Home on the SRSP channel.

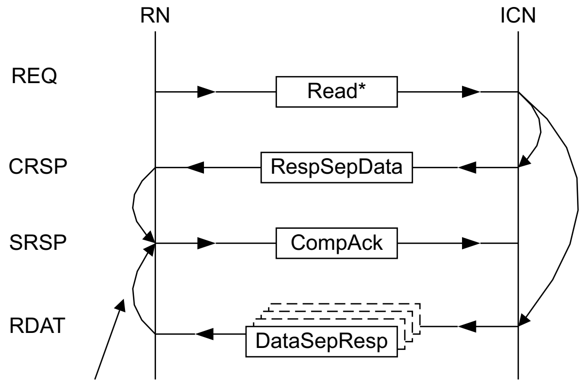
Figure 2-7 on page 2-54 shows the transaction structure.

Comp from Home and Data from Slave



Required for:
ReadOnce and ReadNoSnp
if Order field is set.
Permitted, but not required,
for all other Reads

Comp and Data from Home



Required for:
ReadOnce and ReadNoSnp
if Order field is set.
Permitted, but not required,
for all other Reads

Figure 2-7 Read* DMT structure with separate Non-data and Data-only responses

Note

As Figure 2-7 shows, the CompAck acknowledgement sent by the Requester to ICN is permitted, but not required, to wait for DataSepResp except in the case of an ordered ReadOnce and ReadNoSnp request with CompAck, whereas it is required to wait for RespSepData.

The Request and Response rules are:

- DataSepResp and RespSepData must only be sent by the Completer after the associated request is received. RespSepData is permitted from the Home only. DataSepResp is permitted to be sent by the Slave or Home.
- CompAck acknowledgement must only be sent by the Requester after the RespSepData response is received:
  - For Non-ordered requests, it is permitted, but not required, for a Requester to wait for any or all data to be received before sending a CompAck acknowledgement.
  - For ReadOnce and ReadNoSnp requests with an Order requirement, the Requester is required to wait for at least one Data response to be received before sending a CompAck acknowledgement.
- It is required that the Completer must not wait for CompAck before sending all Data packets.
- ReadNoSnpSep must only be sent by the Home to the Slave after the associated request is received, and Snoop responses are received for all snoops that are required to be sent. The Home is permitted to speculatively send ReadNoSnpSep to the Slave, without waiting for all Snoop responses, if ReturnNID is set to the Home Node ID.
- The Slave must send the ReadReceipt response to the Home only after receiving ReadNoSnpSep and ensuring that it will not respond with RetryAck for that request.
- The ReadReceipt from the Slave is sufficient to indicate to the Home that the request to the Slave will be completed.

  A separate Comp response from the Slave to the Home is not required and must not be sent. For ReadOnce and ReadNoSnp requests with an Order requirement, the Home determines that the request has been completed by receiving a CompAck response.
- For ReadOnce and ReadNoSnp transactions, with an Order requirement, that include separate Comp and Data responses, the Home must not send a ReadReceipt response. The RespSepData response from Home includes an implied ReadReceipt.

In all cases, where a separate Data response and Home response can be used, it is also permitted to use a combined CompData response.

Table 2-7 on page 2-56 shows the expected and permitted use of separate Non-data and Data-only responses, or the combined CompData response, with different combinations of Order and ExpCompAck values for ReadNoSnp and ReadOnce*.

Table 2-7 Use of separate Comp and Data responses with different combinations of Order and ExpCompAck for ReadNoSnp and ReadOnce*

Transaction	Order[1:0]	ExpCompAck	Notes
ReadNoSnp ReadOnce*	00	0	The options to provide responses for the transaction are: • A single CompData response. This can be: — From the Home. — From a snooped RN-F, when DCT is used. — From the Slave, when DMT is used. Home must request a ReadReceipt from the Slave. • A separate RespSepData and DataSepResp response. RespSepData from the Home. DataSepResp can be: — From the Home. — From the Slave. Home must request a ReadReceipt from the Slave. A ReadReceipt from Home to Requester is not used. A CompAck from Requester to Home is not used.
		1	The options to provide responses for the transaction are: • A single CompData response. This can be: — From the Home. — From a snooped RN-F, when DCT is used. — From the Slave, when DMT is used. Home can optionally request a ReadReceipt from the Slave. • A separate RespSepData and DataSepResp response. RespSepData is always from the Home. DataSepResp can be: — From the Home. — From the Slave. Home can optionally request a ReadReceipt from the Slave. A ReadReceipt from Home to Requester is not used. A CompAck is sent from Requester to Home only after: • CompData is received. • RespSepData is received. It is permitted to also wait for the corresponding DataSepResp response before sending CompAck.
	01	-	This is not permitted for transactions from Requester to Home.

Table 2-7 Use of separate Comp and Data responses with different combinations of Order and ExpCompAck for ReadNoSnp and ReadOnce* (continued)

Transaction	Order[1:0]	ExpCompAck	Notes
ReadNoSnp ReadOnce*	10 11	0	The options to provide responses for the transaction are: • A single CompData response is provided. This can be: — From the Home. — From a snooped RN-F, when DCT is used. — DMT is not permitted. • A separate RespSepData and DataSepResp response is not permitted. A ReadReceipt from Home to Requester is used. A CompAck from Requester to Home is not used.
		1	The options to provide responses for the transaction are: • A single CompData response. This can be: — From the Home. — From a snooped RN-F, when DCT is used. — From the Slave, when DMT is used. In this case, the Home can optionally request a ReadReceipt from the Slave. • A separate RespSepData and DataSepResp response. RespSepData is always from the Home. DataSepResp can be: — From the Home. — From the Slave. The Home can optionally request a ReadReceipt from the Slave. A ReadReceipt from the Home to Requester depends on: • If a single CompData response is used, then a ReadReceipt is sent. • If a separate RespSepData and DataSepResp response is used, then no ReadReceipt is sent. RespSepData is used to indicate the same meaning as ReadReceipt. A CompAck is sent from Requester to Home only after: • CompData is received. • Both RespSepData response and DataSepResp response are received.

Dataless

The Dataless transactions can be divided into *Non-Cache Maintenance* (Non-CMO) and *Cache Maintenance* (CMO) transactions.

Non-CMO transactions

The Non-CMO transactions are:

- CleanUnique.
- MakeUnique.
- Evict.
- StashOnceUnique, StashOnceSepUnique.
- StashOnceShared, StashOnceSepShared.

These transactions serve a number of functions such as:

- Obtaining permission to store to a cache.
- Updating the state of a snoop filter.
- Moving data closer to the point of expected future use.

The following structure and rules apply to CleanUnique, MakeUnique and Evict transactions. For the StashOnce and StashOnceSep transaction structure and rules see *StashOnce and StashOnceSep transactions* on page 2-59.

Only a single response is given for Non-CMO Dataless transactions.

The progress of a Non-CMO Dataless transaction from the Requester perspective is as follows:

1. The Requester sends a CleanUnique, MakeUnique, or Evict request on the REQ channel.
2. The Completer returns a Comp response on the CRSP channel.
3. If the transaction request ExpCompAck field is set, the Requester must return an acknowledgement that the transaction has completed with the CompAck opcode on the SRSP channel:
    - ExpCompAck must be asserted for:
        - — CleanUnique.
        - — MakeUnique.
    - ExCompAck must not be asserted for:
        - — Evict.

Figure 2-8 shows the transaction structure.

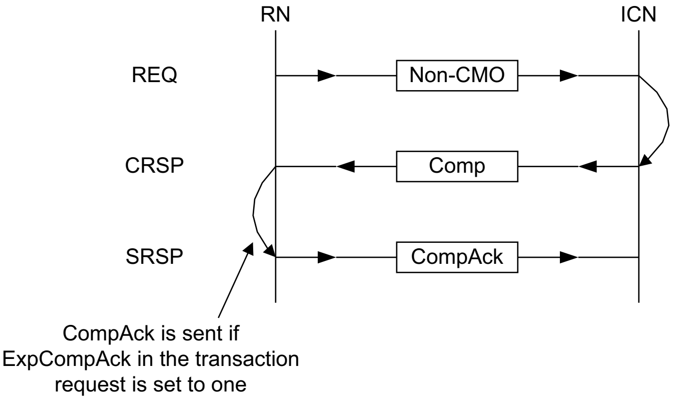


Figure 2-8 Non-CMO dataless transaction structure

The Request/Response rules are:

- Comp must only be sent by the Completer after the associated request is received.
- CompAck must only be sent by the Requester after the associated Comp is received.

StashOnce and StashOnceSep transactions

In this specification, unless specifically stated otherwise, StashOnceSep represents both StashOnceSepUnique and StashOnceSepShared.

The characteristics of StashOnceSepUnique and StashOnceSepShared are the same as for the corresponding Non-CMO StashOnceUnique and StashOnceShared transactions except that they include two responses to the Requester instead of just a Comp.

The progress of a StashOnce and StashOnceSep transaction from the Requester perspective is as follows:

1. The Requester (RN) sends a request on the REQ channel.
2. The Home (ICN) returns a Comp response on the CRSP channel.
3. For a StashOnceSep request the Home also returns a StashDone response on the CRSP channel. The Home is permitted to send a combined CompStashDone response.
4. In addition to the responses to the Requester, the Home optionally sends a SnpStash snoop to the target RN-F on the SNP channel.
5. The target RN-F sends a SnpResp response either with or without a Data Pull indication on the SRSP channel.
6. If the response from the target RN-F includes a DataPull request then, the Home sends a CompData response or DataSepResp and RespSepData to the target RN-F on the DAT channel.

   The Home is permitted to use DMT or DCT to forward the data to the target RN-F.
7. The target RN-F after receiving the CompData response sends a CompAck to Home on the SRSP channel.

Figure 2-9 on page 2-60 shows the transaction structure with and without Data Pull.

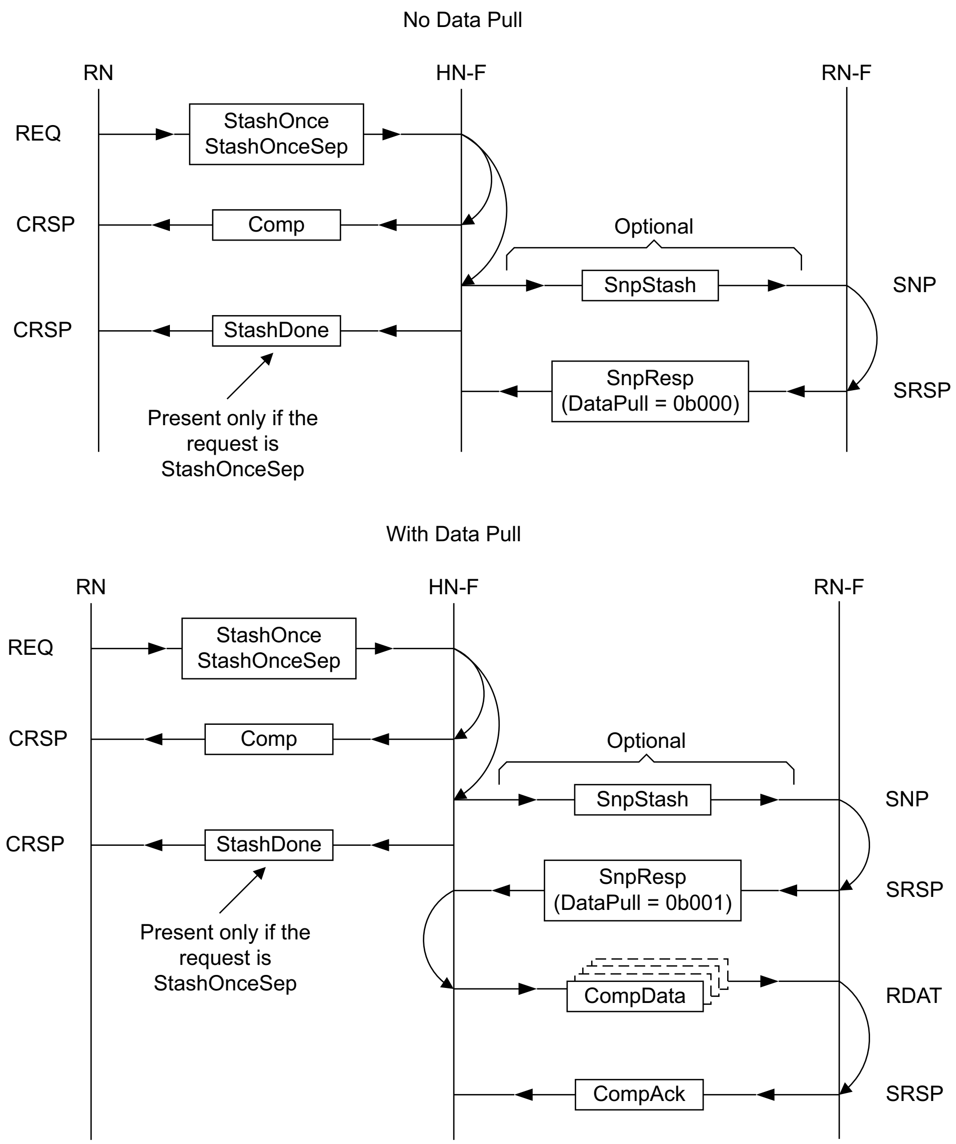


Figure 2-9 StashOnce and StashOnceSep transaction structure

The request/response rules are:

- ExCompAck must not be asserted.
- The Completer must send a Comp response only after receiving the StashOnce or StashOnceSep request.
- The Comp response for a StashOnce request must come from the PoC.
  - — The PoC must send a Comp response only after receiving the StashOnce request and establishing the guarantee that the request is ordered at the Home.
- The Home sends a SnpStash snoop to the target only if the Home accepts the Stash hint and is able to process the Data Pull generated Read request from the Snoopee. Otherwise, the Home concludes the transaction and does not send the SnpStash snoop.
- The Snoopee in response to a SnpStash must send a SnpResp response along with an indication of whether it accepted the Stash hint and is including a Read request embedded as a DataPull in the Snoop response.

- Specific to the StashOnceSep request:
    - The Completer must send a StashDone response only after receiving the StashOnceSep request and establishing the guarantee that the request is ordered at the Home. The Completer must be a PoC.
    - The Completer is permitted to combine Comp and StashDone responses into a single CompStashDone response.
    - The Requester, on receiving the Comp response, is permitted to release some request resources, while keeping track of the number of outstanding StashDone responses. This count of the number of outstanding StashDone responses can be done per Stash Group, using the Requester defined Stash Group ID specified by the StashGroupID field value.

CMO transactions

The CMO transactions are:

- CleanShared.
- CleanSharedPersist.
- CleanSharedPersistSep.
- CleanInvalid.
- MakeInvalid.

These transactions are used to perform cache maintenance on different levels of caches in the system.

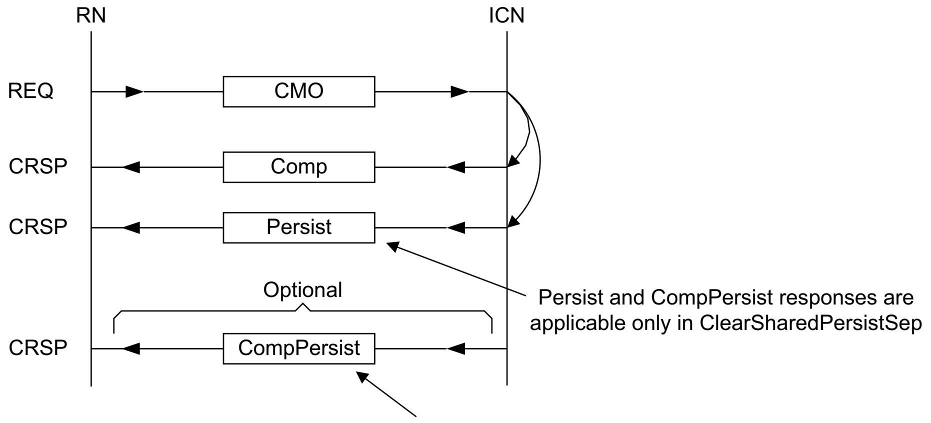
With the exception of CleanSharedPersistSep, only a single response is given for CMO transactions.

The CleanSharedPersistSep transaction can include one or two responses.

The progress of a CMO Dataless transaction from the Requester perspective is as follows:

1. The Requester sends a request on the REQ channel.
2. The Completer returns a Comp response on the CRSP channel.
3. For the CleanSharedPersistSep transaction the Completer also returns a Persist response to the Requester. The Completer is permitted to combine the Comp and Persist responses into a single CompPersist response.

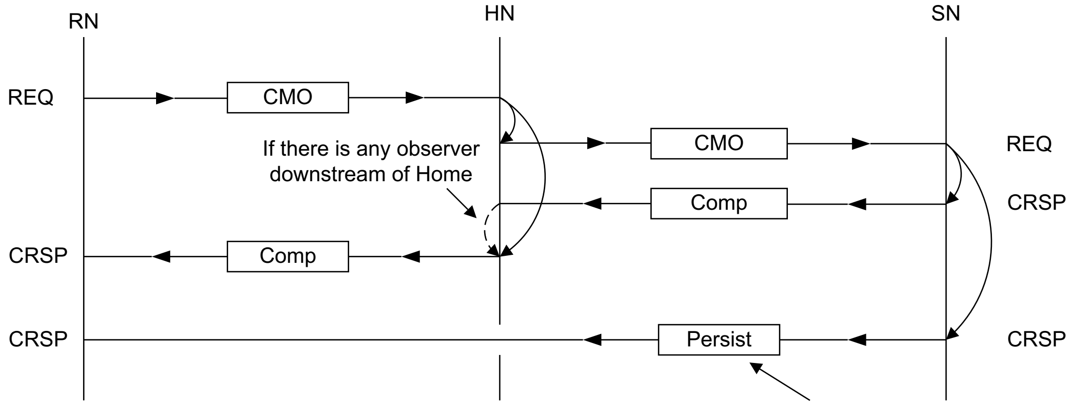
Figure 2-10 on page 2-62 shows the transaction structure and includes the transaction structure when the request is propagated downstream of the ICN.



Persist and CompPersist responses are applicable only in ClearSharedPersistSep

CompPersist can replace separate Comp and Persist responses

Request propagated downstream of the ICN



If there is any observer downstream of Home

Only if request is CleanSharedPersistSep

Figure 2-10 CMO transaction structure

The Request/Response rules are:

- Comp must only be sent by the Completer after the associated request is received.
- If there is an observer downstream of Home, then the Home must send the request downstream and also wait for Comp from downstream before sending the Comp response to the Requester.
- If there is any observer downstream of Home, then a Comp must be sent only after the Slave Node can guarantee that all earlier writes to the same address can be observed by all observers in the system.
- If there is no observer downstream of Home then:
  - — The Home is permitted, but not required, to send the request downstream.
  - — The Home is permitted, but not required, to wait for the Comp from downstream before sending the Comp response to the Requester.

If the Point of Persistence is downstream of the Home then, for CleanSharedPersistSep:

- The Home must send the request downstream.
- The Slave Node must send back a Comp response only after it guarantees that it has accepted the request and will not send a RetryAck response.
- The Slave Node must also send a Persist response to either the Requester or the Home when it can guarantee that all previous writes to the same address, to non-volatile memory, have been made persistent and the address location will contain the updated value even after the removal of power.
- If the Home receives a Persist response from downstream, then it must forward the Persist response to the Requester:
  - The Slave is permitted to send a combined CompPersist response to the Home.
  - The Home is permitted, but not expected, to wait for the Persist response from the Slave and then send a combined CompPersist response to the Requester instead of sending an earlier separate Comp response.
- If the target is volatile memory, then the Persist response can be given immediately. Such a situation must not return an error.

If the Home is the Point of Persistence then, for CleanSharedPersistSep:

- The Home is permitted to not forward the CleanSharedPersistSep request to the Slave. When Home decides not to forward the CleanSharedPersistSep request it must return a Persist response to the Requester.
- When Home is sending a Persist response it is permitted, but not required, to combine it with Comp and send a single CompPersist response to the Requester.

Writes

Write transactions include the following:

- WriteNoSnp.
- WriteNoSnpZero.
- WriteUnique.
- WriteUniqueZero.
- WriteBack.
- WriteCleanFull.
- WriteEvictFull.
- WriteEvictOrEvict.

Non-CopyBack

The Non-CopyBack transactions are:

- WriteNoSnp, WriteNoSnpZero.
- WriteUnique, WriteUniqueZero.

A WriteNoSnp and WriteNoSnpZero transaction is used to carry out a store where the snooping of other masters is not required.

A WriteUnique and WriteUniqueZero transaction is used to perform a store to a location when the snooping of other Snoopable Requesters (RN-Fs) might be required to obtain permission to store.

The WriteNoSnpZero and WriteUniqueZero transactions specifically permit the sending of a Write message without data bytes when the full cache line data value is zero. Partials are not permitted.

There is one optional behavior associated with WriteNoSnp and WriteUnique transactions. The behavior is determined by the Order and ExpCompAck fields in the transaction request. See *Streaming Ordered Write transactions* on page 2-124 for details on the use of CompAck to force the order in which requests are observed.

Figure 2-11 on page 2-65 shows the transaction structure options for the WriteNoSnp and WriteUnique transactions.

The progress of the WriteNoSnp and WriteUnique transaction is as follows:

1. The Requester sends a request with the WriteNoSnp or WriteUnique opcode on the REQ channel.
2. The Completer has one of the following options:
    - Return separate responses:
        - Return a DBIDResp or DBIDRespOrd response that provides a data buffer identifier indicating that it can accept the write data for the transaction.
        - Return a Comp response to indicate that the transaction is observable by other Requesters.

      Both responses are returned on the CRSP channel.
    - Return a single combined CompDBIDResp response to indicate:
        - It can accept the write data for the transaction.
        - The transaction is observable by other Requesters.

      The combined response is returned on the CRSP channel.
3. The Requester sends the write data and any associated byte enables with the NonCopyBackWrData or WriteDataCancel opcode on the WDAT channel. The write data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. If the ExpCompAck field is set in the transaction request, then the transaction completes with a CompAck transaction acknowledge. It is permitted for the CompAck response to be combined with the Data response and sent as NCBWrDataCompAck.

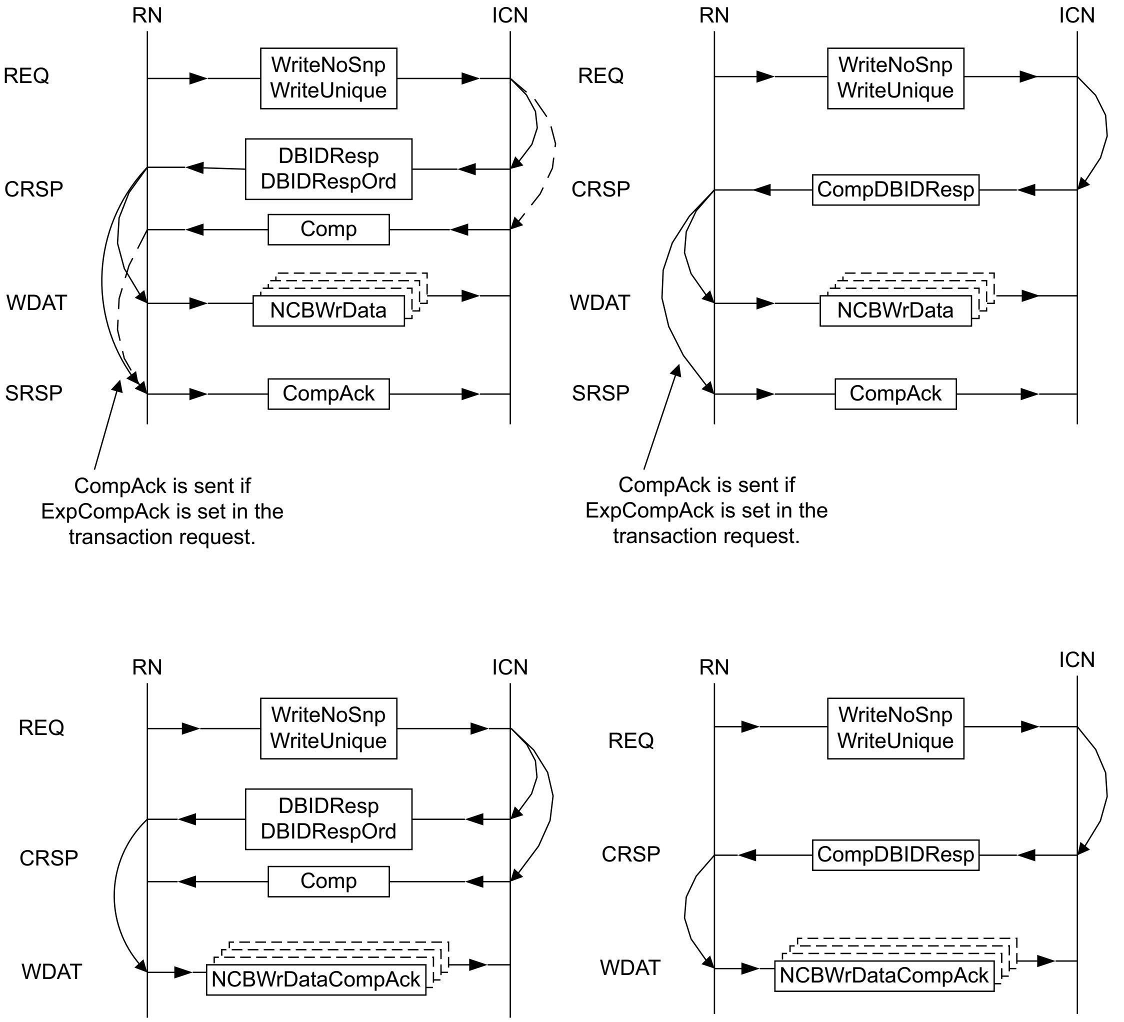


Figure 2-11 WriteNoSnp and WriteUnique transaction structure options

The Request/Response rules are:

The separate DBIDResp* and Comp, or the combined CompDBIDResp, must only be sent by the interconnect after the associated request is received.

WriteData must only be sent by the Requester after DBIDResp or DBIDRespOrd or CompDBIDResp is received.

- If the DBIDResp* and Comp responses are sent separately:
    - Typically, the DBIDResp* is sent by the Completer before Comp. However, it is permitted for DBIDResp* and Comp to be sent in any order.
    - Typically, the DBIDResp* is received by the Requester before Comp. However, it is permitted for DBIDResp* and Comp to arrive in any order.
    - The Requester must send the write data after it has received the DBIDResp* response.
      The Requester must not wait to receive the Comp response before the write data is sent.
    - The Completer is permitted to wait for the WriteData before sending the Comp response.
- If snooping is required, the Home must receive all Snoop responses before sending the Comp or combined CompDBIDResp response.
- If CompAck is required, then:
    - CompAck must only be sent by the Requester after the Comp or DBIDResp or DBIDRespOrd or CompDBIDResp response is received.
    - The Requester that receives a DBIDResp* response first is permitted to send either a combined NCBWrDataCompAck response or separate NCBWrData and CompAck responses.
    - The Requester that receives a Comp response first is permitted to wait for the DBIDResp* response before sending either a combined NCBWrDataCompAck or separate CompAck and NCBWrData responses.
    - The Requester when sending separate responses is permitted to send the WriteData and CompAck in any order.
    - The Requester when sending a combined NCBWrDataCompAck must send the combined response in all data transfers.

Note

Prior to CHI Issue E, the DBIDRespOrd response was not supported.

Figure 2-12 shows the transaction structure options for the WriteNoSnpZero and WriteUniqueZero Non-CopyBack transactions.

The progress of the WriteNoSnpZero and WriteUniqueZero transactions is as follows:

1. The Requester sends a request with the WriteNoSnpZero or WriteUniqueZero opcode on the REQ channel.
2. The Completer has one of the following options:
   - Return separate responses:
     - — Return a DBIDResp or a DBIDRespOrd response.
     - — Return a Comp response.
   - Return a combined response:
     - — Return a CompDBIDResp response.

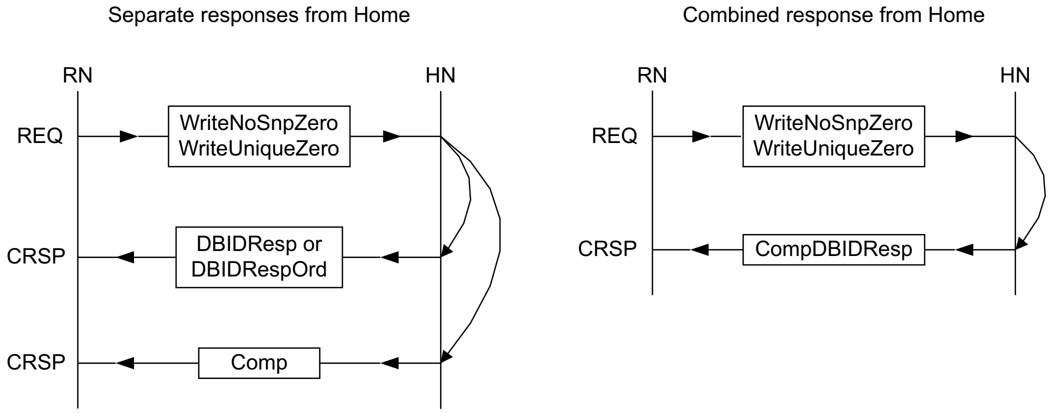


Figure 2-12 WriteNoSnpZero and WriteUniqueZero transaction structure

The Request/Response rules are:

The response to both WriteNoSnpZero and WriteUniqueZero is either DBIDResp* and a Comp or a combined CompDBIDResp.

The separate DBIDResp* and Comp, or the combined CompDBIDResp, must only be sent by the interconnect after the associated request is received.

If snooping is required, the Home must receive all Snoop responses before sending the Comp or CompDBIDResp response.

The expected HN to SN request for writing a zero data value for a cache line is WriteNoSnpZero, instead of WriteNoSnp.

CopyBack

The CopyBack transactions are:

- WriteBackPtl.
- WriteBackFull.
- WriteCleanFull.
- WriteEvictFull.
- WriteEvictOrEvict.

CopyBack transactions, except WriteEvictFull and WriteEvictOrEvict, are used to update main memory or a downstream cache for a coherent location.

A WriteEvictFull or WriteEvictOrEvict transaction is used only to update a downstream cache for a coherent location with a Clean cache line.

A WriteEvictFull or WriteEvictOrEvict transaction must not propagate beyond its Snoop domain.

The progress of a CopyBack transaction, with the exception of WriteEvictOrEvict is:

1. The Requester sends a CopyBack request on the REQ channel.
2. The Completer returns a single combined CompDBIDResp response on the CRSP channel to indicate:
   - It can accept the write data for the transaction.
   - This request will complete before any snoop to the same address is received.
3. After the Requester has received the CompDBIDResp response it sends the write data, and any associated byte enables, with the CopyBackWrData opcode on the WDAT channel. The write data can be sent using multiple transfers. See *Data transfer* on page 2-136.

Figure 2-13 shows the transaction structure.

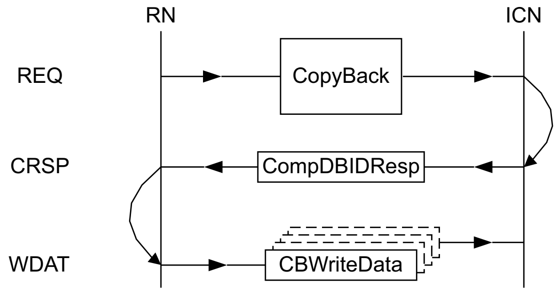


Figure 2-13 CopyBack transaction structure

The Request/Response rules are:

- CompDBIDResp must only be sent by the Completer after the associated request is received.
- WriteData must only be sent by the Requester after the CompDBIDResp response is received.

The progress of a WriteEvictOrEvict transaction is:

1. The Requester sends the request on the REQ channel.
2. The Home, based on its own heuristics, decides to request data from the Requester or not request data.

If Home decides to request data then:

1. Home sends a CompDBIDResp response to the Requester.
2. The Requester after receiving the CompDBIDResp response sends data as a CopyBackWriteData message.

Note

An explicit CompAck message is not sent irrespective of ExpCompAck because a CopyBackWriteData message is treated as an implicit CompAck.

If Home decides not to request data then:

1. Home returns a Comp response to the Requester.
2. The Requester after receiving the Comp response returns a CompAck message.

Figure 2-14 shows the transaction structure.

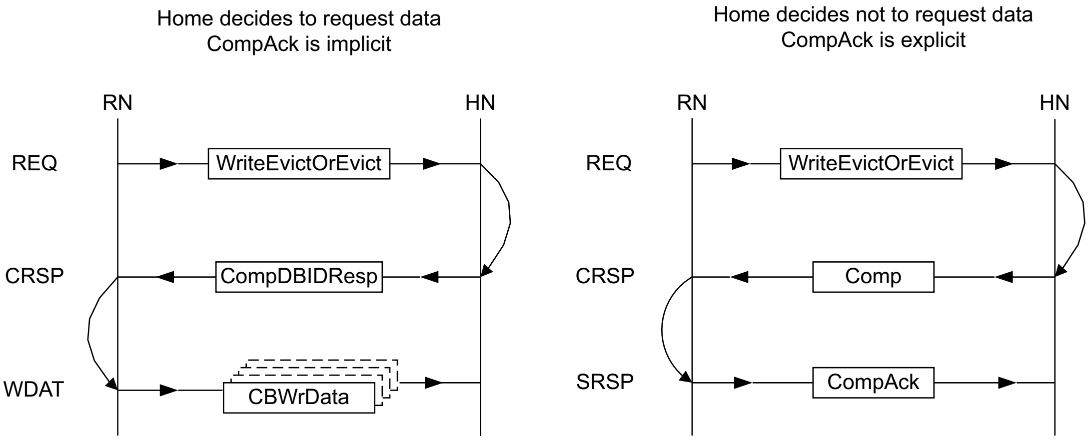


Figure 2-14 WriteEvictOrEvict transaction structure

The Request/Response rules are:

- CompDBIDResp or Comp response must only be sent by the Completer after the associated request is received.
- WriteData must only be sent by the Requester after the CompDBIDResp response is received.
- CompAck must only be sent by the Requester after the Comp response is received.

Direct Write-data Transfer

Direct Write-data Transfer (DWT) allows write data to be passed directly from the Requester to the Slave reducing the use of the DAT channel and removing the need for the Home Nodes to hold a copy of the write data.

DWT is permitted only in writes that are Non-CopyBack.

DWT is not permitted in Non-CopyBack writes that are OWO writes.

DWT support

A DoDWT field is added to the Request channel to support DWT. The DoDWT field is only applicable in WriteNoSnpFull, WriteNoSnpPtl and in Combined Write requests from Home to Slave. See *DoDWT* on page 13-417.

Non-CopyBack Write transaction structure

Figure 2-15 on page 2-71 shows the Non-CopyBack Write transaction structure when DoDWT is set to 1.

The progress of the Non-CopyBack Write transaction, with DWT enabled by setting DoDWT to one in the WriteNoSnp from HN to SN is:

1. The Requester sends a WriteNoSnp or WriteUnique to Home.
2. The Home forwards the request to the Slave as WriteNoSnp with the DoDWT field set to one.
3. The Home also sends a Comp response to the Requester.
4. The Slave, detecting that the DoDWT value is one, sends a DBIDResp to the Requester using:
    a. ReturnNID field value in the request as TgtID.
    b. ReturnTxnID field value in the request as TxnID.

    If the DoDWT value was zero, the Slave would use SrcID and TxnID from the request as described in *DoDWT* on page 13-417.
5. The Requester sends the NCBWrData response to the Slave after receiving the DBIDResp response.

    Alternatively, if the write was canceled, then the Requester sends a WriteDataCancel instead of the NCBWrData message.
6. If the WriteData message indicates that a Tag Match is required, then the Slave sends a TagMatch response after completing the required Tag Match operation.

    As the dotted arrow in Figure 2-15 on page 2-71 shows, if the Slave does not support or does not perform the Tag Match operation then the Slave is permitted to send the TagMatch response after receiving the request without waiting for write data.

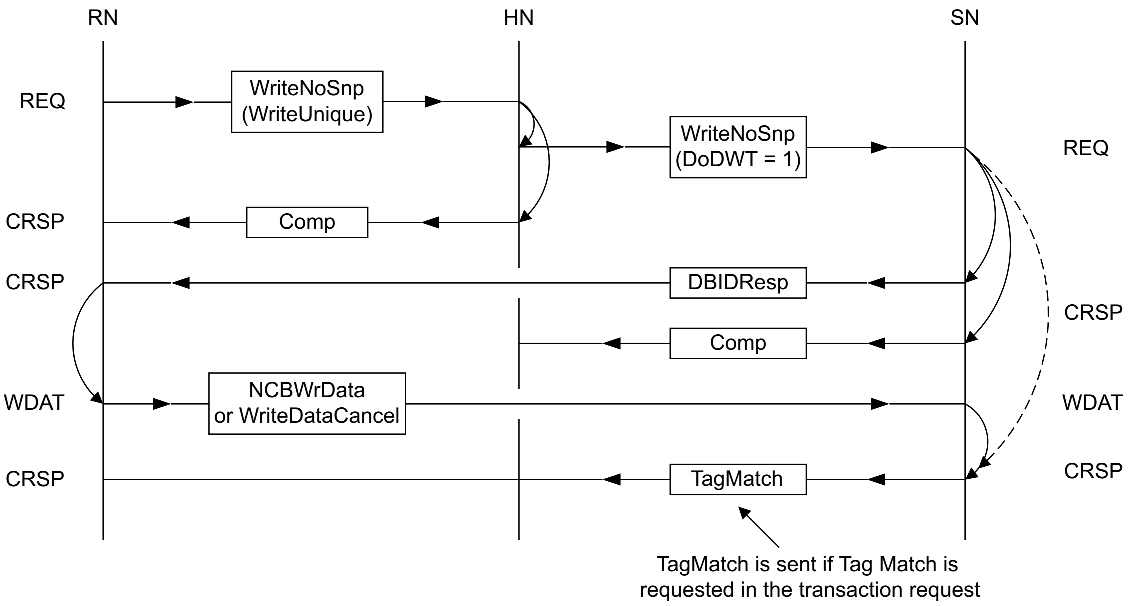


Figure 2-15 DWT transaction structure

The Request/Response rules are:

- Comp response from the Home Node to the Requester must only be sent by the Completer after the associated request is received.
- The request to the Slave Node must only be sent by the Home Node after the Home Node receives the associated request.
- If the request is WriteUniqueFull, and snooping is required, the Home does not need to wait for Snoop responses before forwarding the write to the Slave.
- If the request is WriteUniquePtl, and the Snoop responses include data, then the Home must take appropriate action to enable the Slave to merge this data with the data from the write.

Write and CMO combined request

This specification supports the combining of Write transactions with CMO type transactions, that is, Combined Write requests, when both are to the same address. The point where the two requests are combined can be an RN or an HN.

A Combined Write request is permitted for both Non-CopyBack and CopyBack writes.

Combined Write request transaction structure with DWT

Figure 2-16 on page 2-73 shows the Non-CopyBack Write transaction structure when DoDWT is set to 1. A Combined Write request, when DoDWT is set to 0, is also permitted. In this case, the transaction structure is similar to the Combined Write request structure shown in *Combined Write request transaction structure without DWT* on page 2-74.

The progress of the Non-CopyBack Combined Write transaction, with DWT enabled is:

1. The Requester sends a WriteNoSnp(P)CMO or WriteUnique(P)CMO to the Home Node on the REQ channel.
2. The Home forwards the request to the Slave Node as WriteNoSnp(P)CMO with the DoDWT bit set to one on the REQ channel.

   Alternatively, the Home is permitted to send:

   a. The Combined request with the DoDWT bit set to zero.
   b. A separate WriteNoSnp and CMO or PCMO request with the WriteNoSnp request DoDWT bit set to one.
   c. A separate WriteNoSnp and CMO or PCMO request with the WriteNoSnp request DoDWT bit set to zero.

      See *Combined Write request transaction structure without DWT* on page 2-74 for the transaction structure description when the DoDWT bit is set to zero.
3. The Home also sends a Comp response to the Requester, as a completion for the Write request and a separate CompCMO response for the CMO or PCMO request, both on the CRSP channel.
4. The Slave Node sends the Comp response for the Write to the Home and sends a DBIDResp response to the Requester, both on CRSP channel.

   See *DoDWT* on page 13-417 for how the TgtID and TxnID of the DBIDResp response are determined.

   The Slave Node is permitted to combine the DBIDResp response with the Comp and send a single CompDBIDResp response if the target of both the responses is the Home Node.
5. The Slave Node also sends the CompCMO response to the Home on the CRSP channel for the CMO or PCMO portion of the combined request.
6. The Requester sends the NCBWrData response to the Slave Node on the WDAT channel after receiving the DBIDResp response.

   Alternatively, if the write is canceled, then the Requester sends a WriteDataCancel instead of the NCBWrData message.
7. If the Request includes a PCMO then the Slave Node sends a Persist response to the Requester on CRSP channel.

   Alternatively, the Persist response can be sent to the Home Node, in which case the Home must forward the Persist response to the Requester.

   The Slave Node or the Home Node are permitted to combine the Persist response with CompCMO and send a single CompPersist response if the target of both responses is the same.

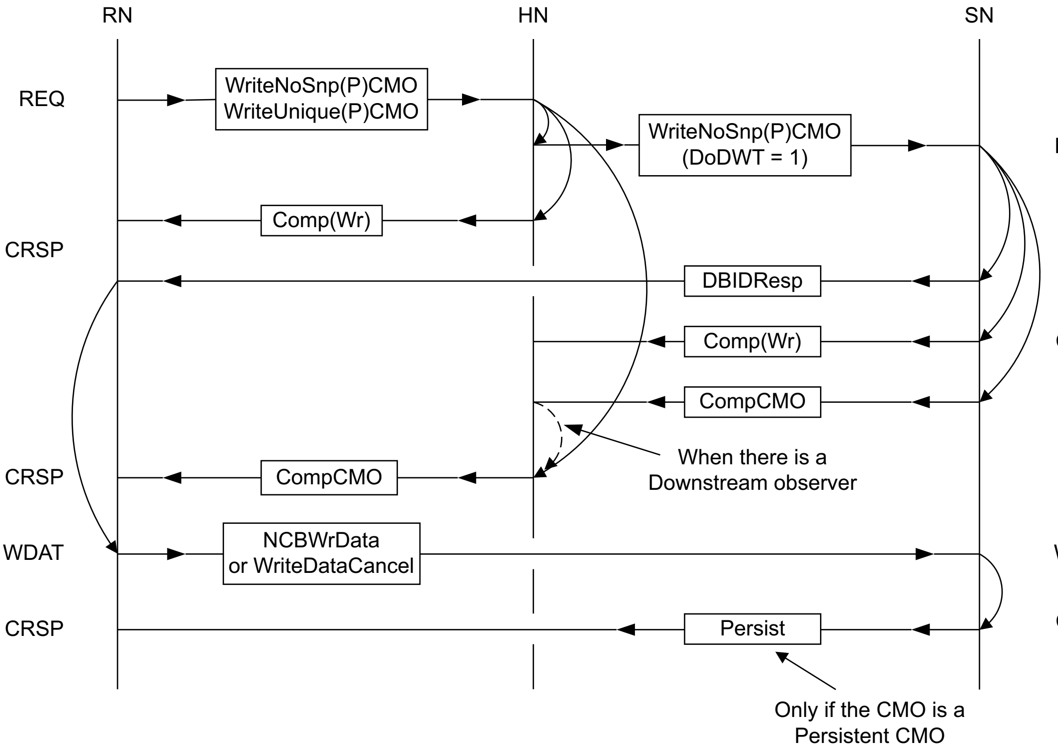


Figure 2-16 Non-CopyBack write combined with a PCMO or a CMO

The Request/Response rules are:

- The Comp and CompCMO response from the Home must only be sent after the associated request is received.
- The Home must send a Combined Write or a separate Write and CMO or PCMO request to the Slave Node only after the associated Combined Write request is received.
  - When there is an observer downstream of the Home Node then the Home must include the CMO or PCMO in the request sent to the Slave Node. Also, the Home must wait for the CompCMO or CompPersist response from the Slave Node before sending the CompCMO or CompPersist response to the Requester.
- The DBIDResp, Comp and CompCMO responses from the Slave must only be sent after the associated request is received.
- The Slave is permitted to send a combined CompDBIDResp response if both the Comp and DBIDResp are to the same target.
- WriteData must only be sent by the Requester after the DBIDResp or DBIDRespOrd or CompDBIDResp is received:
  - The Requester must not wait to receive the Comp response before the WriteData is sent.
  - The Completer is permitted to wait for the WriteData before sending the Comp response.
- If the request includes a PCMO then the Slave Node must send a Persist response after receiving the WriteData.
  - If the request is targeting volatile memory then the Slave Node is permitted, but not required, to send the Persist response after receiving the corresponding request without waiting for the WriteData.
  - The Slave Node is permitted to combine CompCMO and Persist as a CompPersist response if the target of the Persist response is the Home Node.

Combined Write request transaction structure without DWT

Figure 2-17 on page 2-75 shows the combined Write transaction structure without DWT.

This structure:

- Must be used for CopyBack Combined Write transactions.
- Must be used for Non-CopyBack Combined Write transactions with ExpCompAck set to one.
- Is permitted to be used for Non-CopyBack Combined Write transactions with ExpCompAck set to zero.

The progress of a Combined Write transaction, without DWT enabled is:

1. The Requester sends a Combined Write request to Home on the REQ Channel.
2. The Home forwards the request to the Slave Node as WriteNoSnp(P)CMO with the DoDWT bit set to zero on the REQ channel.

   Alternatively, the Home is permitted to send separate WriteNoSnp and CMO or PCMO requests with the WriteNoSnp request DoDWT bit set to zero.
3. The Home also sends a CompDBIDResp response to the Requester as a completion for the Write request, and a separate CompCMO response for the CMO or PCMO request, both on the CRSP channel.
   a. The Home is permitted to send separate DBIDResp and Comp responses to the Requester for a Non-CopyBack Combined Write request.
4. The Slave Node sends the Comp response and a DBIDResp to the Home Node, both on the CRSP channel.

   See *DoDWT* on page 13-417 for how the TgtID and TxnID of the DBIDResp response are determined.

   The Slave Node is permitted to combine the DBIDResp response with the Comp and send a single CompDBIDResp response if the target of both responses is the Home Node.
5. The Slave Node also sends the CompCMO response to the Home on the CRSP channel for the CMO or PCMO portion of the combined request.
6. The Requester sends a CBWrData response for a CopyBack request or a NCBWrData response for a Non-CopyBack request to Home on the WDAT channel.

   Alternatively, if the write is canceled, then the Requester sends a WriteDataCancel instead of the NCBWrData message.

   The Slave Node or the Home Node are permitted to combine the Persist response with the CompCMO into a single CompPersist response if the target of both responses is the same.
7. The Home Node after receiving the WriteData response from the Requester forwards it as either NCBWrData or WriteDataCancel to the Slave on the WDAT channel.
8. If the Request includes a PCMO then the Slave Node sends a Persist response to the Requester on the CRSP channel.

   Alternatively, the Persist response can be sent to the Home Node. In which case the Home must forward the Persist response to the Requester.

   The Slave Node or the Home Node are permitted to combine the Persist response with CompCMO into a single CompPersist response if the target of both responses is the same.

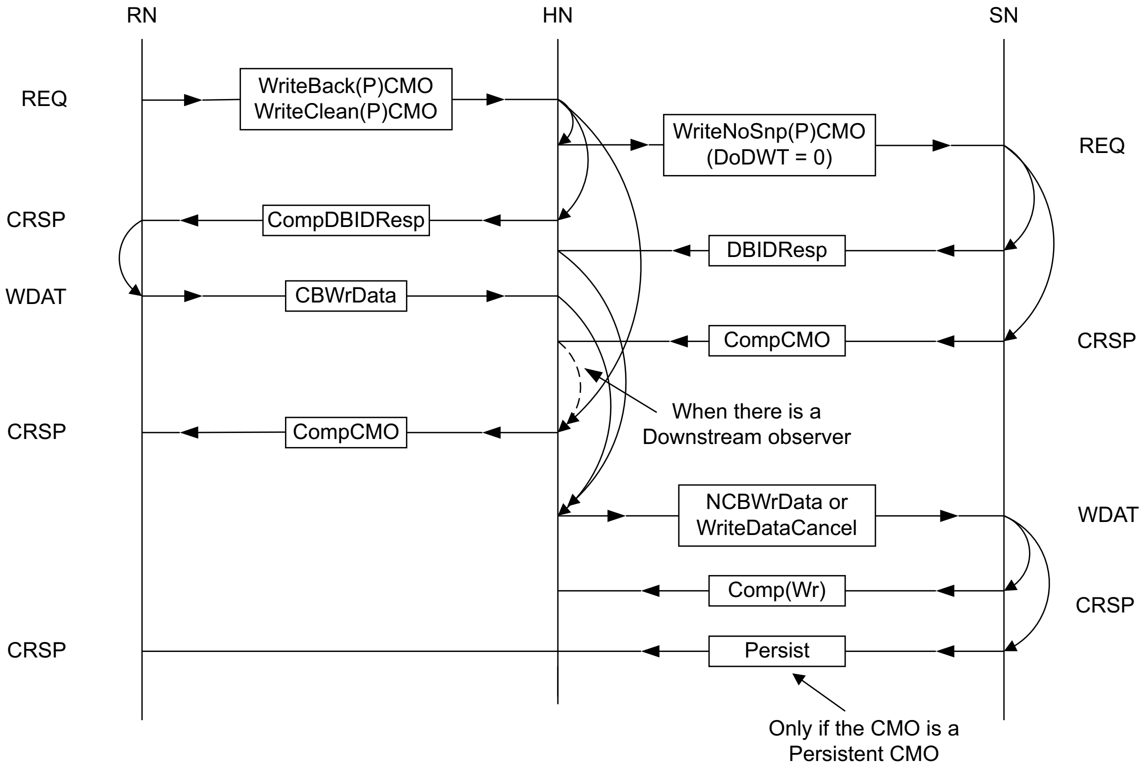


Figure 2-17 CopyBack write combined with a PCMO or a CMO

The Request/Response rules are:

- The CompDBIDResp and CompCMO response from the Home must only be sent after the associated request is received.
- The Home is permitted to send a separate Comp and DBIDResp response if the Combined Write request includes a Non-CopyBack Write request.
- The Home must send a Combined Write or separate Write and CMO or PCMO request to the Slave Node only after the associated Combined Write request is received.
  - When there is an observer downstream of the Home Node then the Home must include the CMO or PCMO request in the request sent to the Slave Node. Also, the Home must wait for the CompCMO or CompPersist response from the Slave Node before sending the CompCMO or CompPersist response to the Requester.
- The DBIDResp, Comp and CompCMO responses from the Slave must only be sent after the associated request is received.
- The Slave is permitted to send a combined CompDBIDResp response if both Comp and DBIDResp are to the same target.
- WriteData must only be sent by the Requester after the DBIDResp or DBIDRespOrd or CompDBIDResp is received.
  - The Requester must not wait to receive the Comp response before the WriteData is sent.
  - The Completer is permitted to wait for the WriteData before sending the Comp response.

- If the request includes a PCMO then the Slave must send a Persist response after receiving the WriteData.
  - If the request is targeting non-persistent memory then the Slave Node is permitted, but not required, to send the Persist response after receiving the corresponding request without waiting for the WriteData.
  - The Slave Node is permitted to combine CompCMO and Persist as a CompPersist response if the target of the Persist response is the Home Node.
- If the Combined Write request has ExpCompAck set to 1, which is only permitted in a Non-CopyBack Combined Write request, then the Requester must send a CompAck response after receiving Comp, DBIDResp, DBIDRespOrd or CompDBIDResp and the Home must set the DoDWT bit to zero in the request to the Slave Node.

WriteDataCancel in Write transactions

The Requester can send WriteDataCancel instead of NonCopyBackWrData on the WDAT channel in WriteNoSnpPtl, WriteUniquePtl, WriteUniquePtlStash and corresponding Combined Write transactions.

The Home Node can send WriteDataCancel instead of NonCopyBackWrData on the WDAT channel in WriteNoSnpFull, WriteNoSnpPtl and corresponding Combined Write transactions to the Slave Node.

The Completer of a Write transaction is permitted to send a Comp as soon as it receives a WriteDataCancel response without dependency on either the processing of the Write request or the completion of any snoops sent due to the write.

A Comp response in a write that is canceled only implies that the transaction loop is completed and makes no statement regarding the completion of coherency action initiated by the write.

CompCMO and Comp responses in Combined Write transactions

In a Combined Write transaction the completion message for the CMO and PCMO is CompCMO.

The CompCMO response is only permitted when the completion is for a CMO and PCMO that is combined with a write.

The CompCMO for the CMO and PCMO is permitted to be combined with Persist, as a CompPersist response.

A completion response for a standalone CMO and PCMO must use the Comp response.

The completion message for the write is Comp, similar to a standalone Write request.

The Comp for the write is permitted to be combined with DBIDResp.

Slave response to a Combined Write transaction

This section provides a summary of responses from an SN and how to determine the TgtID values in these responses.

As Figure 2-16 on page 2-73 and Figure 2-17 on page 2-75 show, an SN receiving a Combined Write request always responds in the following manner:

- Sends CompCMO for the CMO and PCMO.
- Sends Comp for the Write.
- The TgtID value in both the Comp and CompCMO is the value of the SrcID in the request. Both the responses are sent to the Home.
- Sends DBIDResp for the Write.
  - The Comp for the Write can be combined with the DBIDResp if both are targeting the Home. Table 2-8 on page 2-77 shows how the TgtID in the DBIDResp is determined.
- Sends a Persist response if the CMO is a PCMO.
  - Is permitted to combine CompCMO with Persist as a CompPersist response if the two are sent to Home.

The ReturnNID value in the request must be used as the target in the following responses by the Slave:

- In the DBIDResp, if the DoDWT bit in the request is set to one.
- In the Persist, if the CMO in the request is a PCMO.

The SrcID value in the request must be used as the target in the following responses by the Slave:

- In the DBIDResp, if the DoDWT bit in the request is set to zero.
- In Comp, for the write:
  - Comp for the write can be combined with DBIDResp if both are targeting the Home.
- In Comp, for the CMO and PCMO:
  - In a PCMO, the CompCMO can be combined with the Persist as a CompPersist if both are targeting the Home.

Table 2-8 Message field mapping in WriteNoSnpCMO from Home to Slave and its responses

DoDWT	HN to SN	TgtID in SN response	
	CMO type	DBIDResp	Persist
1	All	HN.Req.ReturnNID[a]	HN.Req.ReturnNID[a]
0	CleanShared CleanInvalid	HN.Req.SrcID[b]	-
	Persistent	HN.Req.SrcID[b]	HN.Req.ReturnNID[a]

[a] ReturnNID in the request from HN to SN.

[b] SrcID in the request from HN to SN.

Atomics

Atomic transactions can be classified in two categories based on their transaction structure:

- The following transaction returns only a completion response:
  - AtomicStore.
- The following transactions return Data with a completion response:
  - AtomicLoad.
  - AtomicSwap.
  - AtomicCompare.

Figure 2-18 on page 2-79 shows the structure of Atomic transactions at the Requester interface.

1. The Requester sends a request with the Atomic transaction opcode on the REQ channel.
2. Depending on the type of Atomic transaction the Completer has different options:
   a. If the request is an AtomicStore then the Completer has one of the following options:
      - Return separate responses:
        - Return a DBIDResp response that provides a data buffer identifier indicating that it can accept the write data for the transaction.
        - Return a Comp response to indicate that the transaction is observable by other Requesters.

        Both responses are returned on the CRSP channel.
      - Return a single combined CompDBIDResp response to indicate:
        - It can accept the write data for the transaction.
        - The transaction is observable by other Requesters.

        The combined response is returned on the CRSP channel.

   b. If the request is an Atomic Load, AtomicSwap, or AtomicCompare then the Completer does the following:
      - Returns a DBIDResp response on the CRSP channel. The DBIDResp response provides a data buffer identifier indicating that it can accept the write data for the transaction.
      - Returns the Read data and any associated transaction response with the CompData opcode on the RDAT channel.
3. The Requester sends the write data and any associated byte enables with the NonCopyBackWrData opcode on the WDAT channel. For AtomicCompare the write data can be sent using multiple transfers. See *Data transfer* on page 2-136.
4. If the WriteData message indicates that a Tag Match is required, then the Slave sends a TagMatch response after completing the required Tag Match operation.

   As the dotted line shows, if the Slave does not support or does not perform the Tag Match operation then the Slave is permitted to send the TagMatch response after receiving the request without waiting for write data.

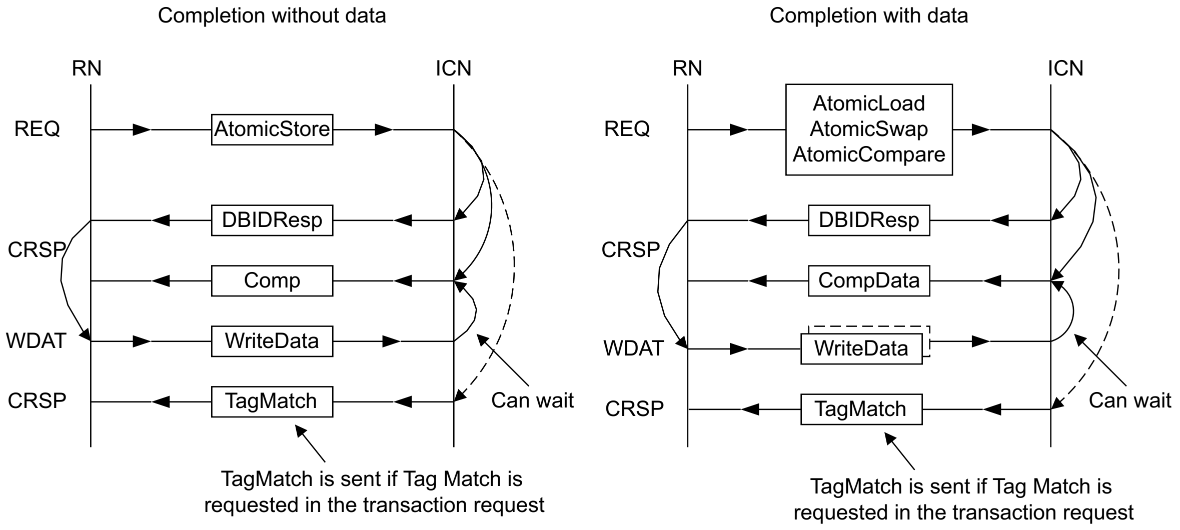


Figure 2-18 Atomic transaction structure

The Request/Response rules are:

The separate DBIDResp and Comp, or the combined CompDBIDResp must only be sent by the Completer after the associated request is received.

The CompData must only be sent by the Completer after the associated request is received.

WriteData must only be sent by the Requester after either DBIDResp or CompDBIDResp is received.

- If the request is an AtomicStore and the DBIDResp and Comp responses are sent separately:
  - Typically, the DBIDResp is sent by the Completer before Comp. However, it is permitted for DBIDResp and Comp to arrive in any order.
  - The Requester must send the write data after it has received the DBIDResp response.
  - The Requester must not wait to receive the Comp response before the write data is sent.
  - The Completer is permitted to wait for the WriteData before sending the Comp response.
- If the request is an AtomicLoad, AtomicSwap, or AtomicCompare transaction:
  - Typically, the DBIDResp is sent by the Completer before CompData. However, it is permitted for DBIDResp and CompData to be sent in any order.
  - Typically, the DBIDResp is received by the Requester before CompData. However, it is permitted for DBIDResp and CompData to arrive in any order.
  - The Requester must send the write data after it has received the DBIDResp response.
  - The Requester must not wait to receive the CompData response before the write data is sent.
  - The Completer is permitted to wait for the WriteData before sending the CompData response.

Self-snoop in Atomic transactions

This specification permits self-snooping of the Requester in Atomic transactions. The optional self-snoop is not shown in the figure. Self-snooping is controlled by the SnoopMe bit value in the Atomic request. See *SnoopMe* on page 13-419. The Request-Response rules for self-snooping in Atomic transactions are:

- An RN that does not invalidate its own cached copy of the cache line before sending an Atomic request must rely on self-snooping to:
  - Invalidate its own cached copy of the cache line.
  - Obtain a copy of the cache line if Dirty.
- The Home Node:
  - Must send a snoop to the Requester if the SnoopMe bit is set and Home determines that the cache line is present at the Requester.
  - Is permitted, but not required, to send a snoop to the Requester if the SnoopMe bit is set, and Home determines that the cache line is not present at the Requester.
  - Is permitted, but not required, to snoop the Requester when the SnoopMe bit is not set in an Atomic request.
  - Is expected to send a SnpUnique in response to an Atomic request, but is permitted to send a SnpCleanInvalid.

Note

An RN is permitted:

- To send a CopyBack request while the Atomic request to the same address with SnoopMe asserted is in progress.
- To issue an Atomic request with SnoopMe asserted while a CopyBack request to the same address is in progress.

Miscellaneous request types

Misc transactions include the following:

- DVM.
- PrefetchTgt.

DVM

The DVM transaction is DVMOp.

A DVM transaction is used to send a *Distributed Virtual Memory* (DVM) operation.

The progress of the DVM transaction is as follows:

1. The Requester sends a request with the DVMOp opcode on the REQ channel.
2. The Completer returns a DBIDResp response that provides a data buffer identifier indicating that it can accept the write data for the transaction.
3. The Requester sends the write data for the DVM transaction, with the NonCopyBackWrData opcode, on the WDAT channel. Only a single data transfer occurs for a DVM transaction.
4. The Completer returns a Comp response on the CRSP channel.

Figure 2-19 on page 2-81 shows the transaction structure.

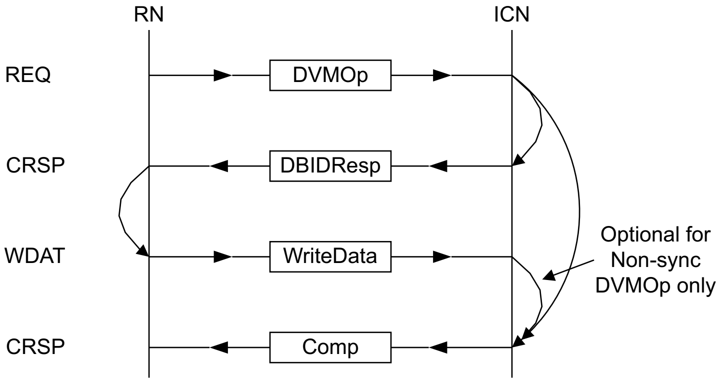


Figure 2-19 DVM transaction structure

The Request/Response rules are:

- DBIDResp must only be sent by the Completer after the associated request is received.
- Comp must only be sent by the Completer after the associated request is received.
- If the DVMOp is a Non-sync DVMOp, then the interconnect is permitted but not required to wait for WriteData before sending the Comp.
- The Completer is permitted to opportunistically combine Comp and DBIDResp into CompDBIDResp response.
- WriteData must only be sent by the Requester after the DBIDResp response is received.

PrefetchTgt

A Request to a shareable memory address sent from a Request Node directly to a Slave Node. The request can be used by the Slave Node to fetch and buffer data from main memory in anticipation of a subsequent Read request to the same location.

The progress of a PrefetchTgt transaction is as follows:

- The Requester sends a PrefetchTgt request on the REQ channel. The PrefetchTgt transaction does not include a response.

Figure 2-20 shows the transaction structure.

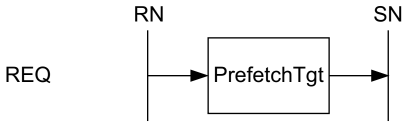


Figure 2-20 PrefetchTgt transaction structure

The Request/Response rules are:

- The Requester can deallocate the request as soon as the request is sent.
- The request must be accepted by the SN without any dependency and no Retry is permitted.
- The SN can drop the Request without taking any action.

2.3.2 Transaction Retry sequence

With the exception of PrefetchTgt, all Request transactions, as described in *Request transaction structure* on page 2-42, can begin with a Retry sequence.

Retry sequence

Request transactions are first sent without a *Protocol Credit* (P-Credit). If the transaction cannot be accepted at its Completer, then a RetryAck response must be given that indicates that the transaction has not been accepted and can be sent again when an appropriate credit is provided. When a transaction is sent a second time, with a credit, it is guaranteed to be accepted.

For further details on the Retry process and the use of credits see *Request Retry* on page 2-147.

The transaction Retry sequence is as follows:

1. The Requester sends a request, without a P-Credit, on the REQ channel.
2. The Completer provides a RetryAck response on the CRSP channel.
3. The Completer provides a PCrdGrant response on the CRSP channel, when appropriate, to indicate that a credit is available to re-send the transaction.
4. The Requester sends the transaction again, with a credit, on the REQ channel.

Figure 2-21 shows the RetryAck sequence.

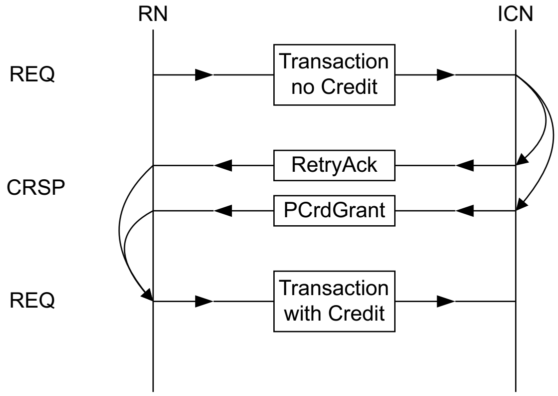


Figure 2-21 Transaction Retry sequence

The transaction Retry sequence rules are:

- RetryAck must only be sent by the Completer after the associated request is received.
- PCrdGrant must only be sent by the Completer after the associated request is received.
- RetryAck is typically sent by the Completer before PCrdGrant. However, it is permitted to send RetryAck after PCrdGrant.
- RetryAck is typically received by the Requester before PCrdGrant. However, it is permitted to receive RetryAck after PCrdGrant.
- The transaction with credit must only be sent by the Requester after both the RetryAck response and an appropriate PCrdGrant response are received.

Not retrying a transaction

The protocol supports deciding not to resend the transaction between the point that it receives a RetryAck and the point that it is resent using a credit. The sequence is identical to the transaction Retry sequence that Figure 2-21 on page 2-82 shows, except that the final transaction with credit is sent as a PCrdReturn transaction. This acts as a null transaction and returns the credit to the Completer.

The sequence and rules are identical to those for a Retry sequence.

Figure 2-22 shows the cancelled transaction sequence.

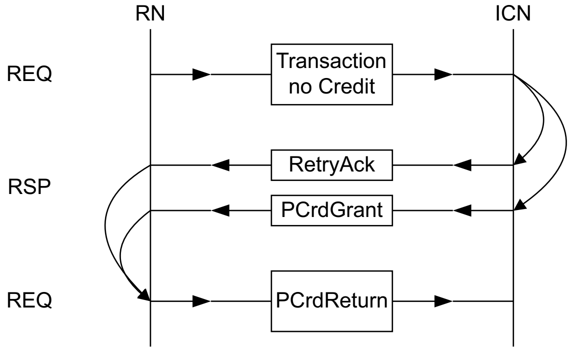


Figure 2-22 Cancelled transaction sequence

2.3.3 Snoop transactions

Snoop transactions are sent from the interconnect to a Request Node:

- An RN-F is fully coherent and is required to accept all Snoop transactions.
- An RN-D supports receiving only SnpDVMOp transactions.
- An RN-F and RN-D must respond to received Snoop requests, except for DVMOp(Sync), in a timely manner, without creating any dependency on completion of outstanding requests.

There are several options for the transaction structure of a snoop:

- Snoop with response to Home.
- Snoop with Data to Home.
- Snoop with Data return to Requester and response to Home.
- Snoop with Data return to Requester and Data to Home.
- Snoop DVM operation and response to Misc Node.

A snoop transaction can also be used to stash data at the Snoopee. The options for the transaction structure of a Stash type snoop are:

- Stashing snoop with Data from Home.
- Stashing snoop with Data using DMT.

Note

Figures relating to Snoop transactions show the snooped Request Node (RN) on the right, and the interconnect (ICN) on the left. This is consistent with the ordering of the Request/Snoop process.

Snoop with response without Data to Home

The progress of a Snoop transaction with response to Home is as follows:

1. The interconnect provides a Snoop request on the SNP channel that can be any Snoop transaction supported by the RN.
2. The RN returns the SnpResp snoop response on the SRSP channel.

Figure 2-23 shows the transaction structure.

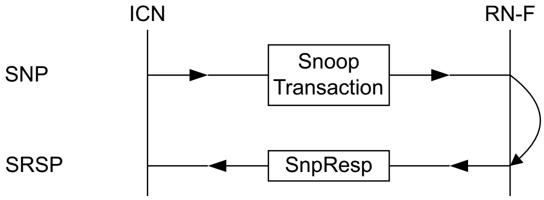


Figure 2-23 Snoop transaction structure with response to Home

The snoop with response to Home rules are:

- SnpResp must only be sent by the RN after the associated Snoop request is received.

Snoop with response with Data to Home

The progress of a Snoop transaction with Data to Home is as follows:

1. The interconnect provides a Snoop request on the SNP channel. This can be one of the following Snoop transactions:
    - SnpOnceFwd, SnpOnce.
    - SnpCleanFwd, SnpClean.
    - SnpNotSharedDirtyFwd, SnpNotSharedDirty.
    - SnpSharedFwd, SnpShared.
    - SnpUniqueFwd, SnpUnique, SnpUniqueStash.
    - SnpPreferUniqueFwd, SnpPreferUnique.
    - SnpCleanShared.
    - SnpCleanInvalid.
2. The RN returns the data and associated response using the SnpRespData or SnpRespDataPtl opcode on the DAT channel.

Figure 2-24 shows the transaction structure.

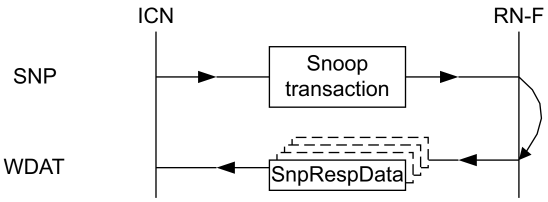


Figure 2-24 Snoop transaction structure with data to Home

The snoop with data to Home rules are:

- SnpRespData or SnpRespDataPtl, as required, must only be sent by the RN-F after the associated Snoop request is received.

Snoop with Data forwarded to Requester without or with Data to Home

The progress of a Snoop transaction with Data forwarded to the Requester is as follows:

1. The interconnect provides a Snoop request on the SNP channel. This can be one of the following Snoop transactions:
    - SnpOnceFwd.
    - SnpCleanFwd.
    - SnpNotSharedDirtyFwd.
    - SnpSharedFwd.
    - SnpUniqueFwd.
    - SnpPreferUniqueFwd.
2. The snooped RN forwards the Data to the Requester using the CompData opcode on the WDAT channel and either:
    - Sends a response to Home using the SnpRespFwded opcode on the SRSP channel.
    - Sends Data to Home using the SnpRespDataFwded opcode on the WDAT channel.

Figure 2-25 shows the transaction structure with response without Data to Home.

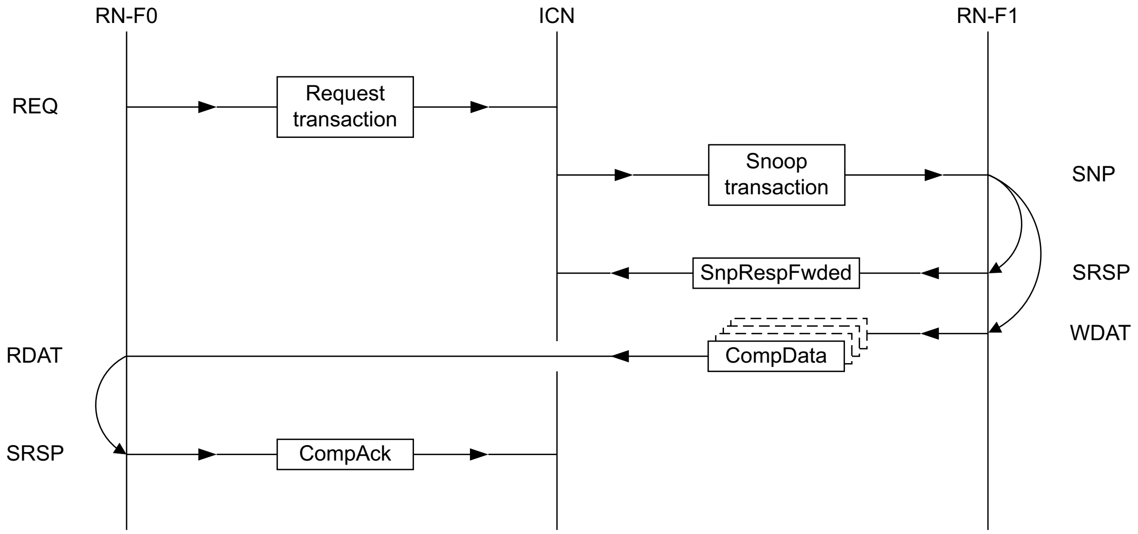


Figure 2-25 Snoop with Data forwarded to Requester with response to Home

Figure 2-26 shows the transaction structure with response with Data to Home.

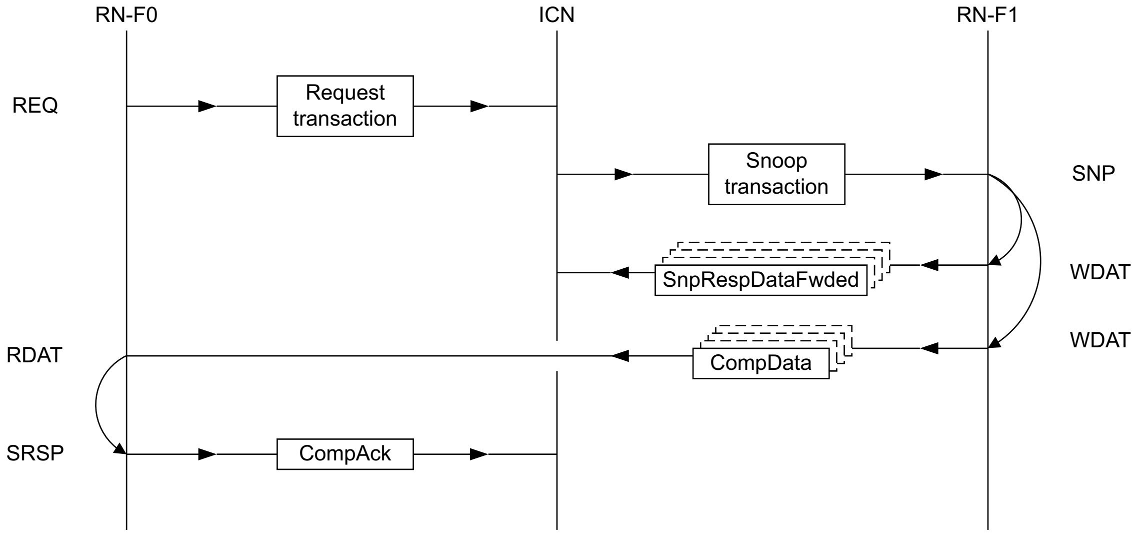


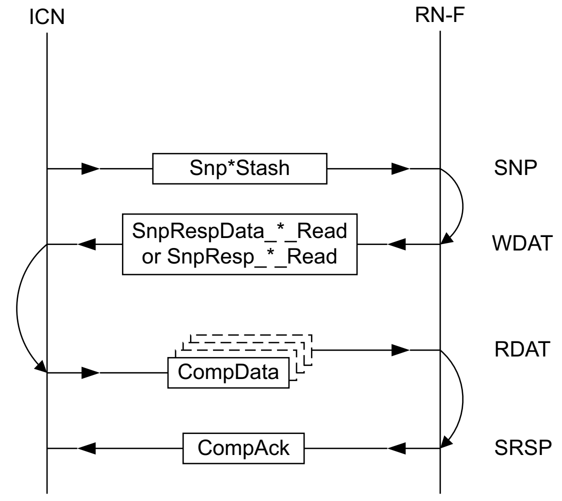
Figure 2-26 Snoop with Data forwarded to Requester with Data to Home

The Snoop request/response rules for Forwarding snoops are:

- SnpRespFwded or SnpRespDataFwded must only be sent by the Snoopee after the associated Snoop request is received.
- CompData must only be sent by the Snoopee after the associated Snoop request is received.
- SnpRespFwded or SnpRespDataFwded and CompData responses by the Snoopee can be sent in any order.
- CompAck must be sent only after receiving the first Data response packet.

Stash snoops

Figure 2-27 shows an example of a stash type snoop with Data Pull. The RN-F is then returned CompData in response to the Read transaction initiated by the SnpResp_*_Read response. Figure 2-27 shows data sent by the ICN to the RN-F. The data sent to the Requester can be a separate data-only response or can be sent using DMT or DCT.



SnpResp with Data is only permitted for SnpUniqueStash

Figure 2-27 Stash type snoop with Data Pull, Data response from Snoopee, and Data from Home

Snoop DVMOp

The progress of a SnpDVMOp transaction is as follows:

1. The interconnect provides two Snoop requests with the SnpDVMOp opcode on the SNP channel.
2. The RN returns a single SnpResp snoop response on the SRSP channel.

Figure 2-28 shows the transaction structure.

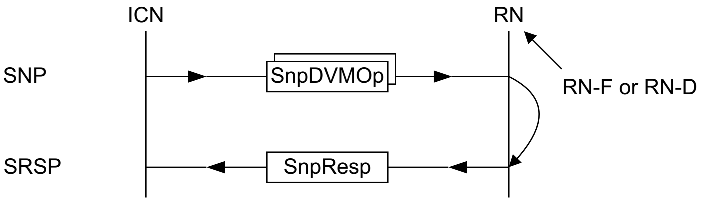


Figure 2-28 SnpDVMOp transaction structure

The SnpDVMOp rules are:

- The SnpResp response must only be sent by the RN after both Snoop requests are received.

2.4 Transaction identifier fields

Each transaction consists of a number of different packets that are transferred across the interconnect. A set of identifier fields, within a packet, are used to provide additional information about a packet. The different identifier fields are:

***Target Identifier* (TgtID), *Source Identifier* (SrcID)**

These identifiers route packets across the interconnect. See *Details of transaction identifier fields* on page 2-89 and Chapter 3 *Network Layer*.

***Transaction Identifier* (TxnID), *Data Buffer Identifier* (DBID), *Return Transaction Identifier* (ReturnTxnID), *Forward Transaction Identifier* (FwdTxnID)**

These fields relate all the packets associated with a single transaction. See *Details of transaction identifier fields* on page 2-89.

***Data Identifier* (DataID), *Critical Chunk Identifier* (CCID)**

These fields identify the individual data packets within a transaction. See *Data packetization* on page 2-138.

***Logical Processor Identifier* (LPID), *Stash Logical Processor Identifier* (StashLPID)**

These fields identify individual processing agents within a single Requester. See *Logical Processor Identifier* on page 2-115.

***Stash Node Identifier* (StashNID)**

This field identifies the node that is the Stash target. See *Supporting REQ packet fields* on page 7-292.

***Return Node Identifier* (ReturnNID), *Forward Node Identifier* (FwdNID)**

These fields identify the node that the response with Data is to be sent to. See *Details of transaction identifier fields* on page 2-89.

***Home Node Identifier* (HomeNID)**

This field is used to identify the node that the CompAck response is to be sent to. See *Details of transaction identifier fields* on page 2-89.

***Persistent Group Identifier* (PGroupID)**

This field is used to identify different sets of CleanSharedPersistSep transactions. See *Details of transaction identifier fields* on page 2-89.

***Stash Group Identifier* (StashGroupID)**

This field is used to identify different sets of StashOnceSep transactions. See *StashGroupID* on page 13-408.

***Tag Group Identifier* (TagGroupID)**

This field is used to identify different sets of Write transactions that require Tag Match operations to be performed. See *TagGroupID* on page 13-421.

2.5 Details of transaction identifier fields

A transaction request includes a TgtID that identifies the target node, and a SrcID that identifies the source node. These IDs are used to route packets across the interconnect.

A transaction request includes a TxnID that is used to identify the transaction from a given Requester. It is required that the TxnID, except for PrefetchTgt, must be unique for a given Requester. The Requester is identified by the SrcID. This ensures that any returning read data or response information can be associated with the correct transaction.

A 12-bit field is defined for the TxnID with the number of outstanding transactions being limited to 1024. A Requester is permitted to reuse a TxnID value after it has received either:

- All responses associated with a previous transaction that have used the same value.
- A RetryAck response for a previous transaction that used the same value.

Transaction identifier field flows on page 2-94 gives more detailed rules for the different transaction types. The TxnID field is not applicable in a PrefetchTgt request and can take any value.

A value used in the TxnID field of a Request from Home to Slave can be reused by Home once all responses that are required to deallocate the request are received or a RetryAck response is received.

A transaction that is retried is not required to use the same TxnID. See *Request Retry* on page 2-147.

A transaction request from Home to Slave includes a ReturnNID that is used to determine the TgtID for the following responses from the Slave Node:

- Data response.
- DBIDResp response.
- Persist response.
- TagMatch response.

Its value must be either the Node ID of Home or the Node ID of the original Requester.

ReturnNID is only applicable in a ReadNoSnp, ReadNoSnpSep, CleanSharedPersistSep, WriteNoSnp, Combined Write, and Atomic requests from Home to Slave. The field is inapplicable and must be set to zero in all other requests from Home to Slave.

ReturnNID is inapplicable and must be set to zero in all requests from Requester to Home and Requester to Slave.

The following are the expected and permitted values for the ReturnNID in requests from the Home Node to the Slave Node.

In ReadNoSnp, ReadNoSnpSep and CleanSharedPersistSep:

- Expected value is the original Requester Node ID but is permitted to be the Home Node ID.
- Used as the TgtID in CompData, DataSepResp and Persist responses.

In Atomic with TagOp *Invalid*:

- For AtomicStore and ReturnNID, can take any value and the value is not used in any responses.
- For non-Store Atomics, the ReturnNID must be the Home Node ID. The value is used as the TgtID in CompData.

In Atomic with TagOp *Match*:

- ReturnNID must be the Home Node ID.
- The value is used as the TgtID in CompData and TagMatch responses.

In WriteNoSnp with TagOp not *Match*:

- When DoDWT = 0, the ReturnNID can take any value, and the value is not used in any responses.
- When DoDWT = 1, the ReturnNID value is expected to be the original Requester Node ID but is permitted to be the Home Node ID. Used as the TgtID in the DBIDResp response.

In WriteNoSnp with TagOp *Match*:

- Irrespective of the value of DoDWT, the ReturnNID value is expected to be the original Requester Node ID but is permitted to be the Home Node ID.
- When DoDWT = 0, the value is used as the TgtID in the TagMatch response only.
- When DoDWT = 1, the value is used as the TgtID in the DBIDResp and TagMatch responses.

In non-PCMO Combined Write:

- When DoDWT = 0, the ReturnNID can take any value, and the value is not used in any responses.
- When DoDWT = 1, the ReturnNID value is expected to be the original Requester Node ID but is permitted to be the Home Node ID. Used as the TgtID in the DBIDResp response.

In WriteNoSnpFullClnShPer and WriteNoSnpPtlClnShPer:

- Irrespective of the value of DoDWT, the ReturnNID value is expected to be the original Requester Node ID but is permitted to be the Home Node ID.
- When DoDWT = 0, the value is used as the TgtID in the Persist response only.
- When DoDWT = 1, the value is used as the TgtID in the DBIDResp and Persist responses.

A transaction request from Home to Slave also includes a ReturnTxnID field to convey the value of TxnID in the data response and the DBIDResp response from the Slave. Its value, when applicable, must be either:

- The TxnID generated by Home, when the ReturnNID is the Node ID of the Home.
- The TxnID of the original Requester, when the ReturnNID is the Node ID of the original Requester.

ReturnTxnID is only applicable in a ReadNoSnp, ReadNoSnpSep, WriteNoSnp, Combined Write, and Atomic requests from Home to Slave. The field is inapplicable and must be set to zero in all other requests from Home to Slave.

ReturnTxnID is inapplicable and must be set to zero in all requests from Requester to Home and Requester to Slave.

The following are the expected and permitted values for ReturnTxnID in requests from the Home Node to the Slave Node.

In ReadNoSnp and ReadNoSnpSep:

- Expected value is the original Requester TxnID but permitted to be the Home TxnID.
- Used as the TxnID in the CompData and DataSepResp responses.

In Atomic with TagOp *Invalid* or *Match*:

- For AtomicStore, the ReturnTxnID can take any value, and the value is not used in any response.
- For non-Store Atomics, the ReturnTxnID must be the Home TxnID. The value is used as the TxnID in the CompData response.

In WriteNoSnp with any TagOp value:

- When DoDWT = 0, ReturnTxnID can take any value, and the value is not used in any response.
- When DoDWT = 1, ReturnTxnID value is expected to be the original Requester TxnID but is permitted to be the Home TxnID. Used as the TxnID in the DBIDResp. response.

In Combined Write:

- When DoDWT = 0, ReturnTxnID can take any value, and the value is not used in any response.
- When DoDWT = 1, ReturnTxnID value is expected to be the original Requester TxnID but is permitted to be the Home TxnID. Used as the TxnID in the DBIDResp response.

CompData from Home, and from the Slave Node, includes the HomeNID field that is used by the Requester to identify the target of the CompAck that it might need to send in response to CompData. HomeNID is applicable in CompData and DataSepResp and is inapplicable and must be set to zero for all other Data messages.

Note

There is no functional requirement for the HomeNID and DBID fields in the DataSepResp response because the values that are provided in the RespSepData response are identical and can always be used. However, this specification requires that these values are included to assist in debugging and protocol checking.

A CleanSharedPersistSep and Combined Write with PCMO request includes a PGroupID to identify the Persistence Group that the request belongs to. If a Requester has persistent CMO requests from different functional agents that it would like to identify for performant persistent CMO handling, it can assign a different PGroupID value to each group of Persist requests. Use of this 8-bit field is applicable in CleanSharedPersistSep and Combined Write with PCMO transactions. It is also applicable in Persist and CompPersist responses. It is inapplicable and must be set to zero in all other requests and responses. See *PGroupID* on page 13-406:

- PGroupID must be sent in the CleanSharedPersistSep request and a Combined Write request that includes a PCMO.
- The PGroupID value returned in the Persist response can be used by a Requester to separately track completions of Persist responses from each group.
- It is expected that a Requester that does not support multiple persistence groups sets the PGroupID value to zero.
- A Requester that is making use of PGroupID for passing a barrier typically will not reuse a PGroupID value until all the earlier sent CleanSharedPersistSep requests from that group have received Persist responses.
- The Completer is required to reflect back PGroupID in the Persist and CompPersist responses, and the responses of Combined Write requests that include a PCMO.
- The PGroupID field in the Comp and CompCMO response from both the Home and Slave is inapplicable and must be set to zero.

A Snoop request from Home to RN-F includes a FwdNID that is used to determine the TgtID for the Data response from the RN-F. Its value must be the Node ID of the original Requester.

The FwdNID field is only applicable in:
- SnpSharedFwd.
- SnpCleanFwd.
- SnpOnceFwd.
- SnpNotSharedDirtyFwd.
- SnpUniqueFwd.
- SnpPreferUniqueFwd.

It is inapplicable and must be set to zero in all other snoops.

A Snoop request from Home to RN-F also includes a FwdTxnID field to convey the value of TxnID in the Data response from the RN-F. Its value must be the TxnID of the original Request.

The FwdTxnID field is only applicable in:

- SnpSharedFwd.
- SnpCleanFwd.
- SnpOnceFwd.
- SnpNotSharedDirtyFwd.
- SnpUniqueFwd.
- SnpPreferUniqueFwd.

It is inapplicable and must be set to zero in all other snoops.

The DBID field permits the Completer of a transaction to provide its own identifier for a transaction. The Completer sends a response that includes a DBID. The DBID value is used as the TxnID field value in the:

- WriteData response of Write, Combined Write, Atomic, and DVMOp transactions.
- CompData response of Stash transactions for Data Pull purposes.
- CompAck response of Read, Dataless, WriteNoSnp, WriteUnique and Non-CopyBack Combined Write transactions that include a CompAck response.

The DBID value used by a Completer in responses of a given transaction must be unique for a given Requester in the following cases:

- DBIDResp or DBIDRespOrd or CompDBIDResp for all Write transactions, except in WriteNoSnpZero and WriteUniqueZero.
- DBIDResp or DBIDRespOrd or CompDBIDResp for all Combined Write transactions.
- DBIDResp or DBIDRespOrd or CompDBIDResp for Atomic transactions.
- DBIDResp or DBIDRespOrd or CompDBIDResp for DVMOp transactions.
- CompData or RespSepData for Read transactions that include CompAck, except in the case when ReadOnce* and ReadNoSnp do not use the resultant CompAck for deallocation of the request at Home.
- Comp for Dataless transactions that include CompAck.

The DBID value is applicable in the DataSepResp response to Read requests that include CompAck and it must be the same as the DBID value in the associated RespSepData response.

A Comp response message sent separate from a DBIDResp or DBIDRespOrd message for a Write transaction must include the same DBID field value in the Comp and DBIDResp or DBIDRespOrd message.

A Comp response message sent separate from a DBIDResp or DBIDRespOrd message for a Combined Write transaction must include the same DBID field value in the Comp and DBIDResp or DBIDRespOrd message.

A Comp response message sent separate from a DBIDResp or DBIDRespOrd message for a Atomic transaction is permitted, but is not required, to include the same DBID field value in the Comp and DBIDResp or DBIDRespOrd message.

A Completer is permitted, but not required, to use the same DBID value for two transactions with different Requesters. A Completer is permitted to reuse a DBID value after it has received all packets required to deallocate a previous transaction that has used the same value. *Transaction identifier field flows* on page 2-94 gives more detailed rules for the different transaction types.

The DBID value used by a Snoop Completer in response to a Stash type snoop that includes a Data Pull must be unique with respect to:

- The DBID values in other Snoop responses to Stash type snoops that use Data Pull.
- The TxnID of any outstanding Request from that Snoop Completer.

The Completer is not required to utilize the DBID field, and is permitted to set it to any value in:

- WriteNoSnpZero and WriteUniqueZero transactions.
- Read transactions without CompAck.
- Dataless transactions without CompAck.
- Snoop response to a Stash type snoop that does not include a Data Pull.
- Snoop response to a Non-stash type snoop.

Note

The advantage of using the DBID assigned by the Completer, instead of the TxnID assigned by the Requester, is that the Completer can use the DBID to index into its request structure instead of performing a lookup using TxnID and SrcID to determine which transaction write data or completion acknowledge is associated with which request.

If a Completer is using the same DBID value for different Requesters, which it must do if its operation requires more than 1024 DBID responses to be active at the same time, then it must use SrcID in combination with DBID to determine which request should be associated with a write data or response message.

The DBIDResp response is also used to provide certain ordering guarantees relating to the transaction. See *Transaction ordering* on page 2-120.

2.6 Transaction identifier field flows

This section shows the transaction identifier field flows for the different transaction types:

- *Read transactions* on page 2-95.
- *Dataless transactions* on page 2-103.
- *Write transactions* on page 2-105.
- *DVMOp transaction* on page 2-113.
- *Transaction requests with Retry* on page 2-113.
- *Protocol Credit Return transaction* on page 2-114.

In the associated figures:

- The fields included in each packet are:
  - For a Request packet: TgtID, SrcID, TxnID, StashNID, StashLPID, ReturnNID, ReturnTxnID, PGroupID, StashGroupID, and TagGroupID.
  - For a Response packet: TgtID, SrcID, TxnID, DBID, PGroupID, StashGroupID, and TagGroupID.
  - For a Data packet: TgtID, SrcID, TxnID, HomeNID and DBID.
  - For a Snoop packet: SrcNID, TxnID, FwdNID, FwdTxnID and StashLPID.
- All fields with the same color are the same value.
- The curved loop-back arrows show how the Requester and Completer use fields from earlier packets to generate fields for subsequent packets.
- A box containing an asterix [*] indicates when a field is first generated, that is, it indicates the agent that determines the original value of the field.
- A field enclosed in parentheses indicates that the value is effectively a fixed value. Typically this is the case for the SrcID field when a packet is sent, and the TgtID field when a packet arrives at its destination.
- A field that is crossed-out indicates that the field is not valid.
- It is permitted for the TgtID of the original transaction to be re-mapped by the interconnect to a new value. This is shown by a box containing the letter R. This is explained in more detail in Chapter 3 *Network Layer*.

Note

An identifier field, in every packet sent, belongs to one of the following categories:

- New value. An asterisk indicates that a new value is generated.
- Generated from an earlier packet. A loop back arrow indicates the source.
- Fixed value. The value is enclosed in brackets.
- Not valid. The field is crossed-out.

In the following examples, any transaction IDs that are not relevant for the example are sometimes omitted for clarity.

2.6.1 Read transactions

This section shows identifier field flows in Read transactions with and without Direct Data Transfer:

- *ID value transfer with DMT*.
- *ID value transfer with DMT and separate Comp and Data* on page 2-97.
- *ID value transfer with DCT* on page 2-99.
- *ID value transfer without Direct Data Transfer* on page 2-101.

ID value transfer with DMT

Figure 2-29 shows how the Target and Transaction ID values in the DMT transaction messages are derived. For example, the value of SrcID in the ReadNoSnp request from ICN is assigned by ICN, whereas the ReturnNID, which is used as TgtID in the Data response, is set to the value of SrcID of the received Read request.

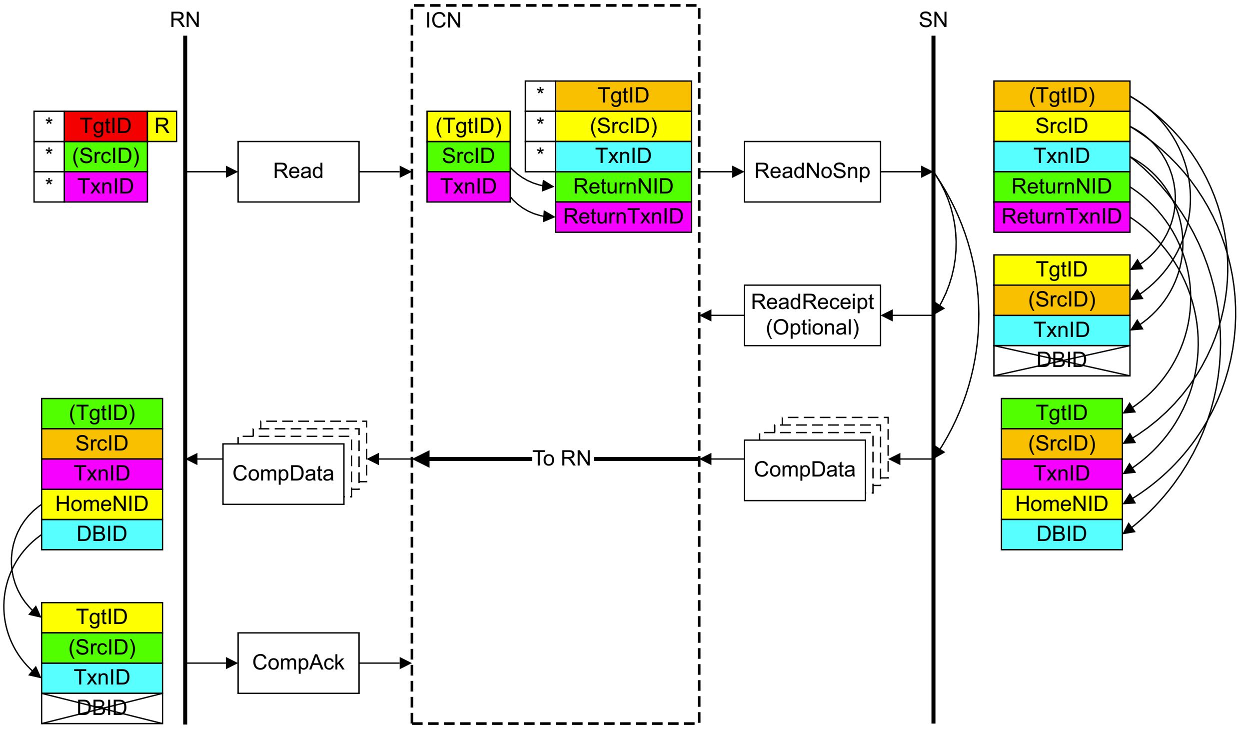


Figure 2-29 ID value transfer in a DMT transaction

The required steps in the flow that Figure 2-29 shows are:

1. The Requester starts the transaction by sending a Request packet.

   The identifier fields of the request are generated as follows:

   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be remapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a TxnID field that is unique for that Requester.

2. The recipient Home Node in the ICN generates a Request to the Slave Node.

   The identifier fields of the request are generated as follows:
   - The TgtID is set to the value required for the Slave.
   - The SrcID is a fixed value for the Home.
   - The TxnID is a unique value generated by the Home.
   - The ReturnNID is set to the same value as the SrcID of the original request.
   - The ReturnTxnID is set to the same value as the TxnID of the original request.

3. If the request to the Slave requires a ReadReceipt, the Slave provides the read receipt.

   The identifier fields of the ReadReceipt response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Slave. This also matches the TgtID received.
   - The TxnID is set to the same value as the TxnID of the request.
   - The DBID field is not valid.

4. The Slave provides the read data.

   The identifier fields of the read data response are generated as follows:
   - The TgtID is set to the same value as the ReturnNID of the request.
   - The SrcID is a fixed value for the Slave. This also matches the TgtID received.
   - The TxnID is set to the same value as the ReturnTxnID of the request.
   - The HomeNID is set to the same value as the SrcID of the request.
   - The DBID is set to the same value as the TxnID of the request.

5. The Requester receives the read data and sends a CompAck acknowledgment.

   The identifier fields of the CompAck are generated as follows:
   - The TgtID is set to the same value as the HomeNID of the read data.
   - The SrcID is a fixed value for the Requester. This also matches the TgtID that was received.
   - The TxnID is set to the same value as the DBID of the read data.
   - The DBID field is not valid.

The CompAck response from Requester to Home is not required for all Requests.

If the original request requires a ReadReceipt, the following additional step is included:

- The Home receives the Request packet and provides the read receipt.

  The identifier fields of the ReadReceipt response are generated as follows:
  - — The TgtID is set to the same value as the SrcID of the request.
  - — The SrcID is a fixed value for the Completer. This also matches the TgtID received.
  - — The TxnID is set to the same value as the TxnID of the request.
  - — The DBID field is not valid.

Details of transaction identifier fields on page 2-89 details when the TxnID value and DBID value can be reused.

ID value transfer with DMT and separate Comp and Data

Figure 2-30 shows how the identifier field values are derived in DMT transaction messages that use separate Comp and Data.

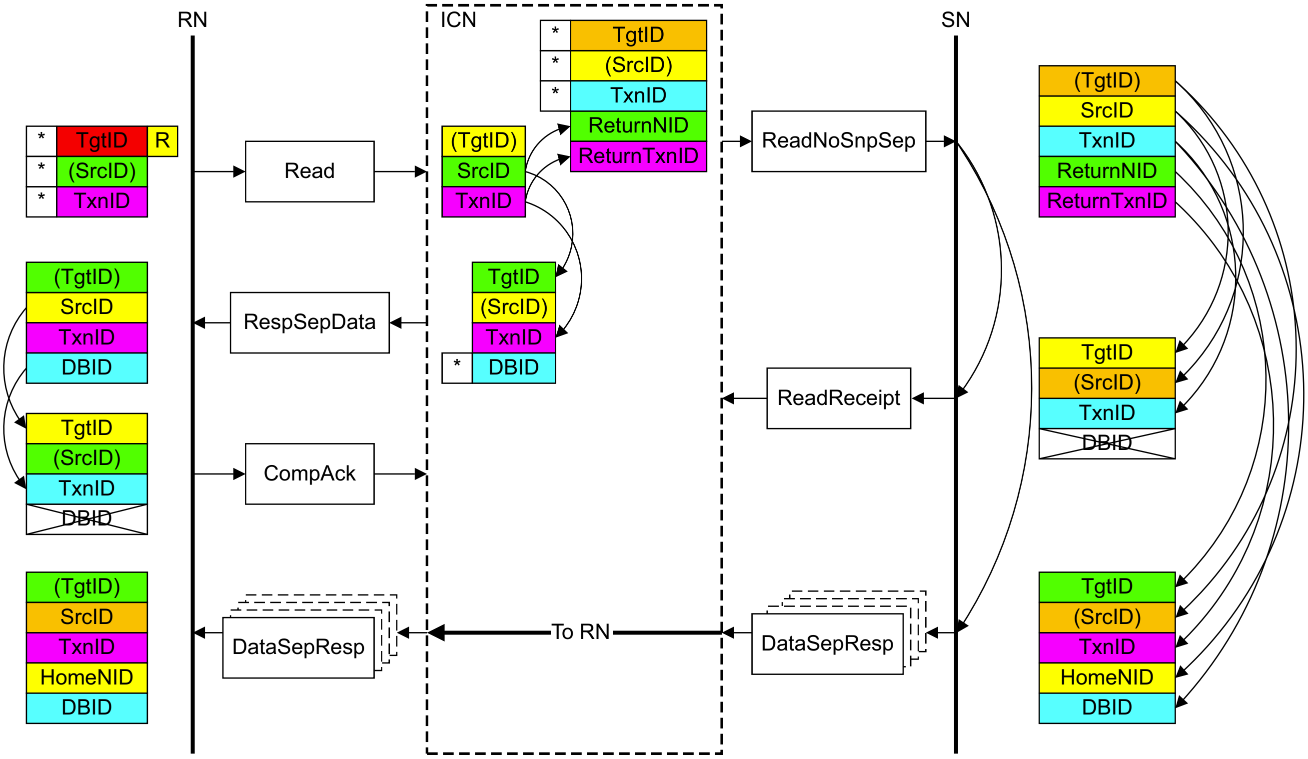


Figure 2-30 ID value transfer in a DMT transaction with separate Comp and Data

The required steps in the flow that Figure 2-30 shows are:

1. The requester starts the transaction by sending a request packet.

   The identifier fields of the request are generated as follows:
   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be remapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a TxnID field that is unique for that Requester.

2. The recipient Home Node in the ICN generates a request to the Slave Node.

   The identifier fields of the request are generated as follows:
   - The TgtID is set to the value required for the Slave.
   - The SrcID is a fixed value for the Home.
   - The TxnID is a unique value generated by the Home.
   - The ReturnNID is set to the same value as the SrcID of the original request.
   - The ReturnTxnID is set to the same value as the TxnID of the original request.

3. The recipient Home Node in the ICN provides the separate read response.

   The identifier fields of the read response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Home.
   - The TxnID is set to the same value as the TxnID of the original request.
   - The DBID value is a unique value generated by the Home and is the same value as the TxnID in the request to the Slave.

4. The Requester receives the read response and sends a CompAck acknowledgment.

   The identifier fields of the CompAck are generated as follows:
   - The TgtID is set to the same value as the SrcID of the read response.
   - The SrcID is a fixed value for the Requester.
   - The TxID is set to the unique DBID value generated by the Home.
   - The DBID value is not valid.

5. The request to the Slave requires a ReadReceipt, the Slave provides the read

   receipt.

   The identifier fields of the ReadReceipt response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Slave. This also matches the TgtID received.
   - The TxnID is set to the same value as the TxnID of the request.

6. The Slave provides the separate read data.

   The identifier fields of the read data are generated as follows:
   - The TgtID is set to the same value as the ReturnNID of the request.
   - The SrcID is a fixed value for the Slave. This also matches the TgtID received.
   - The TxnID is set to the same value as the ReturnTxnID of the request.
   - The HomeNID is set to the same value as the SrcID of the request.
   - The DBID is set to the same value as the TxnID of the request.

ID value transfer with DCT

Figure 2-31 shows how the identifier field values are derived in DCT transaction messages. In this example, the data is forwarded to RN and a Snoop response is sent to HN-F with or without data.

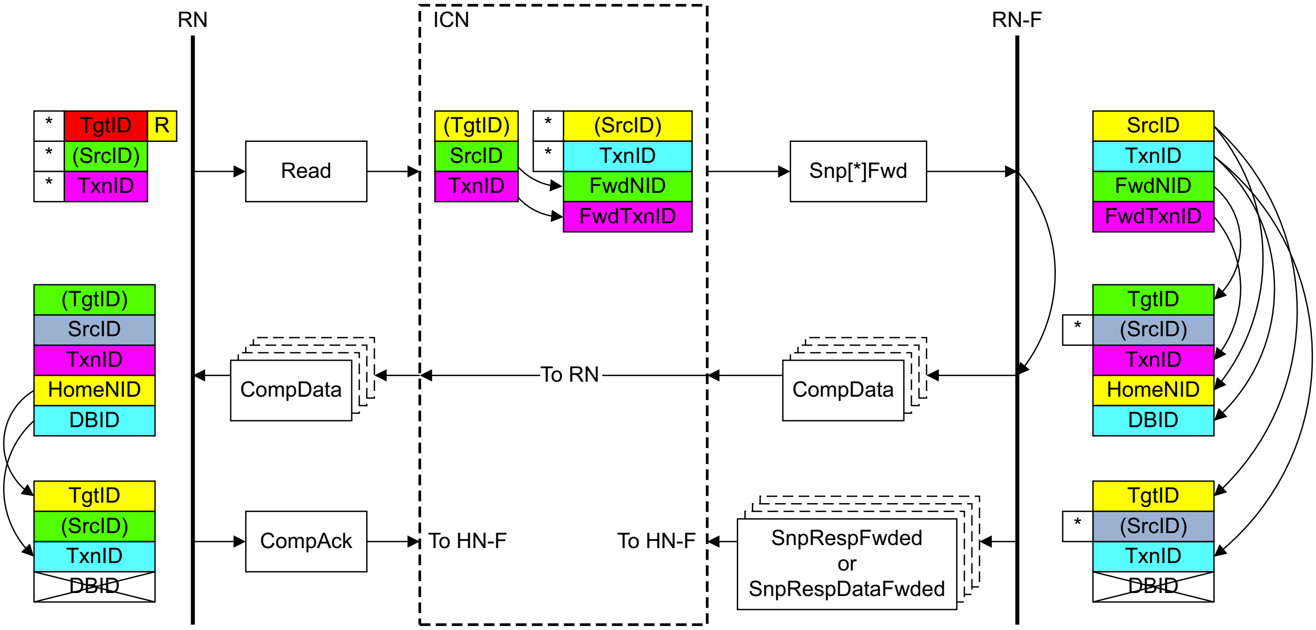


Figure 2-31 ID value transfer in a DCT transaction

The required steps in the flow that Figure 2-31 shows are:

1. The Requester starts the transaction by sending a Request packet.

   The identifier fields of the request are generated as follows:

   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be remapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a TxnID field that is unique for that Requester.

2. The recipient Home Node in the ICN generates a fowarding snoop to the RN-F node.

   The identifier fields of the snoop are generated as follows:

   - The SrcID is a fixed value for the Home.
   - The TxnID is a unique value generated by the Home.
   - The FwdNID is set to the same value as the SrcID of the original request.
   - The FwdTxnID is set to the same value as the TxnID of the original request.

3. The RN-F provides the read data.

   The identifier fields of the read data response are generated as follows:

   - The TgtID is set to the same value as the FwdNID of the snoop.
   - The SrcID is a fixed value for the RN-F.
   - The TxnID is set to the same value as the FwdTxnID of the snoop.
   - The HomeNID is set to the same value as the SrcID of the snoop.
   - The DBID is set to the same value as the TxnID of the snoop.

4. The RN-F also provides a response to Home, either with or without read data.

   The identifier fields of the response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the snoop.
   - The SrcID is a fixed value for the RN-F.
   - The TxnID is set to the same value as the TxnID of the snoop.
   - The DBID field is not valid.
5. The Requester receives the read data and sends a CompAck acknowledgment.

   The identifier fields of the CompAck are generated as follows:
   - The TgtID is set to the same value as the HomeNID of the read data.
   - The SrcID is a fixed value for the Requester. This also matches the TgtID that was received.
   - The TxnID is set to the same value as the DBID of the read data.
   - The DBID field is not valid.

Note

An optional ReadReceipt from ICN to Requester can also be included.

ID value transfer without Direct Data Transfer

This section gives an example of a Read identifier field flow without DMT or DCT and describes the use of the TxnID and DBID fields for Read transactions.

The Requester and Completer in this example are an RN and an HN-F respectively.

The identifier field flow includes an optional ReadReceipt response from the Completer, and an optional CompAck response from the Requester.

For Read transactions that include a CompAck response the DBID is used by the Completer to associate the CompAck with the original transaction.

A Read transaction that does not include a CompAck response does not require a valid DBID field in the data response.

Figure 2-32 shows the ID value transfer.

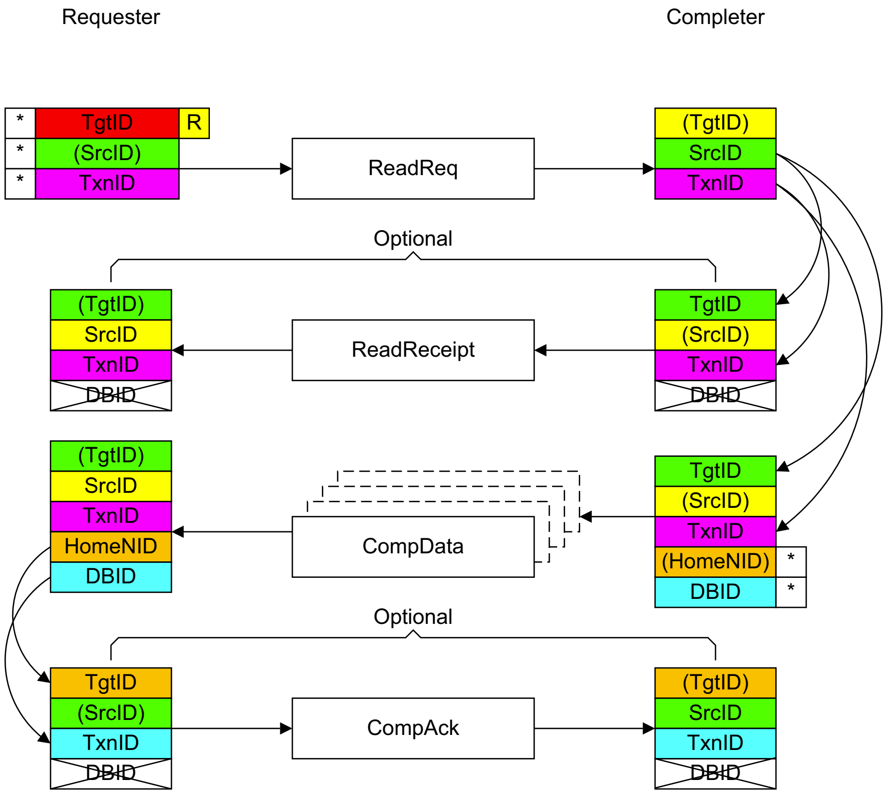


Figure 2-32 ID value transfer in a Read request with ReadReceipt and CompAck

The required steps in the flow that Figure 2-32 shows are:

1. The Requester starts the transaction by sending a Request packet.

   The identifier fields of the request are generated as follows:
   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be re-mapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a TxnID field that is unique for that Requester.

2. If the transaction includes a ReadReceipt, the Completer receives the Request packet and provides the read receipt.

   The identifier fields of the ReadReceipt response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
   - The TxnID is set to the same value as the TxnID of the request.
   - The DBID field is not valid.
3. The Completer receives the Request packet and provides the read data.

   The identifier fields of the read data response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
   - The TxnID is set to the same value as the TxnID of the request.
   - The HomeNID is a fixed value for the Completer. This also matches the TgtID received.
   - The Completer generates a unique DBID value if ExpCompAck in the request is asserted.
4. The Requester receives the read data and sends a CompAck acknowledgment.

   The identifier fields of the CompAck are generated as follows:
   - The TgtID is set to the same value as the HomeNID of the read data.
   - The SrcID is a fixed value for the Requester. This also matches the TgtID that was received.
   - The TxnID is set to the same value as the DBID of the read data.
   - The DBID field is not valid.

2.6.2 Dataless transactions

For Dataless transactions, with the exception of CleanSharedPersistSep and StashOnceSep, the use of identifier fields is similar to *ID value transfer without Direct Data Transfer* on page 2-101. The only difference is that the response from the Completer to the Requester is sent as a single packet on the CRSP channel instead of multiple packets on the RDAT channel.

For StashOnceSep transactions, the StashGroupID value is sent in the request from the RN to ICN and the value is returned in the StashDone and CompStashDone responses. The TxnID value in the StashDone response is inapplicable and must be set to zero.

The description of ID value transfer in a CleanSharedPersistSep transaction follows.

ID value transfer in a CleanSharedPersistSep transaction

Figure 2-33 shows how the identifier field values are derived in CleanSharedPersistSep transaction messages that use separate Comp and Persist.

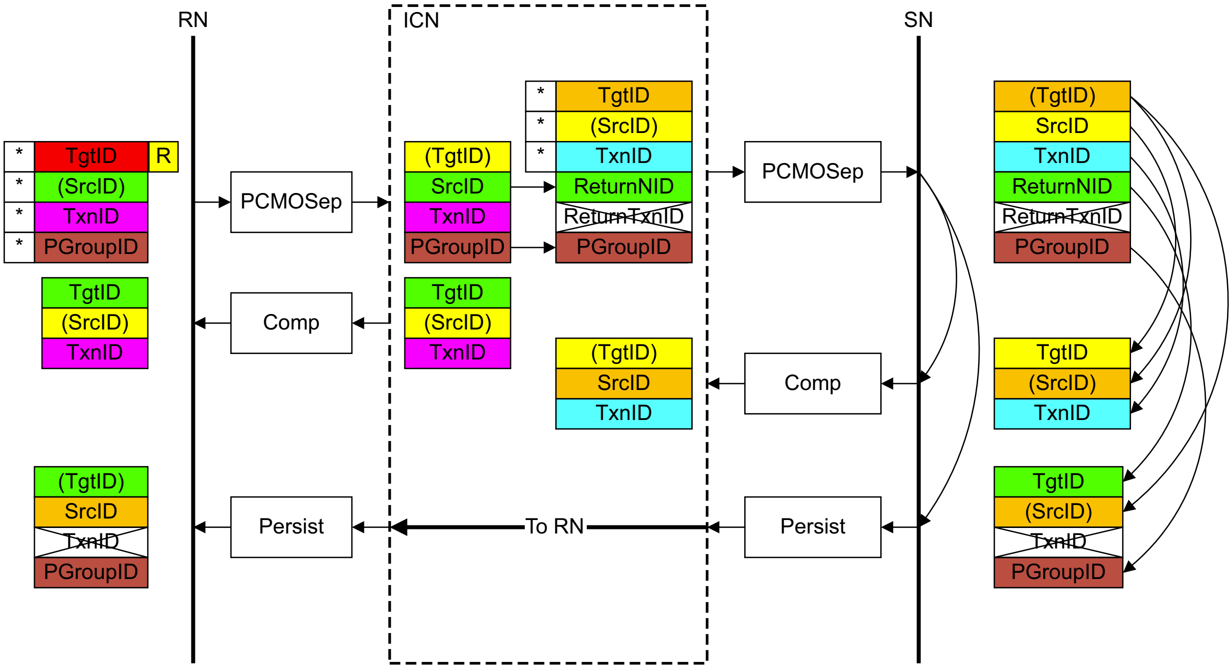


Figure 2-33 ID value transfer in a CleanSharedPersistSep transaction

The required steps in the flow that Figure 2-33 shows are:

1. The Requester starts the transaction by sending a request packet. The identifier fields of the request are generated as follows:
    - The TgtID is determined by the destination of the Request.

      Note

      The TgtID field can be remapped to a different value by the interconnect.

    - The SrcID is a fixed value for the Requester.
    - The Requester generates a TxnID value that is unique for that Requester.
      The TxnID value can be re-used by the Requester after receiving the Comp response.
    - The Requester generates a new PGroupID value, or reuses a PGroupID value currently in use.

2. The recipient Home Node in the ICN generates a request to the Slave.

   The identifier fields of the request to the Slave Node are generated as follows:
   - The TgtID is set to the value required for the Slave.
   - The SrcID is a fixed value for the Home.
   - The TxnID is a unique value generated by the Home.

     The TxnID value can be re-used by the Home after receiving the Comp response.
   - The ReturnNID is set to the same value as the SrcID of the original request.
   - The ReturnTxnID is inapplicable and must be set to zero.
   - The PGroupID is set to the same value as the PGroupID of the original request.

3. The recipient Home Node in the ICN sends a Comp response to the Requester.

   The identifier fields of the Comp response to the Requester are generated as follows:
   - The TgtID is set to the same value as the SrcID of the original request.
   - The SrcID is a fixed value for the Home.
   - The TxnID is set to the same value as the TxnID of the original request.

4. The recipient Home Node can optionally send a Persist response to the Requester.

   The identifier fields of the optional Persist response from the Home Node to the Requester, not shown in Figure 2-33 on page 2-103, are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Home Node.
   - The TxnID is inapplicable and must be set to zero.
   - The PGroupID is set to the same value as the PGroupID of the request.

   The recipient Home Node can optionally send a combined CompPersist response to the Requester, instead of separate Comp and Persist responses.

   The identifier fields in the CompPersist response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the original request.
   - The SrcID is a fixed value for the Home.
   - The TxnID is set to the same value as the TxnID of the original request.
   - The PGroupID is set to the same value as the PGroupID of the original request.

5. The Slave Node generates a Comp to the Home Node.

   The identifier fields of the Comp response from the Slave Node are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Slave.
   - The TxnID is set to the same value as the TxnID of the request.

6. The Slave Node also generates a Persist response to either to the Requester or the Home.

   The identifier fields of the Persist response from the Slave Node are generated as follows:
   - The TgtID is set to the same value as the ReturnNID of the request.
   - The SrcID is a fixed value for the Slave.
   - The TxnID is inapplicable and must be set to zero.
   - The PGroupID is set to the same value as the PGroupID of the request.

7. The Slave Node can optionally send a combined CompPersist response to the Home Node, instead of separate Comp and Persist responses if the ReturnNID and SrcID of the request are the same value.

   The identifier fields of the CompPersist response from the Slave are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the Slave.
   - The TxnID is set to the same value as the TxnID of the request.
   - The PGroupID is set to the same value as the PGroupID of the request.

2.6.3 Write transactions

This section describes the use of the TxnID and DBID fields for Write transactions:

- *CopyBack transaction*.
- *WriteNoSnp transaction* on page 2-107.
- *WriteUnique transaction* on page 2-109.
- *StashOnce or StashOnceSep transaction* on page 2-111.

CopyBack transaction

This section describes the use of the identifier fields for a CopyBack transaction.

Figure 2-34 shows the ID value transfer.

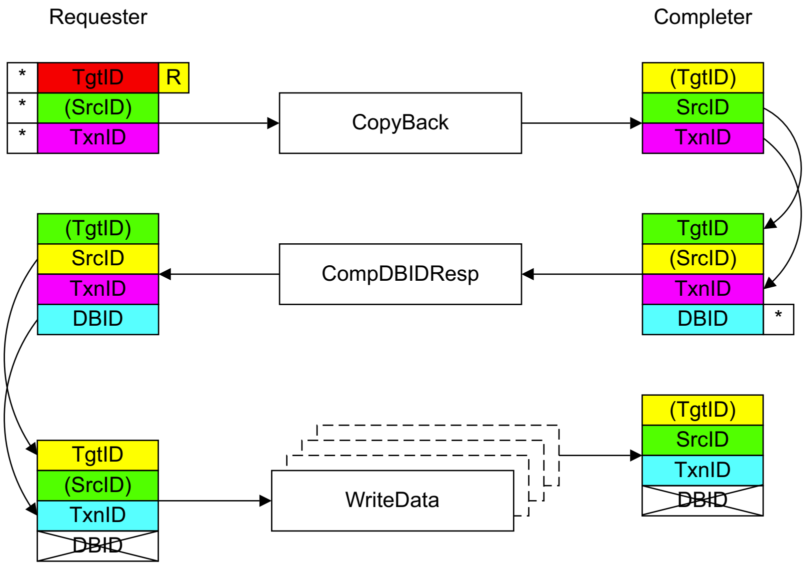


Figure 2-34 ID value transfer in a CopyBack

The required steps in the flow that Figure 2-34 shows are:

1. The Requester starts the transaction by sending a Request packet. The identifier fields of the request are generated as follows:
    - The TgtID is determined by the destination of the Request.

      Note

      The TgtID field can be remapped to a different value by the interconnect.

    - The SrcID is a fixed value for the Requester.
    - The Requester generates a unique TxnID field.
2. The Completer receives the request packet and generates a CompDBIDResp response. The identifier fields of the response are generated as follows:
    - The TgtID is set to the same value as the SrcID of the request.
    - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
    - The TxnID is set to the same value as the TxnID of the request.
    - The Completer generates a unique DBID value.

3. The Requester receives the CompDBIDResp response and sends the write data. The identifier fields of the write data are generated as follows:
    - The TgtID is set to the same value as the SrcID of the CompDBIDResp response. This can be different from the original TgtID of the request if the value was remapped by the interconnect.
    - The SrcID is a fixed value for the Requester.
    - The TxnID is set to the same value as the DBID value provided in the CompDBIDResp response.
    - The DBID field in the write data is not used.
    - The TgtID, SrcID, and TxnID fields must be the same for all write data packets.

    After receiving the CompDBIDResp response, the Requester can reuse the same TxnID value used in the request packet for another transaction.

4. The Completer receives the write data and uses the TxnID field, which now contains the DBID value that the Completer generated, to determine which transaction the write data is associated with.

    After receiving all write data packets, the Completer can reuse the same DBID value for another transaction.

WriteNoSnp transaction

This section describes the use of the identifier fields for a WriteNoSnp transaction.

Figure 2-35 does not show the Memory Tagging supported TagGroupID flow, For TagGroupID transfer details see *Write transactions* on page 12-368 in Chapter 12 *Memory Tagging*.

Figure 2-35 shows the ID value transfer for separate Comp and DBIDResp. The Completer can opportunistically combine the Comp and DBIDResp into a single CompDBIDResp response. The Requester can opportunistically combine NCBWrData with CompAck.

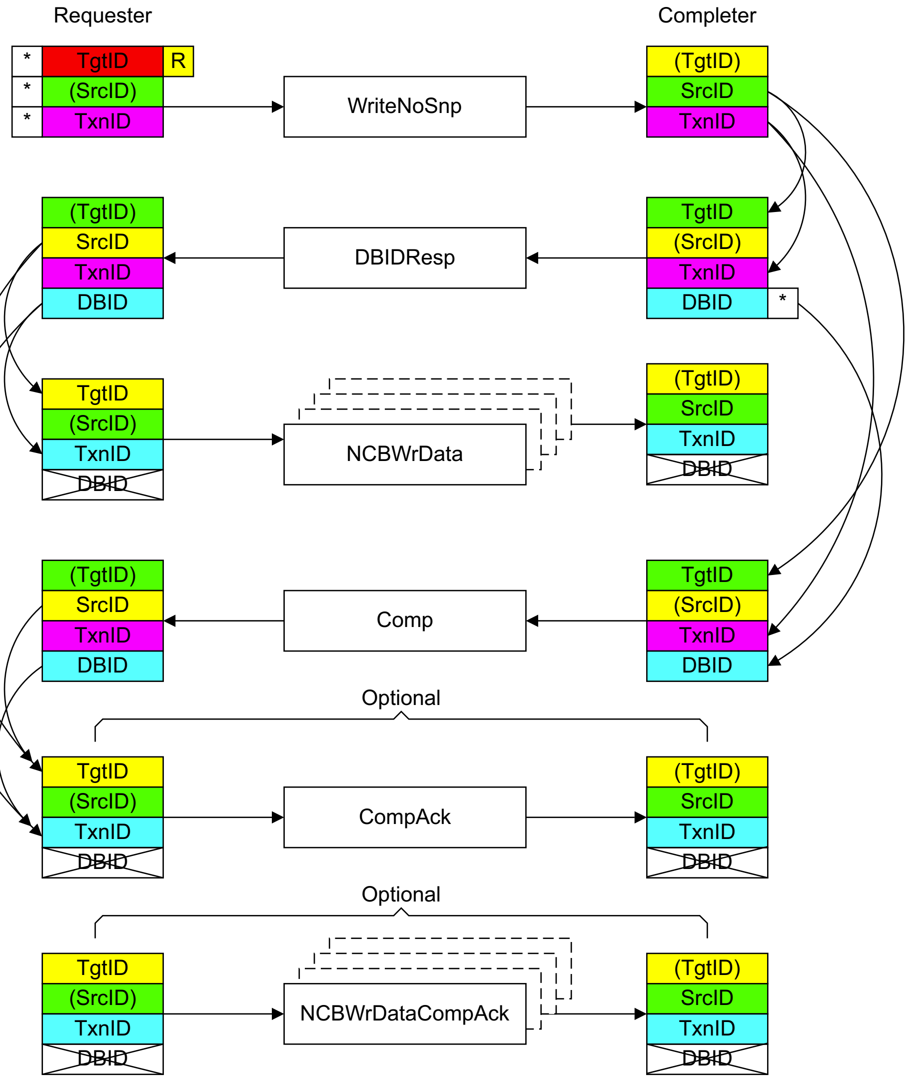


Figure 2-35 ID value transfer in a WriteNoSnp

The use of the identifier fields are the same as for a transaction with a combined response with the additional requirements that:

- The identifier fields used for the separate DBIDResp and Comp responses must be identical.
- The TxnID value must only be reused by a Requester when both the DBIDResp and Comp responses have been received.

The required steps in the flow that Figure 2-35 on page 2-107 shows are:

1. The Requester starts the transaction by sending a Request packet. The identifier fields of the request are generated as follows:
    - The TgtID is determined by the destination of the Request.

      Note

      The TgtID field can be remapped to a different value by the interconnect.

    - The SrcID is a fixed value for the Requester.
    - The Requester generates a unique TxnID field.
2. The Completer receives the Request packet and generates a DBIDResp response. The identifier fields of the response are generated as follows:
    - The TgtID is set to the same value as the SrcID of the request.
    - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
    - The TxnID is set to the same value as the TxnID of the request.
    - The Completer generates a unique DBID value.
3. The Requester receives the DBIDResp response and sends the write data. The identifier fields of the write data are generated as follows:
    - The TgtID is set to the same value as the SrcID of the DBIDResp response. This can be different from the original TgtID of the request if the value was remapped by the interconnect.
    - The SrcID is a fixed value for the Requester.
    - The TxnID is set to the same value as the DBID value provided in the DBIDResp response.
    - The DBID field in the write data is not used.
    - The TgtID, SrcID, and TxnID fields must be the same for all write data packets.
4. The Completer receives the write data and uses the TxnID field, which now contains the DBID value that the Completer generated, to determine which transaction the write data is associated with.
5. The Completer generates a Comp response when it has completed the transaction.

    The identifier fields of the Comp response must be the same as the DBIDResp response and are generated as follows:
    - The TgtID is set to the same value as the SrcID of the request.
    - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
    - The TxnID is set to the same value as the TxnID of the request.
    - The Completer uses the same DBID value as is used in the DBIDResp response.
6. The Requester sends a CompAck message, if the transaction requires it, after receiving DBIDResp or Comp. The identifier fields of the CompAck are generated as follows:
    - The TgtID is set to the same value as the SrcID of the DBIDResp or Comp response.
    - The SrcID is a fixed value for the Requester. This also matches the TgtID that was received.
    - The TxnID is set to the same value as the DBID of the DBIDResp or Comp response.
    - The DBID field is not valid.

After receiving both the Comp and DBIDResp response, the Requester can reuse the same TxnID value for another transaction.

After receiving all the write data packets, the Completer can reuse the same DBID value for another transaction.

Note

There is no ordering requirement between the separate DBIDResp and Comp responses. The specification requirement is that the values used are identical.

WriteUnique transaction

This section describes the use of the identifier fields for a WriteUnique transaction.

Figure 2-36 does not show the Memory Tagging supported TagGroupID flow, For TagGroupID transfer details see *Write transactions* on page 12-368 in Chapter 12 *Memory Tagging*.

The WriteUnique transaction can, under certain circumstances, additionally include a CompAck response from the Requester to the Completer. In this case, the additional rules for the use of the identifier fields are:

- The TgtID, SrcID, and TxnID identifier fields of the CompAck response from the Requester to the Completer must be the same as the fields used for the write data, that is:
    - The TgtID is set to the same value as the SrcID of the CompDBIDResp response. If separate Comp and DBIDResp responses are given, the TgtID is set to the same value as the SrcID of either the Comp or DBIDResp response because the SrcID value in both must be identical. However, this can be different from the original TgtID of the request if the value has been remapped by the interconnect.
    - The SrcID is a fixed value for the Requester.
    - The TxnID is set to the same value as the DBID value provided in the CompDBIDResp response. If separate Comp and DBIDResp responses are given, the TxnID is set to the same value as the DBID of either the Comp or DBIDResp response because the DBID value in both must be identical.
    - The DBID field in the WriteData and in the CompAck is not used.
    - If a combined WriteData and CompAck response is sent, then the TgtID is set to the same value as the SrcID in the Comp, DBIDResp, or CompDBIDResp, and the TxnID in the combined response is set to the same value as the DBID in Comp, DBIDResp, or CompDBIDResp.
- The Completer must receive all items of write data and the CompAck response before reusing the same DBID value for another transaction.

Figure 2-36 shows the ID value transfer with a combined CompDBIDResp response.

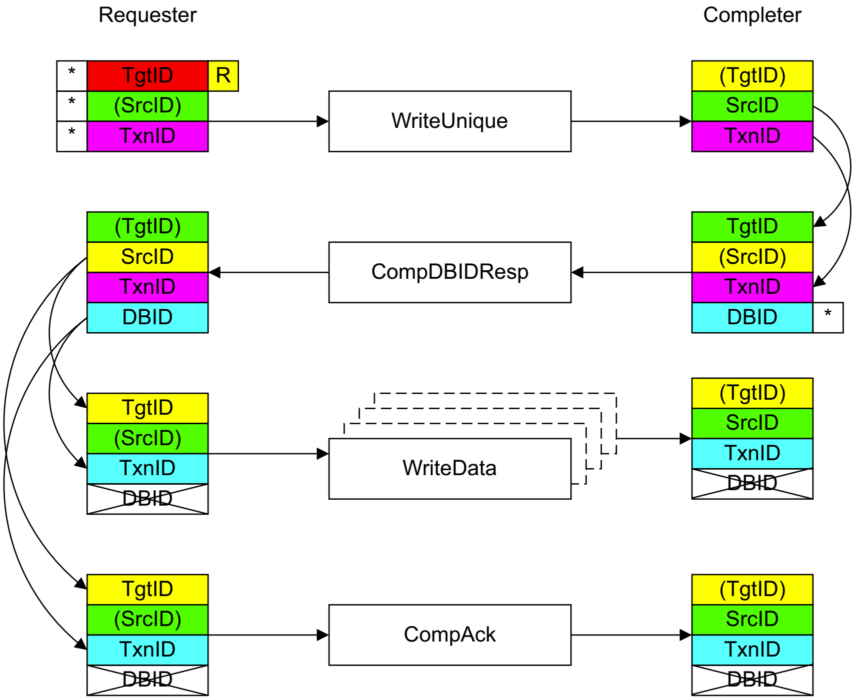


Figure 2-36 ID value transfer with a combined CompDBIDResp response

Figure 2-37 shows the ID value transfer with a combined WriteData and CompAck response.

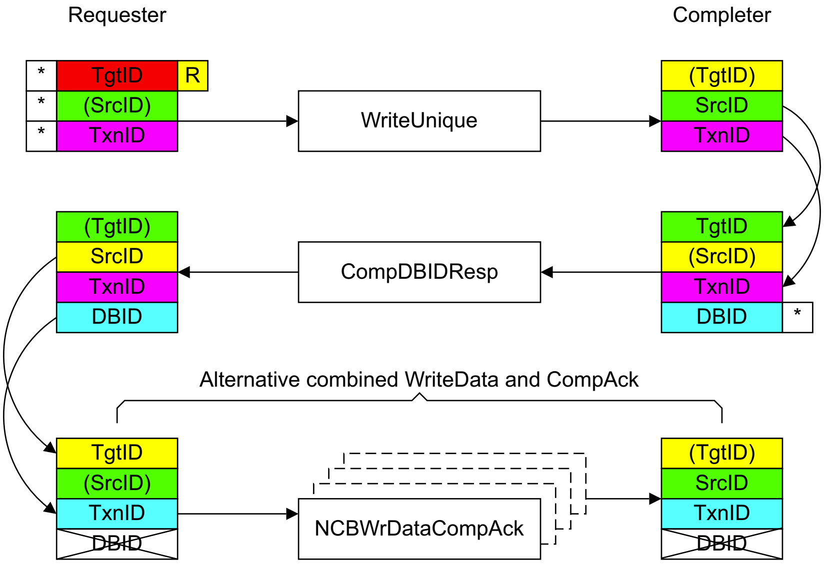


Figure 2-37 ID value transfer with a combined WriteData and CompAck response

StashOnce or StashOnceSep transaction

This section describes the use of the identifier fields for a StashOnce or StashOnceSep transaction with DataPull.

Figure 2-38 shows the ID value transfer.

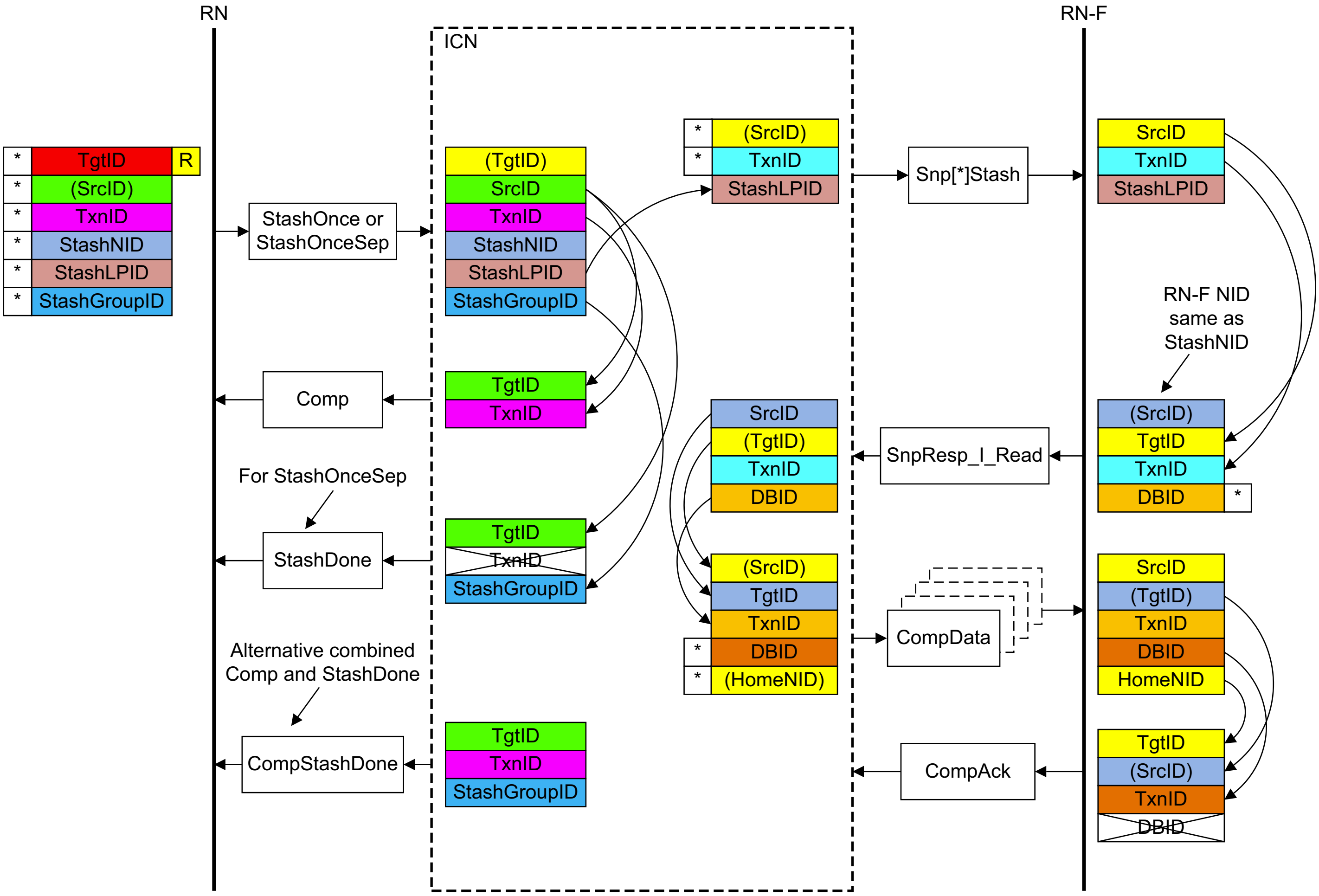


Figure 2-38 ID value transfer in a Stash transaction

The required steps in the flow that Figure 2-38 shows are:

1. The Requester starts the transaction by sending a Stash request packet.

   The identifier fields of the request are generated as follows:

   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be remapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a TxnID field that is unique for that Requester.
   - The Requester includes the StashNID field to indicate the RN-F to send the Stash to.
   - The Requester includes the StashLPID field to indicate the logical processor within the RN-F.

2. The Home Node in the ICN receives the Stash request packet and sends the Comp to the RN.

   The identifier fields of the Comp response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The TxnID is set to the same value as the TxnID of the request.

3. The Home Node in the ICN, for the StashOnceSep request, sends the StashDone response to the RN.

   The identifier fields of the StashDone response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The TxnID is not valid.
   - StashGroupID is set to the same value as the StashGroupID of the request.

   Alternatively, the Home Node in the ICN, for the StashOnceSep request, sends the combined CompStashDone response, instead of separate Comp and StashDone responses to the RN.

   The identifier fields of the CompStashDone response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The TxnID is set to the same value as the TxnID of the request.
   - The StashGroupID is set to the same value as the StashGroupID of the request.

4. The Home Node in the ICN generates a Snoop with Stash to the appropriate RN-F.

   The identifier fields of the request are generated as follows:
   - The SrcID is a fixed value for the Home.
   - The TxnID is a unique value generated by the Home.
   - The StashLPID is set to the same value as the StashLPID of the original request.

     Note

     A Snoop request does not include a TgtID field.

5. The snooped RN-F generates a Snoop response. In this example, it includes a Data Pull indication.

   The identifier fields of the Snoop response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the request.
   - The SrcID is a fixed value for the RN-F.
   - The TxnID is set to the same value as the TxnID of the request.
   - The DBID field is a unique value generated by the RN-F.

6. The Home provides the read data.

   The identifier fields of the read Data response are generated as follows:
   - The TgtID is set to the same value as the SrcID of the Snoop response.
   - The SrcID is a fixed value for the Home.

     Note

     In this example the read data is being provided by the Home.

   - The TxnID is set to the same value as the DBID of the Snoop response.
   - The DBID field is a unique value generated by Home.
   - The HomeNID is a fixed value for the Home.

7. The RN-F receives the read data and sends a CompAck acknowledgment.

   The identifier fields of the CompAck are generated as follows:
   - The TgtID is set to the same value as the HomeNID of the read data.
   - The SrcID is a fixed value for the RN-F. This also matches the TgtID received.
   - The TxnID is set to the same value as the DBID of the read data.
   - The DBID field is not valid.

2.6.4 DVMOp transaction

The use of the TgtID, SrcID, TxnID and DBID identifier fields for a DVMOp transaction is identical to that for the *WriteNoSnp transaction* on page 2-107.

2.6.5 Transaction requests with Retry

For transactions that receive a RetryAck response, there are specific rules on how the identifier fields are used. See *Request Retry* on page 2-147, for more details on the Retry mechanism, and *Protocol Credit Return transaction* on page 2-114, for rules about the return of unused credits.

Figure 2-39 shows the ID value transfer.

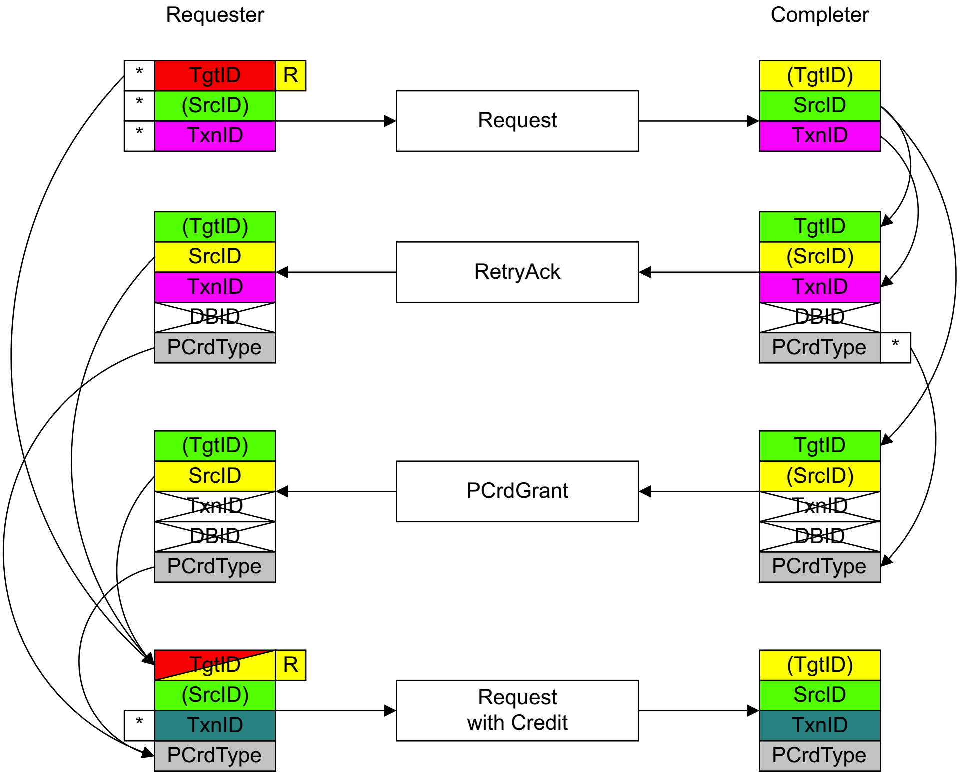


Figure 2-39 ID value transfer in a transaction request with retry

The required steps in the flow that Figure 2-39 shows are:

1. The Requester starts the transaction by sending a Request packet. The identifier fields of the request are generated as follows:
   - The TgtID is determined by the destination of the Request.

     Note

     The TgtID field can be remapped to a different value by the interconnect.

   - The SrcID is a fixed value for the Requester.
   - The Requester generates a unique TxnID field.

2. The Completer receives the Request packet and determines that it is going to send a RetryAck response. The identifier fields of the RetryAck response are generated as follows:
    - The TgtID is set to the same value as the SrcID of the request.
    - The SrcID is a fixed value for the Completer. This also matches the TgtID received.
    - The TxnID is set to the same value as the TxnID of the request.
    - The DBID field is not valid.
    - The Completer uses a PCrdType value that indicates the type of credit required to retry the transaction.
3. When the Completer is able to accept the retried transaction of a given PCrdType it sends a credit to the Requester, using the PCrdGrant response. The identifier fields of the PCrdGrant response are generated as follows:
    - The TgtID is set to the same value as the SrcID of the request.
    - The SrcID is a fixed value for the Completer. This also matches the TgtID of the request.
    - The TxnID field is not used and must be set to zero.
    - The DBID field is not used and must be set to zero.
    - The PCrdType value is set to the type required to issue the original transaction again.
4. The Requester receives the credit grant and reissues the original transaction by sending a Request packet. The identifier fields of the request are generated as follows:
    - The TgtID is set to either the same value as the SrcID of the RetryAck response, which is also the same as the SrcID of the PCrdGrant response, or the value used in the original request.
    - The SrcID is a fixed value for the Requester.
    - The Requester generates a unique TxnID field. This is permitted to be different from the original request that received a RetryAck response.
    - The PCrdType value is set to the PCrdType value in the RetryAck response to the original request, which is also the same as the PCrdType of the PCrdGrant response.

2.6.6 Protocol Credit Return transaction

A P-Credit Return transaction uses the PCrdReturn Request to return a granted, but no longer required, credit. The TgtID, SrcID, and TxnID requirements are:

- The Requester sends the Protocol Credit Return transaction by sending a PCrdReturn Request packet. The identifier fields of the request are generated as follows:
    - The TgtID must match the SrcID of the credit that was obtained.
    - The SrcID is a fixed value for the Requester.
    - The TxnID field is not used and must be set to zero.

The PCrdType must match the value of the PCrdType in the original PCrdGrant that was required to issue the original transaction again.

There is no response or use made of the DBID field associated with Protocol Credit Return transactions.

2.7 Logical Processor Identifier

The specification defines a *Logical Processor Identifier* (LPID) field within a transaction request. This field is used when a single Requester contains more than one logically separate processing agent.

The LPID must be set to the correct value for the following transactions:

- For any Non-snoopable Non-cacheable or Device access:
  - ReadNoSnp.
  - WriteNoSnp.
- For Exclusive accesses, that can be one of the following transaction types:
  - ReadClean.
  - ReadShared.
  - ReadNotSharedDirty.
  - ReadPreferUnique.
  - MakeReadUnique.
  - CleanUnique.
  - ReadNoSnp.
  - WriteNoSnp.

  See Chapter 6 *Exclusive Accesses* for further details.

Note

For other transactions, the LPID value can be used to indicate the original logical processor that caused a transaction to be issued. However, this information is not required by this specification and is optional.

2.8 Ordering

This section describes the mechanisms that the protocol includes to support system ordering requirements. It contains the following subsections:

- *Multi-copy atomicity*.
- *Completion Response and Ordering*.
- *Completion acknowledgement* on page 2-117.
- *Transaction ordering* on page 2-120.

For the meaning of the terms EWA, Device, and Cacheable see *Memory Attributes* on page 2-128.

2.8.1 Multi-copy atomicity

This specification requires a multi-copy atomic architecture. All compliant components must ensure that all write-type requests are multi-copy atomic. A write is defined as multi-copy atomic if both of the following conditions are true:

- All writes to the same location are serialized, that is, they are observed in the same order by all Requesters, although some Requesters might not observe all of the writes.

- A read of a location does not return the value of a write until all Requesters observe that write.

In this specification, two addresses are considered to be the same with respect to coherence, observability, and hazarding if their cache line addresses and NS attribute are the same.

2.8.2 Completion Response and Ordering

To guarantee the ordering of a transaction with respect to later transactions, either from the same agent or from another agent, the Comp, RespSepData, CompData, or Persist response is used as follows:

- For a Read transaction to a Non-cacheable or Device location, a RespSepData or CompData response guarantees that the transaction is observable to a later transaction from any agent to the same endpoint address range. The size of the endpoint address range is IMPLEMENTATION DEFINED.

- For a Read transaction to a Cacheable location, a CompData or DataSepResp response guarantees that the transaction is observable to a later transaction from any agent to the same location.

- For a Read transaction to a Cacheable location, a RespSepData response guarantees that no earlier transaction will send a snoop to this Requester, and all later transactions will send a snoop only if required after the Home receives the CompAck response for this transaction.

- For a Dataless transaction, except for StashOnceSep, a Comp response guarantees that the transaction is observable to a later transaction from any agent to the same memory location.

- In addition, for CleanSharedPersist transactions, a Comp response guarantees that any data written earlier to the same memory location is made persistent.

- For Combined Write transactions a CompCMO response guarantees that the CMO and PCMO operation is observable to a later transaction from any agent to the same memory location.

- For CleanSharedPersistSep transactions, either standalone or in a Combined Write, a Persist response guarantees that any data written earlier to the same memory location is made persistent.

- For a StashOnceSep transaction, a Comp response guarantees that the Completer has accepted the request and will not send a RetryAck response.

- For a StashOnceSep transaction, a StashDone response guarantees that the transaction is observable to a later transaction from any agent to the same memory location.

- For a Write or an Atomic transaction to Non-cacheable or Device nRnE or Device nRE or Device RE, a Comp or CompData response guarantees that the transaction is observable to a later transaction from any agent to the same endpoint address range.

- For a Write or Atomic transaction to a Cacheable location, a Comp or CompData response guarantees that the transaction is observable to a later transaction, from any agent, to the same location.

In a combined OWO Write request, if the write is canceled, that is, the Requester sends a WriteDataCancel, then the required cache maintenance on the write data is not carried out. The Requester must resend both the Write request and the CMO:

- If the combined request that had its write canceled is a Write with PCMO then:
  - The Persist response for the combined request does not indicate that the write data is made persistent.
  - The Requester must resend the PCMO either combined with the resent Write request or after the resent Write request succeeds.
- If the combined request that had its write canceled is a Write with CMO then:
  - The Requester must resend the CMO either combined with the resent Write request or after the resent Write request succeeds.

Note

- The size of an endpoint address range is IMPLEMENTATION DEFINED. Typically, this is:
  - The size of a peripheral device, for a region used for peripherals.
  - The size of a cache line, for a region used for memory.
- A Cacheable location can be determined by the assertion of the MemAttr[2] Cacheable bit in the request. A Non-cacheable or Device location can be determined by the deassertion of the MemAttr[2] Cacheable bit in the request.

A component must only give a Comp or CompDBIDResp response when it is guaranteed that all observers will see the result of the atomic operation.

A Comp response in a write that is canceled only implies that the transaction loop is completed and makes no statement regarding the completion of coherency action initiated by the write. Therefore, the Completer is permitted to send a Comp as soon as it receives a WriteDataCancel response without dependency on either the processing of the write request or the completion of any snoops sent due to the write.

2.8.3 Completion acknowledgement

The relative ordering of transactions issued by a Requester, and Snoop transactions caused by transactions from different Requesters, is controlled by the use of a Completion Acknowledgment response. This ensures that a Snoop transaction that is ordered after the transaction from the Requester is guaranteed to be received after the transaction response.

The sequencing of the completion of a Read transaction and the sending of CompAck is as follows:

1. An RN-F sends a CompAck after receiving Comp, RespSepData or CompData, or both RespSepData and DataSepResp.
2. An HN-F, except in the case of ReadOnce*, waits for CompAck before sending a subsequent snoop to the same address. For CopyBack transactions, WriteData acts as an implicit CompAck and an HN-F must wait for WriteData before sending a snoop to the same address.

This sequence guarantees that an RN-F receives completion for a transaction and a snoop to the same cache line in the same order as they are sent from an HN-F. This ensures transactions to the same cache line are observed in the correct order.

When an RN-F has a transaction in progress that uses CompAck, except for ReadNoSnp and ReadOnce*, then it is guaranteed not to receive a Snoop request to the same address between the point that it receives Comp and the point that it sends CompAck.

For WriteNoSnp and WriteUnique transactions that require a CompAck message, an RN sends the CompAck after receiving the Comp, DBIDResp, or CompDBIDResp response.

The use of CompAck for a transaction is determined by the Requester setting the ExpCompAck field in the original request. The rules for an RN setting the ExpCompAck field and generating a CompAck response are as follows:

- An RN-F must include a CompAck response in all Read transactions except ReadNoSnp and ReadOnce*.
- Although not required, an RN-F is permitted to include a CompAck response in ReadNoSnp and ReadOnce* transactions.
- An RN-F must not include a CompAck response in StashOnce*, CMO, Atomic or Evict transactions.
- An RN-I or RN-D is permitted, but not required, to include a CompAck response in Read transactions.
- An RN-I or RN-D must not include a CompAck response in Dataless or Atomic transactions.
- An RN that wants to make use of DMT must include a CompAck response in ordered ReadNoSnp and ReadOnce* transactions.
- For Write transactions, CompAck can only be used for WriteUnique and WriteNoSnp transactions when they require Ordered Write Observation guarantees. See *Streaming Ordered Write transactions* on page 2-124.
- For WriteEvictOrEvict, ExpCompAck must be set to one indicating the transaction will include a CompAck when the Completer sends a Comp instead of a CompDBIDResp.

For transactions between an RN and an HN, where the HN is the Completer, the HN must support the use of CompAck for all transactions that are required or permitted to use CompAck.

An SN is not required to support the use of CompAck.

A Requester, such as an HN-F or HN-I that communicates with an SN-F or SN-I respectively, must not send a CompAck response.

Table 2-9 shows the Request types that require a CompAck response, and the corresponding Requester types that are required to provide that response.

Table 2-9 Requester CompAck requirement

Request type	CompAck Required	
	RN-F	RN-D, RN-I
ReadNoSnp	Optional	Optional
ReadOnce*	Optional	Optional
ReadClean	Yes	-
ReadNotSharedDirty	Yes	-
ReadShared	Yes	-
ReadUnique	Yes	-
ReadPreferUnique	Yes	-
MakeReadUnique	Yes	-
CleanUnique	Yes	-
MakeUnique	Yes	-
CleanShared	No	No
CleanSharedPersist*	No	No
CleanInvalid	No	No

Table 2-9 Requester CompAck requirement (continued)

Request type	CompAck Required	
	RN-F	RN-D, RN-I
MakeInvalid	No	No
WriteBack	No	-
WriteClean	No	-
WriteUnique	Optional	Optional
WriteUniqueZero	No	No
Evict	No	-
WriteEvictFull	No	-
WriteEvictOrEvict	Yes	-
WriteNoSnp	Optional	Optional
WriteNoSnpZero	No	No
Atomics	No	No
StashOnce*	No	No

In a Combined Write transaction, the CompAck requirement is the same as the CompAck requirement for the type of Write in the Combined Write transaction.

2.8.4 Ordering semantics of RespSepData and DataSepResp

When a Requester receives the first DataSepResp it can consider the Read transaction to be globally observed, as there is no action which can modify the read data received.

A Home must ensure that all required Snoop transactions are completed before initiating a transaction, such as a ReadNoSnpSep, which could result in the DataSepResp response being sent to the Requester.

When a Requester receives a RespSepData response from Home, the request that it applies to has been ordered at Home and the Requester will not receive any snoops for transactions that are scheduled before it. The Home, before sending RespSepData response to the Requester, must ensure that no Snoop transactions are outstanding to that Requester to the same address. Receiving of RespSepData does not guarantee that Home has completed snooping of other agents in the system.

When the Requester gives a CompAck acknowledgement, this Requester is indicating that it will accept responsibility to hazard snoops for any transaction that is scheduled after it. The following rules apply:

- For all transactions, except as described immediately below, the CompAck acknowledgement must be sent after the RespSepData response is received. It is permitted, but not required, to wait for the DataSepResp response before the CompAck acknowledgement is given.
- For ReadOnce and ReadNoSnp transactions with an ordering requirement, that is, Order field is set to `0b10` or `0b11` and ExpCompAck field is asserted, it is required that the CompAck acknowledgement is given only after both DataSepResp and RespSepData responses are received.
- The Requester must wait to receive both RespSepData and DataSepResp before issuing another request to the same address.

Note

This specification requires that CompAck must not be given when only DataSepResp is received.

2.8.5 Transaction ordering

In addition to using a Comp response to order a sequence of requests from a Requester, this specification also defines mechanisms for ordering of requests between an RN, HN pair and a HN-I, SN-I pair. Between an HN-F, SN-F pair the order field is used to obtain a request accepted acknowledgment.

Requester Order between an RN, HN pair and a HN-I, SN-I pair is supported by the Order field in a request. The Order field indicates that the transaction requires one of the following forms of ordering:

Request Order This guarantees the order of multiple transactions, from the same agent, to the same address location.

Endpoint Order This guarantees the order of multiple transactions, from the same agent, to the same endpoint address range.

Ordered Write Observation

This guarantees the observation order by other agents in the system, for a sequence of write transactions from a single agent.

Request Accepted

This guarantees that the Completer will send a positive acknowledgement only when it has accepted the request.

Table 2-10 shows the Order field encodings.

Table 2-10 Order field encodings

Order[1:0]	Description	Notes	Between pairs
`0b00`	No ordering required	-	All pairs
`0b01`	Request Accepted	Applicable in Read request from HN-F to SN-F and HN-I to SN-I only. Reserved in all other cases.	HN-F to SN-F HN-I to SN-I
`0b10`	Request Order/Ordered Write Observation required	Reserved in Read requests from HN-F to SN-F.	RN to HN HN-I to SN-I
`0b11`	Endpoint Order required		

Ordering requirements

A Requester that changes the ordering requirements of a transaction to a stronger ordering requirement, is required to be consistent in changing the ordering requirement of Request Order to Endpoint Order on all its transactions.

The Order field must only be set to a non-zero value for the following transactions:

- ReadNoSnp.
- ReadNoSnpSep.
- ReadOnce*.
- WriteNoSnp.
- WriteNoSnp*CMO.
- WriteNoSnpZero.
- WriteUnique.
- WriteUnique*CMO.
- WriteUniqueZero.
- Atomic.

When a ReadNoSnp or ReadOnce* transaction requires Request Order or Endpoint Order:

- The Requester requires a ReadReceipt to determine when it can send the next ordered request.
- At the Completer a ReadReceipt means the request has reached the next ordering point that will maintain requests in the order they were received:
  - For requests that require Request Order it will maintain order between requests to the same address from the same source.
  - For requests that require Endpoint Order it will maintain order between requests to the same endpoint address range from the same source.
- A completer that is capable of sending separate Non-data and Data-only responses can send RespSepData response instead of ReadReceipt and achieve the same functional behavior.

When a WriteNoSnp, WriteNoSnpZero or a Non-snoopable Atomic transaction requires Request Order or Endpoint Order:

- The Requester requires a DBIDResp or DBIDRespOrd to determine when it can send the next ordered request.
- The Completer sending a DBIDResp or DBIDRespOrd response means that a data buffer is available, and that the Write request has reached a PoS that will maintain requests in the order they were received:
  - For requests that require Request Order it will maintain order between requests to the same address from the same source.

  - For requests that require Endpoint Order it will maintain order between requests to the same endpoint address range from the same source.

When a WriteUnique transaction without ExpCompAck asserted, or a WriteUniqueZero or a Snoopable Atomic transaction require Request Order:

- The Requester requires a DBIDResp or DBIDRespOrd to determine when it can send the next ordered request.
- The Completer sending a DBIDResp or DBIDRespOrd response means that it will maintain order between requests to the same address from the same source.

Additionally, when a Completer sends DBIDRespOrd for a request with a no order or Request Order requirement, the Completer guarantees to order all subsequent no order or Request Order received requests to the same address from the same source against this request, where these later received requests are of any transaction type, not necessarily Write transactions. When the write includes a CMO, the order is guaranteed against both the write and the CMO.

When a WriteUnique or WriteNoSnp transaction requires Ordered Write Observation:

- CompAck is required. The RN must assert ExpCompAck.
- The RN requires a DBIDResp or DBIDRespOrd.
- The Completer is a PoS. A PoS sending DBIDResp or DBIDRespOrd means:
  - A data buffer is available.
  - The PoS guarantees that the completion of the coherence action on this write does not depend on completion of the coherence action on a subsequent write that requires Ordered Write Observation.
  - The write is not made visible until CompAck is received.

All architectural mechanisms applicable to increasing streaming efficiency and corresponding constraints are defined in *Streaming Ordered Write transactions* on page 2-124.

When a ReadNoSnp or ReadNoSnpSep has the Order field set to `0b01`, a ReadReceipt response from the Completer guarantees that the Completer has accepted the request and will not send a RetryAck response.

Read Request Order example

Figure 2-40 shows the request ordering of three Read requests.

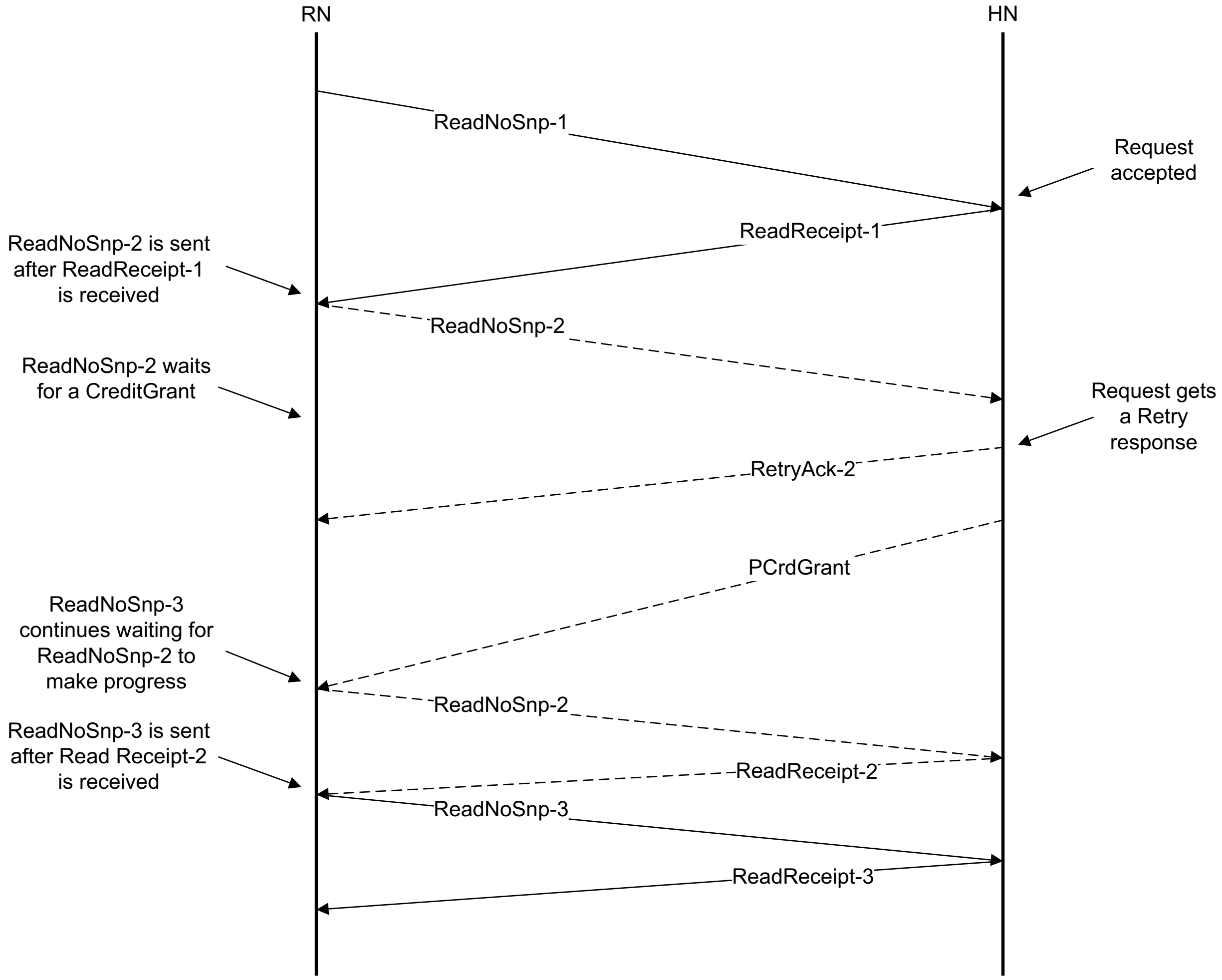


Figure 2-40 Series of ordered Read requests

Three ordered requests are sent from RN to HN as follows:

1. RN sends the ReadNoSnp-1 request to HN.
2. HN accepts the request and returns the ReadReceipt-1 response to RN.
3. After the ReadReceipt-1 response is received, RN sends the ReadNoSnp-2 request to HN.
4. HN cannot immediately accept the ReadNoSnp-2 request and returns the RetryAck-2 response to RN.
5. RN must now wait for a PCrdGrant to be sent from HN before resending the ReadNoSnp-2 request. RN does not send ReadNoSnp-3 at this point, as it wants to order ReadNoSnp-3 behind ReadNoSnp-2. This ordering requires that ReadNoSnp-2 must be accepted at HN before ReadNoSnp-3 is sent to HN.
6. After receipt of an appropriate PCrdGrant, RN resends the ReadNoSnp-2 request.
7. HN accepts the request and returns a ReadReceipt-2 response to RN.
8. After receipt of the ReadReceipt-2 response, RN sends the ReadNoSnp-3 request to HN.
9. HN accepts the request and returns the ReadReceipt-3 response to RN.
10. HN completes the Request transactions by sending a combined Completion and Data response to the RN for each request.

Note

Figure 2-40 on page 2-123 shows a single ordered stream of three reads from RN. However, an RN can have multiple streams of reads, in which case requests must be ordered within a stream, but ordering dependency does not exist between streams. One example of this is when the streams are from different threads within the RN, in which case, the RN waits for the ReadReceipt of the previous request from the same thread only before sending out the next ordered request from that stream.

CopyBack Request order

An RN-F must wait for the CompDBIDResp response to be received for an outstanding CopyBack transaction before issuing another request to the same cache line. It is permitted for an Atomic transaction with SnoopMe asserted to be issued before the CompDBID response is received for an outstanding CopyBack to the same cache line.

Streaming Ordered Write transactions

The architectural mechanisms applicable to increasing *Ordered Write Observation* (OWO) Write streaming efficiency, and the corresponding constraints, are applicable to WriteUnique and WriteNoSnp transactions only.

If a Requester requires a sequence of Write transactions to be observed in the same order as they are issued, then the Requester can wait for completion for a write before issuing the next write in the sequence. Such an observation ordering is typically termed Ordered Write Observation. This specification provides a mechanism termed Streaming Ordered Writes to more efficiently stream such ordered Write transactions.

The Streaming Ordered Write mechanism relies on the use of the Ordered Write Observation ordering requirement and CompAck. Responsibilities of Requesters and HN-F when utilizing the Streaming Ordered Write solution are:

- The Requester must set the Order field to require Ordered Write Observation and set ExpCompAck on the Write request.
- The Ordered Write Observation requirement in a Write request indicates to the HN-F that the completion of coherence action on this write must not depend on completion of coherence action on a subsequent write.
- The Requester must wait for DBIDResp, DBIDRespOrd, CompDBIDResp, or Comp for a Write transaction before sending the next Write request.

- The Requester, after receiving DBIDResp, DBIDRespOrd, CompDBIDResp or Comp for the corresponding write, as well as Comp responses for all earlier related ordered writes, must send a CompAck response. If write data is to be sent, then the Requester is permitted to combine the CompAck with the WriteData response into a NCBWrDataCompAck response. When the Requester combines CompAck with the WriteData response it must send a combined response for all WriteData transfers. The method by which a Requester determines if a group of ordered writes are related is IMPLEMENTATION DEFINED.

  Note

  Waiting to send CompAck until all prior ordered writes have received their Comp responses ensures that they have completed their operations at their respective HN-Fs and any Requester observing the write for which CompAck is sent will also observe all prior ordered writes.

- The Requester that receives DBIDResp* and is ready to send CompAck must not wait for Comp to send CompAck.
- HN-F must wait for a CompAck response from RN before deallocating a Write transaction and making the write visible to other observers.

Optimized Streaming Ordered Write transactions

The writes in this section refer to WriteUnique or WriteNoSnp only. The Streaming Ordered Writes mechanism can be further optimized. If a previously sent write is to a different target, then the Requester does not need to wait for the DBIDResp* for the request before sending the next ordered write. However, if the interconnect can remap the TgtID, then the Requester must presume that all Write transactions are targeting the same HN-F and must not use the optimized version of the Streaming Ordered Writes flow.

An implementation using an optimized or non-optimized Streaming Ordered Writes solution must avoid deadlock and livelock situations.

Note

- A technique for avoiding resource related deadlock or livelock issues is to limit Streaming Ordered Writes optimization to one Requester in the system. All other Requesters in the system can use the Streaming Ordered Writes solution without the optimization.
- In a typical system, the optimized Streaming Ordered Writes solution is most beneficial to an RN-I that is a conduit for PCIe style, non-relaxed order, Snoopable writes. In most systems, one RN-I hosting this type of PCIe traffic is adequate.
- Optimized Streaming Ordered Writes can be used by more than one Requester by making use of WriteDataCancel messages to avoid Resource related deadlocks and livelocks.

Figure 2-41 shows a typical transaction flow in which an RN-I uses Streaming Ordered WriteUnique transactions. This flow prevents a read acquiring the new value of Write-B before Write-A has completed.

Note

For clarity, in Figure 2-41 the Write-B DBIDResp* and the NCBWrData flow is omitted.

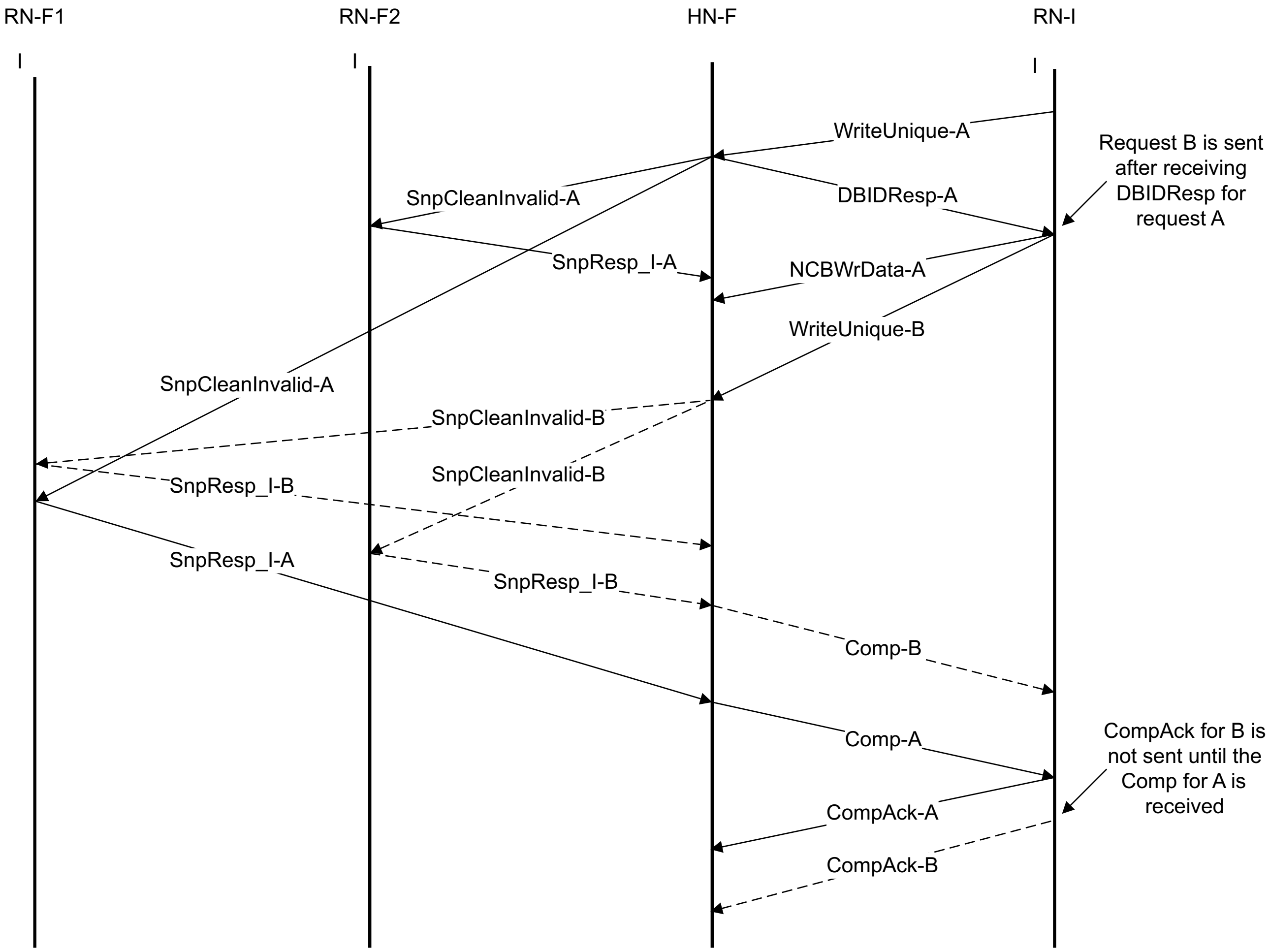


Figure 2-41 Streaming Ordered WriteUnique transactions flow

2.9 Address, Control, and Data

A transaction includes attributes defining the manner in which the transaction is handled by the interconnect. These include the address, memory attributes, snoop attributes, and data formatting. Each attribute is defined in this section.

In this section, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

2.9.1 Address

This specification supports:

- *Physical Address* (PA) of 44 to 52 bits, in one bit increments.
- *Virtual Address* (VA) of 49 to 53 bits.

The REQ and SNP packet address fields are specified as follows:

- REQ channel: Addr[(MPA-1):0]
- SNP channel: Addr[(MPA-1):3]

MPA is the maximum PA supported.

Table 2-11 shows the relationship between the physical address field width and the supported virtual address.

Table 2-11 Addr field width and supported PA and VA size

REQ Addr field width in bits	Maximum address supported in bits	
	PA	VA
44	44	49
45	45	51
46 to 52	46 to 52	53

See *DVMOp and SnpDVMOp packet* on page 8-307 for DVM payload mapping in the REQ and SNP fields with different ADDR field widths.

The Req_Addr_Width parameter is used to specify the maximum physical address in bits that is supported by a component. Valid values for this parameter are 44 to 52, when not specified, the parameter takes the default value of 44.

The REQ and SNP channel messages address field in the REQ channel is a 44-bit to 52-bit field labeled Addr[(43-51):0] and in the SNP channel it is a 41-bit to 49-bit field labeled Addr[(43-51):3]. This field is used by the different message types as follows:

- For Read, PrefetchTgt, Dataless, Write, and Atomic transactions the Addr field includes the address of the memory location being accessed.
- For a Snoop request, except SnpDVMOp, the field includes the address of the location being snooped:
  - Addr[(43-51):6] is the cache line address and is sufficient to uniquely identify the cache line to be accessed by the snoop.
  - Addr[5:4] identifies the critical chunk being accessed by the transaction. See *Critical Chunk Identifier* on page 2-141. This specification recommends that the snooped cache returns the data in wrap order with the critical chunk returned first.

    Note

    Addr[3] is supplied, but is not used.

- For a DVMOp and SnpDVMOp request the Addr field is used to carry information related to a DVM operation. See Chapter 8 *DVM Operations*.
- The Addr field value is not used for the PcrdReturn transaction and must be set to zero.

2.9.2 Non-secure bit

Secure and Non-secure transactions are defined to support Secure and Non-secure operating states.

This bit is defined so that when it is asserted the transaction is identified as a Non-secure transaction.

For Snoopable transactions this field can be considered as an additional address bit that defines two address spaces, a Secure address space, and a Non-secure address space. Any aliasing between the Secure and Non-secure address spaces must be handled correctly.

Note

Hardware coherency does not manage coherency between Non-Secure and Secure address spaces.

The NS assertion requirements are:

- Can be asserted in any Read, Dataless, Write and Atomic transaction.
- Can be asserted in PrefetchTgt transaction.
- Is not applicable in the DVMOp or PCrdReturn transaction, and must be set to zero.

2.9.3 Memory Attributes

The *Memory Attributes* (MemAttr) consist of *Early Write Acknowledgement* (EWA), Device, Cacheable, and Allocate.

EWA

EWA indicates whether the write completion response for a transaction:

- Is permitted to come from an intermediate point in the interconnect, such as a Home Node.
- Must come from the final endpoint that a transaction is destined for.

If EWA is asserted, the write completion response for the transaction can come from an intermediate point or from the endpoint. A completion that comes from an intermediate point must provide the same guarantees required by a Comp as described in *Completion Response and Ordering* on page 2-116.

If EWA is deasserted, the write completion response for the transaction must come from the endpoint.

Note

It is permitted for an implementation not to use the EWA attribute, in this case completion must be given from the endpoint.

The EWA assertion requirements are:

- Can take any value in a ReadNoSnp and ReadNoSnpSep transaction.
- Can take any value in a WriteNoSnp transaction.
- Can take any value in CMO transactions.
- Can take any value in Atomic transactions.
- Must be asserted in any Read or Dataless transaction that is not a ReadNoSnp, ReadNoSnpSep, or CMO transaction.
- Must be asserted in any Write transaction that is not a WriteNoSnp transaction.
- Is inapplicable and must be set to zero in the DVMOp or PCrdReturn transactions.
- Is inapplicable and can take any value in the PrefetchTgt transaction.

Device

Device attribute indicates if the memory type is either Device or Normal.

Device memory type

Device memory type must be used for locations that exhibit side-effects. Use of Device memory type for locations that do not exhibit side-effects is permitted.

The requirements for a transaction to a Device type memory location are:

- A Read transaction must not read more data than requested.
- Prefetching from a Device memory location is not permitted.
- A read must get its data from the endpoint. A read must not be forwarded data from a write to the same address location that completed at an intermediate point.
- Combining requests to different locations into one request, or combining different requests to the same location into one request, is not permitted.
- Writes must not be merged.
- Writes to Device memory that obtain completion from an intermediate point must make the write data visible to the endpoint in a timely manner.

Accesses to Device memory must use the following types, exclusive variants are permitted:

- Read accesses to a Device memory location must use ReadNoSnp.
- Write accesses to a Device memory location must use either WriteNoSnpFull or WriteNoSnpPtl.
- CMO transactions are permitted to Device memory locations.
- Atomic transactions are permitted to Device memory locations.
- The PrefetchTgt transaction is not permitted to Device memory locations. The MemAttr field is inapplicable and can take any value in the PrefetchTgt transaction.

Normal memory type

Normal memory type is appropriate for memory locations that do not exhibit side-effects.

Accesses to Normal memory do not have the same restrictions regarding prefetching or forwarding as Device type memory:

- A Read transaction that has EWA asserted can obtain read data from a Write transaction that has sent its completion from an intermediate point and is to the same address location.
- Writes can be merged.

Any Read, Dataless, Write, PrefetchTgt or Atomic transaction type can be used to access a Normal memory location. The transaction type used is determined by the memory operation to be accomplished, and the Snoopable attributes.

Cacheable

The Cacheable attribute indicates if a transaction must perform a cache lookup:

- When Cacheable is asserted the transaction must perform a cache lookup.
- When Cacheable is deasserted the transaction must access the final destination.

The Cacheable attribute value requirements are:

- Must not be asserted for any Device memory transaction.
- Must be asserted for any Read transaction except for ReadNoSnp and ReadNoSnpSep.
- Must be asserted for any Dataless transaction except for CleanShared, CleanSharedPersist*, CleanInvalid, MakeInvalid.
- Must be asserted for any Write transaction except WriteNoSnpFull and WriteNoSnpPtl.
- Can take any value for ReadNoSnp, ReadNoSnpSep, WriteNoSnpFull, and WriteNoSnpPtl to a Normal memory location.
- Can take any value for CleanShared, CleanSharedPersist*, CleanInvalid and MakeInvalid.
- Can take any value for an Atomic transaction.
- Is inapplicable and must be set to zero in DVMOp and PCrdReturn transactions.
- Is inapplicable and can take any value in the PrefetchTgt transaction.

Note

In a transaction that can take any Cacheable value, the value is typically determined from the page table attributes.

Allocate

The Allocate attribute is a an allocation hint. It indicates the recommended allocation policy for a transaction:

- If Allocate is asserted, it is recommended that the transaction is allocated into the cache for performance reasons. However, it is permitted to not allocate the transaction.
- If Allocate is deasserted, it is recommended that the transaction is not allocated into the cache for performance reasons. However, it is permitted to allocate the transaction.

The Allocate attribute value requirements are:

- Can be asserted for transactions that have the Cacheable attribute asserted.
- Must be asserted for the WriteEvictFull transaction.

  Note

  A Requester can convert a WriteEvictFull with the Allocate bit not asserted to an Evict transaction.

- Must not be asserted for Device memory transactions.
- Must not be asserted for Normal Non-cacheable memory transactions.
- Is inapplicable and must be set to zero in DVMOp, PCrdReturn and Evict transactions.
- Is inapplicable and can take any value in the PrefetchTgt transaction.

Propagation of Attr

The MemAttr bits EWA, Device, Cacheable, and Allocate, must be preserved on a request from HN to SN that is sent in response to a request to HN. The only exception to this rule is when the downstream memory is known to be Normal, then the Device field value can be set to `0b0` to indicate Normal.

The SnpAttr attribute bit value in a request from HN to SN must always be set to `0b0`.

When the write and CMO transactions in a received Combined Write request transaction are separated, the Write transaction inherits the MemAttr and SnpAttr values of the original combined request. The separated CMO transaction SnpAttr and Cacheable bits must be set to the most pervasive, so as to effect all caches at RN-F nodes and caches downstream.

For a ReadNoSnp or WriteNoSnp generated within the interconnect due to a Prefetch from Home, or an eviction from the System cache:

- MemAttr bits EWA, Cacheable, and Allocate must all be set to `0b1`.
- Device field value must be set to `0b0` to indicate Normal.
- SnpAttr field value must be set to `0b0` to indicate Non-snoopable.

2.9.4 Transaction attribute combinations

Table 2-12 lists the legal combinations of MemAttr, SnpAttr, and Order field values and the equivalent ARM memory type. The Order field is described in *Ordering* on page 2-116.

Table 2-12 Legal combinations of MemAttr, SnpAttr, and Order field values

MemAttr[3:0]						Order[1:0]		ARM Memory Type
[1]	[3]	[2]	[0]			[1]	[0]	
Device	Allocate	Cacheable	EWA	SnpAttr	LikelyShared			
1	0	0	0	0	0	1	1	Device nRnE
	0	0	1	0	0	1	1	Device nRE
	0	0	1	0	0	0/1[a]	0	Device RE
	All other values[b]							Not valid
0	0	0	0	0	0	0/1[a]	0	Non-cacheable Non-bufferable[c]
	0	0	1	0	0	0/1[a]	0	Non-cacheable Bufferable
	0	1	1	0	0	0/1[a]	0	Non-snoopable WriteBack No-allocate
	1	1	1	0	0	0/1[a]	0	Non-snoopable WriteBack Allocate
	0	1	1	1	0/1[d]	0/1[a]	0	Snoopable WriteBack No-allocate
	1	1	1	1	0/1[d]	0/1[a]	0	Snoopable WriteBack Allocate
	All other values[b]							Not valid

a. Order = `0b10` is permitted in ReadOnce*, WriteUnique, ReadNoSnp, WriteNoSnp and Atomic transactions only.

b. Order = `0b01` is not used for transactions' ordering. See *Order field encodings* on page 2-121.

c. Non-cacheable Non-bufferable is an AXI memory type, not an ARM memory type.

d. See *Likely Shared* on page 2-133.

Memory type

This section specifies the required behavior for each of the memory types that Table 2-12 on page 2-131 shows.

Device nRnE

The required behavior for Device nRnE memory type is:

- The write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- A read must not fetch more data than is required.
- A read must not be prefetched.
- Writes must not be merged.
- A write must not write to a larger address range than the original transaction.
- All Read and Write transactions from the same source to the same endpoint must remain ordered.

Device nRE

The required behavior for the Device nRE memory type is the same as for the Device nRnE memory type except that:

- The write response can be obtained from an intermediate point.

Device RE

The required behavior for the Device RE memory type is same as for the Device nRE memory type except that:

- Read and Write transactions from the same source to the same endpoint need not remain ordered.
- Read and Write transactions from the same source to addresses that overlap must remain ordered.

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.
- Writes can be merged.
- Read and Write transactions from the same source to addresses that overlap must remain ordered.

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.

  Note

  There is no mechanism to determine when a Write transaction is visible at its final destination.

- Read data must be obtained either from:
  - 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.
- Writes can be merged.
- Read and Write transactions from the same source to addresses that overlap must remain ordered.

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 is indistinguishable from the Read and Write transactions propagating to arrive at the final destination at the same time. Read data returned in this manner does not indicate that the Write transaction is visible at the final destination.

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.
- Reads can be prefetched.
- Writes can be merged.
- A cache lookup is required for Read and Write transactions.
- Read and Write transactions from the same source to addresses that overlap must remain ordered.
- The No-allocate attribute is an allocation hint, that is, it is a recommendation to the memory system that, for performance reasons, the transaction is not allocated. However, the allocation of the transaction is not prohibited.

Write-back Allocate

The required behavior for the WriteBack Allocate memory type is the same as for WriteBack No-allocate memory. However, in this case, the allocation hint is a recommendation to the memory system that, for performance reasons, the transaction is allocated.

2.9.5 Likely Shared

The LikelyShared attribute is a cache allocation hint. When asserted this attribute indicates that the requested data is likely to be shared by other Request Nodes within the system. This acts as a hint to shared system level caches that the allocation of the cache line is recommended for performance reasons.

There is no required behavior associated with this transaction attribute.

The LikelyShared assertion requirements are:

- Can be asserted in:
  - ReadClean.
  - ReadNotSharedDirty.
  - ReadShared.
  - StashOnceUnique, StashOnceSepUnique.
  - StashOnceShared, StashOnceSepShared.
  - WriteUniquePtl.
  - WriteUniqueFull.
  - WriteUniqueZero.
  - WriteUniquePtlStash.
  - WriteUniqueFullStash.
  - WriteBackFull.
  - WriteCleanFull.
  - WriteEvictFull.
  - WriteEvictOrEvict.

- Must not be asserted in any other Read, Write or Combined Write transaction.
- Must not be asserted in any Dataless or Atomic transaction.
- Is not applicable in DVMOp or PCrdReturn transaction, and must be set to zero.
- Is not applicable in PrefetchTgt transaction and can take any value.

2.9.6 Snoop Attribute

The *Snoop Attribute* (SnpAttr) indicates if a transaction requires snooping.

Table 2-13 shows the SnpAttr field encodings.

Table 2-13 SnpAttr field encodings

SnpAttr	Snoop attribute
0	Non-snoopable
1	Snoopable

Table 2-14 shows the snoop attributes for the different transaction types.

Table 2-14 Snoop attributes for the different transaction types

Transaction	Non-snoopable	Snoopable
ReadNoSnp, ReadNoSnpSep	Y	-
ReadOnce*, ReadClean, ReadShared, ReadNotSharedDirty, ReadUnique, ReadPreferUnique, MakeReadUnique	-	Y
CleanUnique, MakeUnique, StashOnce	-	Y
CleanShared, CleanSharedPersist*, CleanInvalid, MakeInvalid	Y	Y
Evict	-	Y
WriteNoSnp	Y	-
WriteBack, WriteClean, WriteEvictFull	-	Y
WriteUnique	-	Y
Atomic transactions	Y	Y
DVMOp	n/a[a]	n/a[a]
PrefetchTgt	n/a[b]	n/a[b]

a. Not applicable, must be set to zero.
b. Not applicable, can take any value.

The SnpAttr field value in a CMO, in an Atomic, and in ReadNoSnp and ReadNoSnpSep from Home to Slave, must be set to zero, irrespective of the field value in the Request from the original Requester to Home.

Note

For transactions that can take more than one value of SnpAttr, the value is typically determined from page table attributes.

2.9.7 Mismatched Memory attributes

It is permitted for two different agents to access the same location using mismatched MemAttr or SnpAttr values, at the same point in time.

Memory accesses from the different agents made with mismatched snoopability or cacheability attributes are considered as software protocol errors. A software protocol error can cause loss of coherency and result in the corruption of data values. It is required that the system does not deadlock on a software protocol error, and that transactions always make forward progress.

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 software cache maintenance can be used to return memory locations to a defined state.

The use of mismatched memory attributes can result in an RN-F observing a Snoop transaction to the same address that it is performing a ReadNoSnp or WriteNoSnp transaction to. In this situation there is no defined relationship between the Snoop transaction and the transaction that the RN-F has issued.

2.10 Data transfer

Read transactions, Write transactions, Atomic transactions, and Snoop responses with data, include a data payload. This section defines the data alignment rules, and the data bytes that are accessed for different combinations of address, transaction size, and memory type.

In this section, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

2.10.1 Data size

The Size field in a packet is used, in combination with other fields, to determine the number of bytes transferred. Table 2-15 shows the Size field value encodings. Snoop transactions do not include a Size field. All snoop data transfers are 64-byte.

Table 2-15 Size field value encodings

Size[2:0]	Bytes
`0b000`	1
`0b001`	2
`0b010`	4
`0b011`	8
`0b100`	16
`0b101`	32
`0b110`	64
`0b111`	Reserved

2.10.2 Bytes access in memory

The MemAttr[1] bit field determines if the memory type is Device or Normal. See *Memory Attributes* on page 2-128. The bytes that are accessed are determined by the memory type as follows:

Normal memory

Transactions with a Normal memory type access the number of bytes defined by the Size field. Data access is from the `Aligned_Address`, that is, the transaction address rounded down to the nearest Size boundary, and ends at the byte before the next Size boundary.

This is calculated as:

`Start_Address` = Addr field value.

`Number_Bytes` = $2^{\wedge \text{Size}}$ field value.

`INT(x)` = Rounded down integer value of x.

`Aligned_Address = (INT(Start_Address / Number_Bytes)) x Number_Bytes`.

The bytes accessed are from `(Aligned_Address)` to `(Aligned_Address + Number_Bytes) - 1`.

Device

Transactions with a Device memory type access the number of bytes from the transaction address up to the byte before the next Size boundary.

The bytes accessed are from `(Start_Address)` to `(Aligned_Address + Number_Bytes) - 1`.

For Write transactions to Device locations, byte enables must only be asserted for the bytes that are accessed. See *Byte Enables* on page 2-137.

2.10.3 Byte Enables

Byte Enables, also referred to as BE, are used alongside Write transactions, and Snoop responses with Data.

In this section, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

For Write transactions, an asserted byte enable indicates that the associated data byte is valid and must be updated in memory or cache. A deasserted byte enable indicates that the associated data byte is not valid and must not be updated in memory or cache.

In WriteData and Snoop response Data a byte enable value of zero must set the associated data byte value to zero.

A Requester must deassert all BE values in CopyBackWrData_I and WriteDataCancel packets.

The following Write transactions must have all byte enables asserted during the data transfers, except when the write data is a CopyBackWrData_I packet:
- WriteNoSnpFull.
- WriteBackFull.
- WriteCleanFull.
- WriteEvictFull.
- WriteUniqueFull.
- WriteUniqueFullStash.

The following Write transactions are permitted to have any combination of byte enables asserted during the data transfers. This includes asserting all and asserting none:
- WriteBackPtl.
- WriteUniquePtl.
- WriteUniquePtlStash.

For the WriteNoSnpPtl transaction the following rules apply:

- For a transaction to Normal memory, any combination of byte enables can be asserted during the data transfers. This includes asserting all and asserting none.

- For a transaction to Device memory, byte enables must only be asserted for bytes at or above the address specified in the transaction. Any combination of byte enables can be asserted that meets this requirement. This includes asserting all and asserting none.

For all Write transactions, byte enables that are not within the data window, specified by Addr and Size, must be deasserted.

For Atomic transactions, byte enables that are not within the data window, as specified below by Addr and Size, must be deasserted:
- If Addr is aligned to Size, then the Data window is `[Addr:(Addr+Size-1)]`.
- If Addr is not aligned to Size, then the Data window is `[(Addr-Size/2):(Addr+Size/2-1)]`.
- For Atomic transactions all byte enables within the data window must be asserted.

For snoop responses with data that use the SnpRespData opcode, all byte enables must be asserted.

For snoop responses with data that use the SnpRespDataPtl opcode, any combination of byte enables can be asserted alongside the data transfers. This includes asserting all and asserting none.

2.10.4 Data packetization

For each transaction that involves data, the data bytes can be transferred in multiple packets.

The number of packets required is determined by:

- Number of bytes.
- Data bus width.

The number of bytes transferred in each packet is determined by:

- Data bus width.

This specification supports the following data bus widths:

- 128-bit.
- 256-bit.
- 512-bit.

The Data Identifier and Critical Chunk Identifier fields are used to identify data packets within a transaction.

A transaction size of up to 16-byte is always contained in a single packet. The DataID field value must be set to Addr[5:4] because the DataID field represents Addr[5:4] of the lowest addressed byte within the packet.

Table 2-16 shows the relationship between the DataID field and the bytes that are contained within the packet, for different data bus widths.

Table 2-16 DataID and the bytes within a packet for different data widths

DataID	Data Width		
	128-bit	**256-bit**	**512-bit**
`0b00`	Data[127:0]	Data[255:0]	Data[511:0]
`0b01`	Data[255:128]	Reserved	Reserved
`0b10`	Data[383:256]	Data[511:256]	Reserved
`0b11`	Data[511:384]	Reserved	Reserved

Within a data packet, all bytes are located at their natural byte positions. This is true even if fewer data bytes are transferred than the width of the data bus.

The number of data packets used for transactions to Device memory is independent of the address of the transaction. The number of data packets required is determined only by the Size field and the data bus width.

Note

For some transactions to Device memory, it can be determined from the address at the start of the transaction that some data packets will not contain valid data and are redundant. However, this specification requires that these data packets are transferred.

2.10.5 Size, Address and Data alignment in Atomic transactions

This section describes the data size and alignment requirements for Atomic transactions. It contains the following sub-sections:

- *Size*.
- *Address and Data alignment*.
- *Endianness* on page 2-140.

Size

The Size field of the packet specifies the total data size of the Atomic transaction.

For the AtomicCompare transaction, the data size is the sum of the Compare and Swap data values.

Table 2-17 shows the permitted data sizes, and the relationship between inbound and outbound valid data size for each Atomic transaction type. The size of the data value returned in response to an AtomicCompare transaction is half the number of bytes specified in the Size field in the associated Request packet.

Table 2-17 Atomic transaction outbound and inbound data sizes

Atomic transaction	Outbound	Inbound
AtomicStore	1, 2, 4 or 8 byte	-
AtomicLoad	1, 2, 4 or 8 byte	Same as outbound
AtomicSwap	1, 2, 4 or 8 byte	Same as outbound
AtomicCompare	2, 4, 8, 16 or 32 byte	Half size of outbound

Address and Data alignment

In the AtomicStore, AtomicLoad and AtomicSwap transactions:

- The byte address is aligned to the outbound data size.
- The position of data bytes in the Data packet matches the endianness of the operation, as specified in the Endian field of the request.
- The big-endian data is byte invariant.

The write data associated with an AtomicCompare transaction is provided as if it were for a transaction that is aligned to the outbound data size.

In the AtomicCompare transaction:

- The byte address must be aligned in the Data packet to the inbound data size, which is equivalent to half the outbound data size.

The two data values in an AtomicCompare transaction are placed in the data field in the following manner:

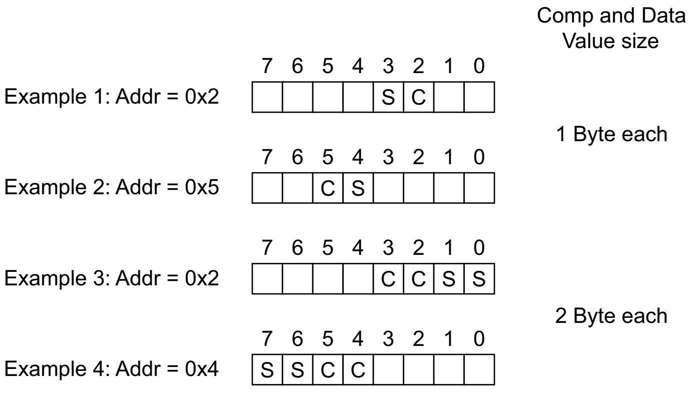
- The Compare and Swap data values are concatenated and the resulting data payload is aligned in the Data packet to the outbound data size.
- The Compare data is always at the addressed byte location.
- The Swap data is always in the remaining half of the valid data.

For any given Compare data address, the Swap data address can be determined by inverting bit[n] in the Compare data address where:

- $n = \log_2(\text{Compare data size in bytes})$

Alignment example

Figure 2-42 shows examples of data placement with different addresses and different Data size.



Only 8 bytes of the 16 byte data packet are shown

Figure 2-42 Data value packing for AtomicCompare transaction

In the first example that Figure 2-42 shows, the addressed byte location is `0x2` and the total size of data is two bytes. In this case, the Compare and Swap data must be placed in an address location aligned to a two byte boundary that includes the addressed location, that is, addresses `0x2` to `0x3`. Compare data is placed in location `0x2` and Swap data is placed in location `0x3`.

Note

The address of the Swap data can be determined by inverting bit[0] of the Compare data address. Bit[0] is inverted because the size of the Compare data and the size of the Swap data is one byte.

In the third example that Figure 2-42 shows, the addressed location is `0x2` and the total size of data is four bytes. In this case, the Compare and Swap data must be placed in an address location aligned to a four byte boundary that includes the addressed location, that is, addresses `0x0` to `0x3`. Compare data is placed in location `0x2` and Swap data is placed in location `0x0`.

Note

The address of the Swap data can be determined by inverting bit[1] of the Compare data address. Bit[1] is inverted because the size of the Compare data and the size of the Swap data is two bytes.

Endianness

The data on which an atomic operation executes can be in either little-endian or big-endian format. For arithmetic operations, such as ADD, MAX, and MIN the component performing the operation needs to know the format of the data.

The endian format of the data is defined by the Endian bit in the Atomic transaction Request packet. See *Endian* on page 13-418.

2.10.6 Critical Chunk Identifier

The CCID field is used to identify the data bytes that are the most critical in the transaction request.

The CCID field must match the value of Addr[5:4] of the original request. Transactions which contain multiple data packets must use the same CCID value for all data packets.

When read data or write data is reordered by the interconnect, the CCID field permits quick identification of the most critical bytes within a transaction by comparing the CCID value with the DataID value. When the two values match, the data bytes being transferred are the critical bytes.

The bits to match is dependent on the data bus width:

- For a data bus width of 128 bits, the CCID and DataID bits must match for the critical chunk.
- For a data bus width of 256 bits only the most significant CCID and DataID bits must match for the critical chunk.

2.10.7 Critical chunk first wrap order

The Sender of Data is permitted, but is not required, to send individual Data packets of a transaction in critical chunk first wrap order.

The interface property, CCF_Wrap_Order defines the capabilities of a Sender, and the guarantees provided by the Receiver:

- CCF_Wrap_Order at the Sender:

True	The Sender signals that it is capable of sending Data packets in critical chunk first wrap order.
False	The Sender signals that it is not capable of sending Data packets in critical chunk first wrap order.

- CCF_Wrap_Order at the ICN:

True	ICN guarantees that it will maintain the Data packets of a transaction in the order they are received.
False	ICN signals that it does not guarantee it can maintain the Data packets of a transaction in the order they are received.

- CCF_Wrap_Order at the Receiver that is not an ICN:

True	The Receiver requires the Data packets to be received in critical chunk first wrap order.
False	The Receiver does not require the Data packets to be received in critical chunk first wrap order.

If some components in the system do not support sending Data packets in critical chunk first wrap order then the receiver of Data must not rely on Data being received in critical chunk first wrap order.

Note

At design time, the CCF_Wrap_Order parameter can help a component to identify if Data packets need to be sent in critical chunk first wrap order. For example, if the component knows that it is connected to an out-of-order interconnect, then it might be able to simplify its Data packet path by not returning the Data packets in critical chunk first wrap order.

If the interconnect has the CCF_Wrap_Order property set to True, then a component interfacing to that interconnect, if capable, can send Data packets in critical chunk first wrap order, and the receiver can make use of possible latency optimization due to receiving the critical chunk first.

2.10.8 Data Beat ordering

This specification permits reordering of data packets within a transaction when passing across an interconnect. However, the original source of data packets is permitted, but not required, to provide the packets in a critical chunk first, wrap order. See *Critical chunk first wrap order* on page 2-141.

Note

Critical chunk first wrap order ensures that interfacing to protocols that do not support data reordering, such as AXI, can be done in the most efficient manner when an ordered interconnect is used.

Wrap order is defined as follows:

`Start_Address` = Addr

`Number_Bytes` = 2^Size

`INT(x)` = Rounded down integer value of x

`Aligned_Address` = `(INT(Start_Address / Number_Bytes)) x Number_Bytes`

`Lower_Wrap_Boundary` = `Aligned_Address`

`Upper_Wrap_Boundary` = `Aligned_Address` + `Number_Bytes` - 1

To maintain wrap order, the order must be:

1. The first data packet must correspond to the data bytes specified by the `Start_Address` of the transaction.
2. Subsequent packets must correspond to incrementing byte addresses up to the `Upper_Wrap_Boundary`.
3. Subsequent packets must correspond to the `Lower_Wrap_Boundary.`
4. Subsequent packets must correspond to incrementing byte addresses up to the `Start_Address`.

Note

Some of the steps to maintain wrap order might overlap and not be required if the required bytes are included in a previous step.

2.10.9 Data transfer examples

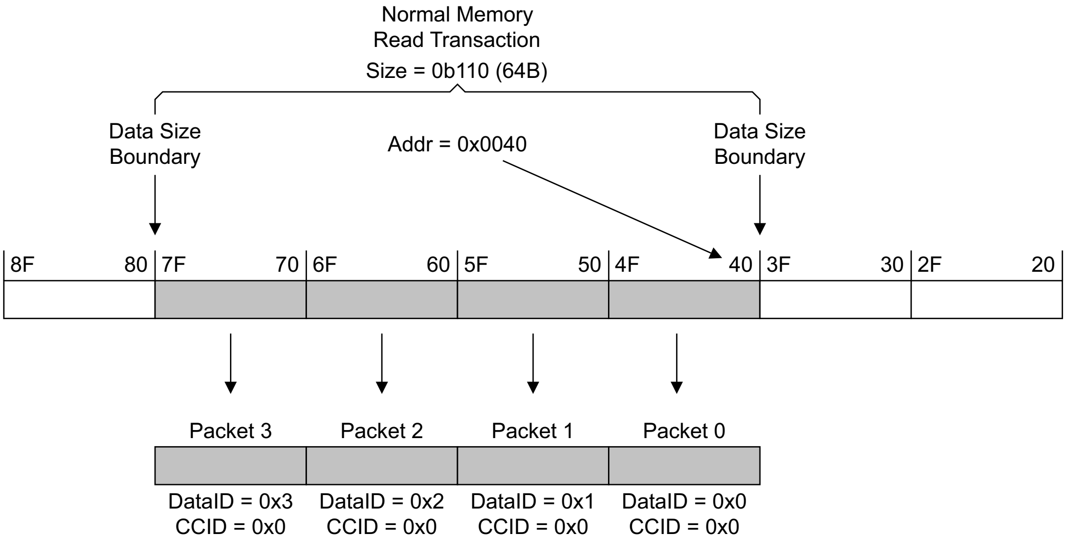
This section gives a number of examples of the data transfer requirements defined in this specification.

In most of the examples, the size of the transaction is 64-byte and the data bus width is 128-bit. This requires 4 data packets for each transaction.

In the following examples, the accompanying text highlights some interesting aspects. It is not intended to describe all aspects of the example.

Example 2-1 Normal memory 64-byte Read transaction from an aligned address

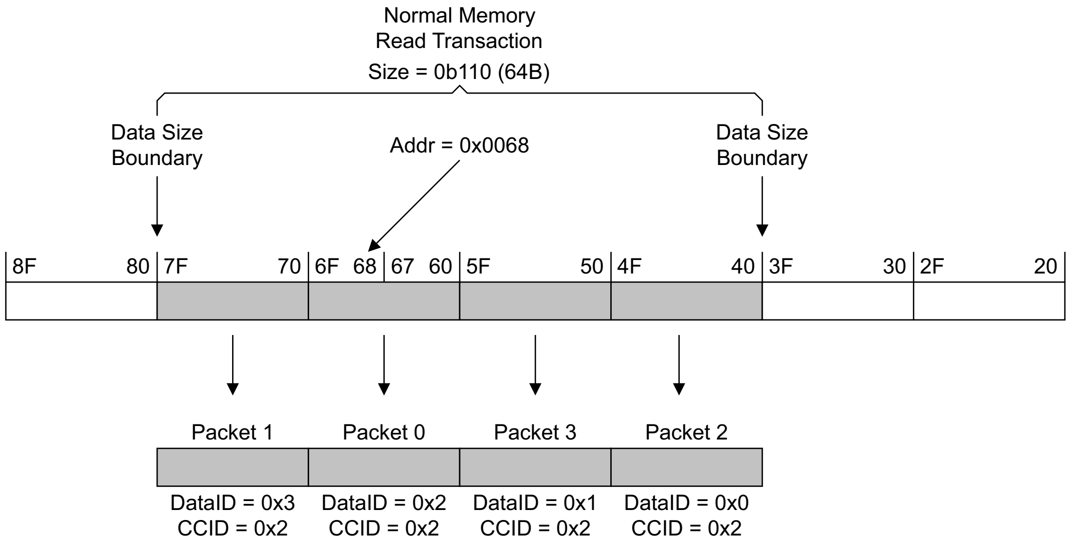
---



- The order of the data packets, as indicated by Packet 0, Packet 1, Packet 2, and Packet 3, is such that they follow wrap order.
- The DataID changes for each packet, while the CCID field remains constant.
- The packet containing the data bytes specified by the address of the transaction has the same value for the CCID and DataID fields.

Example 2-2 Normal memory 64-byte Read transaction from an unaligned address

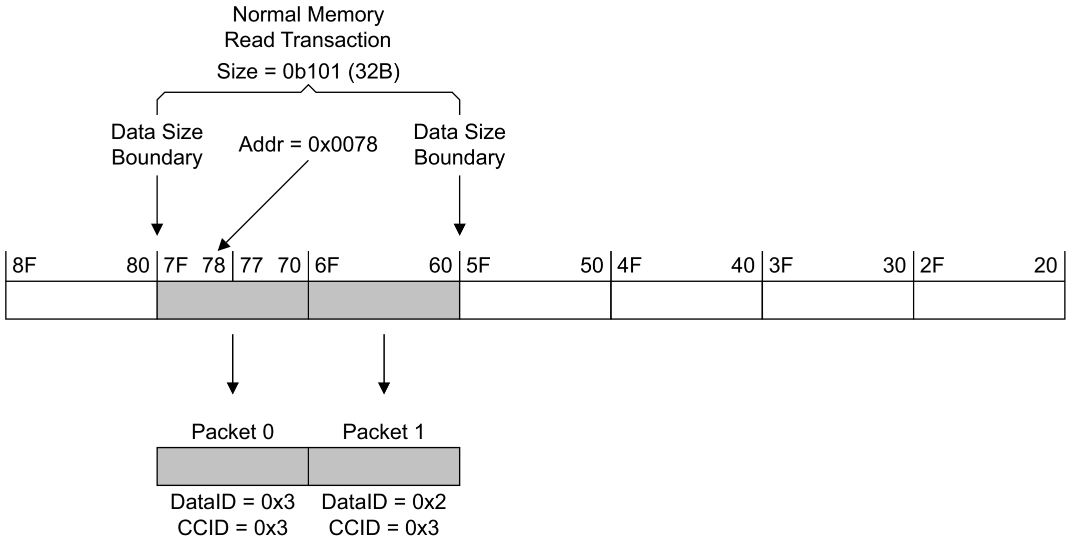
---



- The order of the data packets, as indicated by Packet 0, Packet 1, Packet 2, and Packet 3, is such that they follow wrap order.
- The DataID changes for each packet, while the CCID field remains constant.
- The packet containing the data bytes specified by the address of the transaction has the same value for the CCID and DataID fields.

Example 2-3 Normal memory 32-byte Read transaction from an unaligned address

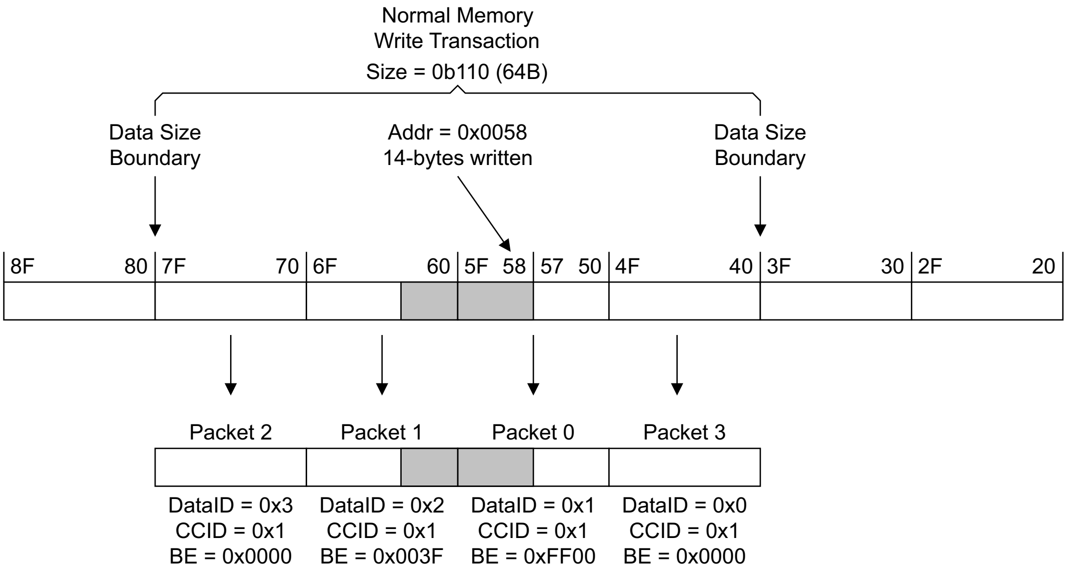
---



- The size of the transaction is 32-byte and the data bus width is 128-bit, resulting in 2 data packets.
- The order of the data packets, as indicated by Packet 0 and Packet 1, is such that they follow wrap order.

Example 2-4 Normal memory 14-byte consecutive write transaction from an unaligned address

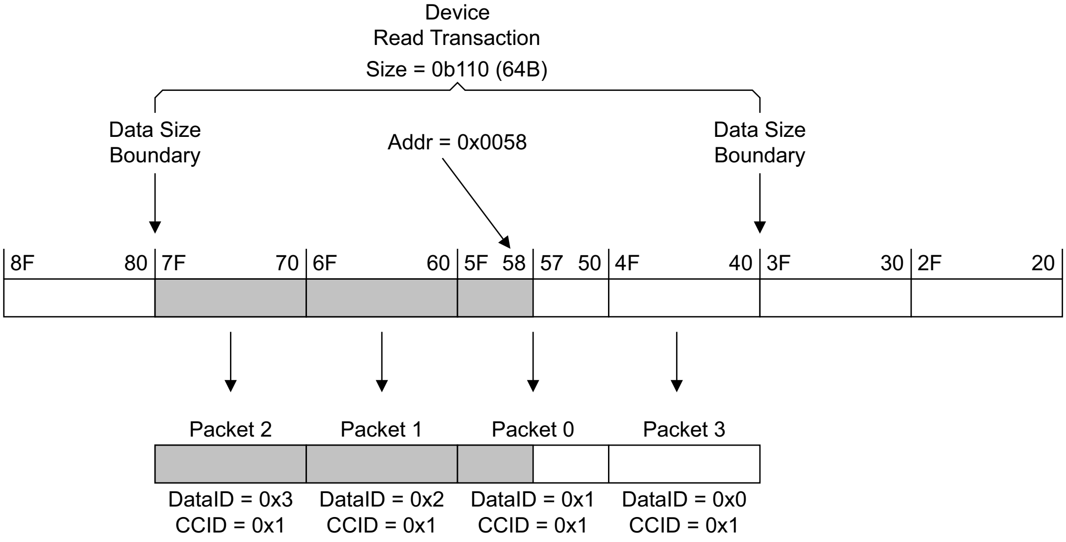
---



- The order of the data packets, as indicated by Packet 0, Packet 1, Packet 2, and Packet 3, is such that they follow wrap order.
- The DataID changes for each packet, while the CCID field remains constant.
- The packet containing the data bytes specified by the address of the transaction has the same value for the CCID and DataID fields.
- Fourteen consecutive bytes in memory are written, as indicated by the byte enables. However, other combinations of byte enables are permitted. See *Byte Enables* on page 2-137.

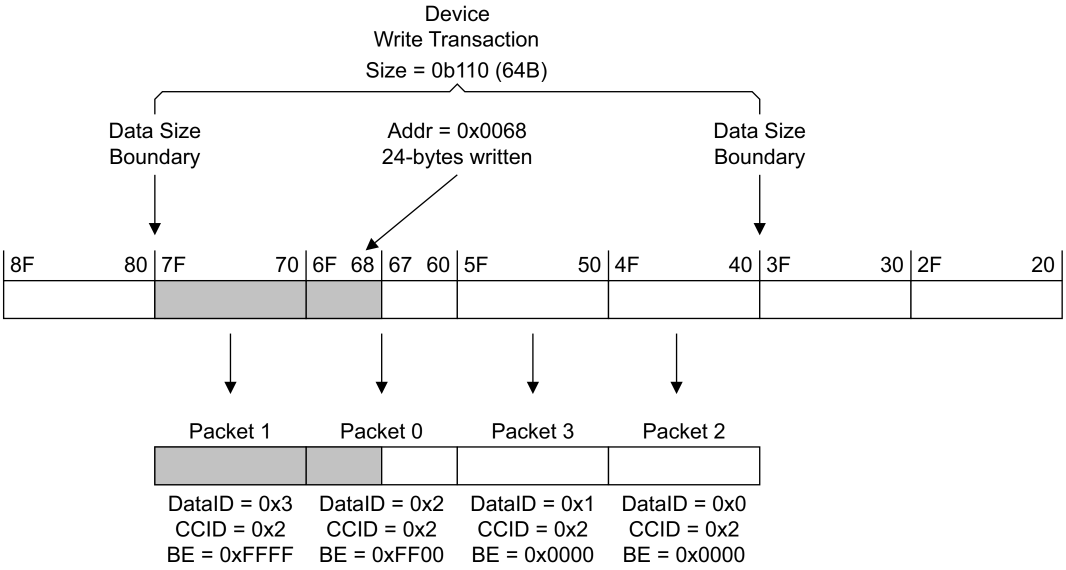
Example 2-5 Device Read transaction from an unaligned address

---



- The shaded area indicates the valid bytes in the transaction. The valid bytes extend from the transaction address up to the next Size boundary.
- The transaction includes the transfer of a packet that contains no valid data.

Example 2-6 Device write transaction to an unaligned address



- Byte enables are only permitted to be asserted for the bytes from the transaction address up to the next Size boundary. It is not required that all byte enables meeting this criteria are asserted.
- Byte enables for bytes below the start address must not be asserted.

2.11 Request Retry

This specification provides a request retry mechanism that ensures when a request reaches a Completer it is either accepted, or is given a RetryAck response, to prevent blocking of the REQ channel.

Request Retry is not applicable to the PrefetchTgt transaction. The PrefetchTgt transaction cannot be retried because there is no response associated with this request.

A Requester is required to hold all the details of the request until it receives a response indicating that the request has either been accepted, or must be sent again at a later point in time. To meet this requirement, with the exception of PrefetchTgt, the AllowRetry field must be asserted the first time a transaction is sent.

A Completer that is receiving requests is able to give a RetryAck response to a request that it is not able to accept. Typically, it will not be able to accept a request when it has limited resources and insufficient storage to hold the current request until some earlier transactions have completed.

When a Completer gives a RetryAck response it is responsible for recording where the request came from, as determined by the SrcID of the request. The Completer is also responsible for determining and recording the type of Protocol Credit required to process the request. The PCrdType field in the RetryAck encodes the type of Protocol Credit that will be granted by the Completer. When required resources become available, at a later point in time, the Completer must then send a P-Credit to the Requester, using a PCrdGrant response. The PCrdGrant response indicates to the Requester that the transaction can be retried.

Note

There is no explicit mechanism to request a credit. A transaction that is given a RetryAck response implicitly requests a credit.

It is possible that a reordering interconnect can reorder the responses such that the PCrdGrant is received by the Requester before the RetryAck response for the transaction is received. In this case, the Requester must record the credit it has received, including the credit type, so that it can assign the credit appropriately when it does receive the RetryAck response.

Note

It is expected to be rare that a PCrdGrant would be re-ordered with respect to a RetryAck response, as the delay between a RetryAck and a PCrdGrant response will typically be much longer than any delay caused by interconnect re-ordering.

When the Requester receives a credit, it can then resend the request with an indication that it has been allocated a credit. This is done by deasserting the AllowRetry field. This second attempt to carry out the transaction is guaranteed to be accepted.

The transaction that is resent must have the same field values as the original request, except when the field is inapplicable or is one of the following:

- TgtID. See *Target ID determination for Request messages* on page 3-154.
- QoS.
- TxnID.
- ReturnTxnID for ReadNoSnp and ReadNoSnpSep from HN to SN.
- RSVDC.
- AllowRetry, which must be deasserted.
- PCrdType, which must be set to the value in the Retry response for the original transaction.
- TraceTag.

When the Requester receives a RetryAck response for a CopyBack Write request whose data is invalidated by a snoop, the Requester:

- Can drop the Write request and return the received credit.
- Is permitted to send the Write request with AllowRetry set to zero, knowing that the subsequent data response would be CopyBackWrData_I.

There is no fixed relationship between credits and particular transactions. If a Requester has received multiple RetryAck responses for different transactions and it then receives a credit, there is no fixed credit allocation, the Requester is free to choose the most appropriate transaction from the list of transactions that received a RetryAck response with that particular Protocol Credit Type.

The retry mechanism supports up to sixteen different credit types. This lets the Completer use different credit types for different resources. For example, a Completer might use one credit type for the resources associated with Read transactions, and another credit type for Write transactions. Using different credit types gives the Completer the ability to efficiently manage its resources by controlling which of the retried requests can be sent again.

The transaction must only be retried by the Requester when a PCrdGrant is received with the correct PCrdType.

Note

If a Completer is only using one credit type, this specification recommends that the PCrdType value of `0b0000` is used. See *PCrdType* on page 2-149.

A Completer that is giving RetryAck responses must be able to record all the RetryAck responses that it has given to ensure it can correctly distribute credits. If the Completer is using more than one credit type the RetryAck responses that have been given for each credit type must be recorded.

A Requester must limit the transactions it issues so that the Completer is never required to track more than 1024 transactions that require a PCrdGrant response. This is achieved by limiting the maximum number of outstanding transactions to 1024 for each Requester.

A transaction is outstanding from the cycle that the request is first issued until either:

- The transaction is fully completed, as determined by the return of all the following responses that are expected for the transaction:
  - ReadReceipt, CompData, RespSepData, DataSepResp, DBIDResp*, Comp, and CompDBIDResp.
- It receives RetryAck and PCrdGrant and is either:
  - Retried using a credit of the appropriate PCrdType and then is fully completed as determined by the return of all responses.
  - Cancelled and returns the received credit using the PCrdReturn message.

A Requester can reuse the TxnID value used by a request either:

- As soon as it receives a RetryAck response for that request.
- As soon as it receives all the required responses for that request if the received responses are non-RetryAck responses.

Each transaction request includes a QoS value which can be used by the Completer to influence the allocation of credits as resources become available. See Chapter 10 *Quality of Service* for further details.

2.11.1 Credit Return

It is possible for a Requester to be given more credits than it requires.

This specification does not define when this can occur, but two typical scenarios are:

- A transaction is canceled between the first attempt and the point at which it can be resent with P-Credit.
- A transaction is requested multiple times with increasing QoS values. However, only a single completion of the transaction is required.

Note

If a Requester makes a second request before the first request has been given a RetryAck response then it must be acceptable for both transactions to occur. However, as an example, this behavior would typically not be acceptable for accesses to a peripheral device.

A Requester returns a credit by the use of the PCrdReturn transaction. This is effectively a *No Operation* (NOP) transaction that uses the credit that is not required. This transaction is used to inform the Completer that the allocated resources are no longer required for the given PCrdType.

Any credits that are not required must be returned in a timely manner.

Note

Any unused pre-allocated credit must be returned to avoid components holding on to credits in expectation of using them later. Such behavior is likely to result in an inefficient use of resources and to make analysis of the system performance difficult.

2.11.2 Transaction Retry mechanism

The following sections describe the Request transaction fields used by the Retry mechanism.

The transaction retry mechanism is not applicable to the PrefetchTgt transaction.

AllowRetry

The AllowRetry field indicates if the Request transaction can be given a RetryAck response. See Table 13-28 on page 13-418 for the AllowRetry value encodings. The AllowRetry field must be asserted the first time a transaction is sent.

The AllowRetry field must be deasserted when either:

- The transaction is using a pre-allocated P-Credit.
- The transaction is PrefetchTgt.

PCrdType

The PCrdType field indicates the credit type associated with the request and is determined as follows:

- For a Request transaction:
  - If the AllowRetry field is asserted, the PCrdType field must be set to `0b0000`.
  - If the AllowRetry field is deasserted, the PCrdType field must be set to the value that was returned in the RetryAck response from the Completer when the transaction was first attempted.
- A PCrdReturn transaction must have the credit type set to the value of the credit type that is being returned. See *PCrdType* on page 13-420 for the PCrdType value encodings.
- For destinations that have a single credit class, or do not implement credit type classification, this specification recommends that the PCrdType field is set to `0b0000`.

Note

The value a Completer assigns to PCrdType is IMPLEMENTATION DEFINED.

The Completer must implement a starvation prevention mechanism to ensure that all transactions, irrespective of QoS value or credit type required, will eventually make forward progress, even if over a significantly long time period. This is done by ensuring that credits are eventually given to every transaction that has received a RetryAck response. See Chapter 10 *Quality of Service* for more details on the distribution of credits for the purposes of QoS.

2.11.3 Transaction Retry flow

Figure 2-43 shows a typical Transaction Retry flow.

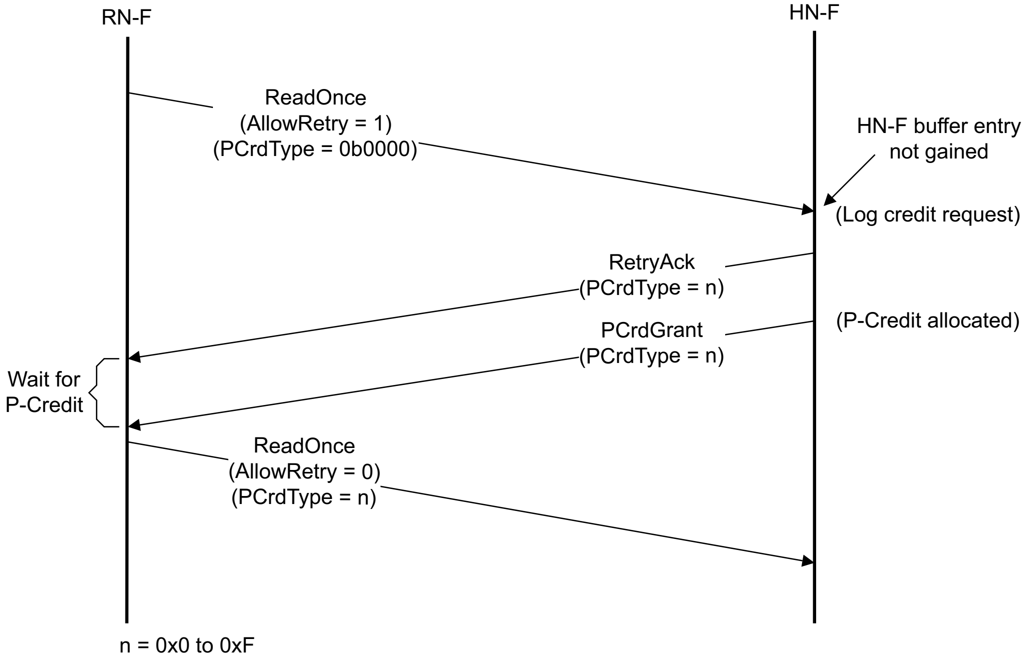


Figure 2-43 Transaction Retry flow

The steps that Figure 2-43 shows are:

1. The Requester sends a ReadOnce request.
    - This is done without a credit, so AllowRetry is asserted.
2. The Completer receives the request and sends a RetryAck response because it is not able to process the transaction at this time.
    - The request is logged and a PCrdType is determined at the Completer.
3. The Completer sends a P-Credit, using the PCrdGrant response, when it has allocated resource for the transaction.
    - The PCrdGrant includes the PCrdType allocated for the original request.
4. The Requester re-sends the transaction with AllowRetry deasserted.
    - The request uses the P-Credit and sets the PCrdType field to the value allocated for the original request.

It is permitted, but not expected, for a Completer to send a PCrdGrant before it has sent the associated RetryAck response.

Note

The Requester might receive PCrdGrant before RetryAck.

The second attempt at a transaction must not be sent until both a RetryAck response and an appropriate P-Credit is received for the transaction.

Chapter 3
Network Layer

This chapter describes the network layer that is responsible for determining the node ID of a destination node. It contains the following sections:

3.1 System address map

Each Requester, that is, each RN and HN in the system, must have a *System Address Map* (SAM) to determine the target ID of a request. The scope of the SAM might be as simple as providing a fixed node ID value to all the outgoing requests.

The exact format and structure of the SAM is IMPLEMENTATION DEFINED and is outside the scope of this specification.

The SAM must provide a complete decode of the entire address space. This specification recommends that any address that does not correspond to a physical component is sent to an agent that can provide an appropriate error response.

3.2 Node ID

Each component connected to a Port on the interconnect is assigned a node ID that is used to identify the source and destination of packets communicated over the interconnect. A Port can be assigned multiple node IDs. A node ID value can be assigned only to a single Port.

This specification supports a variable NodeID field width of 7 to 11 bits.

The width can be configured to any fixed value within this range for a given implementation and this value must be consistent across all NodeID fields.

Defining and assigning a node ID for each node in a system is IMPLEMENTATION DEFINED and is outside the scope of this specification.

3.3 Target ID determination

This section describes how the target ID is determined for the different message types. It contains the following sections:

- *Target ID determination for Request messages*.
- *Target ID determination for Response messages* on page 3-155.
- *Target ID determination for Snoop Request messages* on page 3-156.

3.3.1 Target ID determination for Request messages

For mapping of target ID in requests from the RN, this specification requires the *System Address Map* (SAM) logic to be present in the RN or in the interconnect. In the case of the interconnect, it might remap the target ID in the Request packet provided by the RN.

The target ID of a Request message is determined in the following manner using the system address map logic.

Except for PCrdReturn:

- If the request does not use a pre-allocated credit, then the target ID is determined by:
  - Opcode for DVMOp.
  - Address to node ID mapping for all other requests.

    PrefetchTgt uses a different Address to Node ID mapper than other Requests. Two Requests from an RN to the same Address, where one is a PrefetchTgt, target different nodes. PrefetchTgt always targets an SN. All other Requests from an RN that use an Address to Node ID mapper target an HN.
- If the request uses pre-allocated credit, the target ID of the request must be obtained from either the source ID of the RetryAck, provided as a response to the original Request message, or the target ID of the original request.

For PCrdReturn:

- The target ID provided by the RN must match the source ID included in the prior PCrdGrant which provided the credit being returned.

An RN must expect the interconnect to remap the target ID of a request.

For transactions from an RN, with the exception of PrefetchTgt which is targeted to an SN-F, this specification expects a Snoopable transaction to be targeted to HN-F and a Non-snoopable transaction to target HN-I or HN-F. It is legal for a Snoopable transaction to be targeted at an HN-I. This might occur, for example, due to a software programming error. In this case, the HN-I is required to respond to the transaction in a protocol compliant manner, but coherency is not guaranteed.

An HN might also use address map logic to determine the target Slave Node ID for each Request.

3.3.2 Target ID determination for Response messages

Response packets are issued as a result of a received message. The target ID in Response packets must match either the SrcID or the HomeNID, or the ReturnNID or the FwdNID in the received message that resulted in the response being sent. Table 3-1 shows the source of the Response packet target ID for each Response message type and the field in the received message that determines the target ID.

Table 3-1 Source of response packet target ID

Response Message	Target ID obtained from		
	At the HN	**At the SN**	**At the RN**
RetryAck	Request.SrcID	Request.SrcID	-
PCrdGrant	Request.SrcID	Request.SrcID	-
ReadReceipt	Request.SrcID	Request.SrcID	-
RespSepData	Request.SrcID	-	-
Comp	Request.SrcID	Request.SrcID	-
CompCMO	Request.SrcID	Request.SrcID	-
DataSepResp	Request.SrcID	Request.ReturnNID	-
CompData	Request.SrcID or SnpResp∗.SrcID[a]	Request.ReturnNID	Snoop.FwdNID
CompAck (reads)	-	-	Comp.SrcID or RespSepData.SrcID or CompData.HomeNID
CompAck (writes)	-	-	DBIDResp∗.SrcID or CompDBIDResp.SrcID
DBIDResp	Request.SrcID	Request.SrcID[b] or Request.ReturnNID[c]	-
DBIDRespOrd	Request.SrcID	-	-
WriteData	-	-	DBIDResp∗.SrcID or CompDBIDResp.SrcID
NCBWrDataCompAck	-	-	Comp.SrcID or DBIDResp∗.SrcID or CompDBIDResp.SrcID
Persist	Request.SrcID	Request.ReturnNID	-
CompPersist	Request.SrcID	Request.SrcID or Request.ReturnNID	-
StashDone	Request.SrcID	-	-
CompStashDone	Request.SrcID	-	-
TagMatch	Request.SrcID	Request.ReturnNID	-
SnpResp∗[d]	-	-	Snoop.SrcID

[a] For Data Pull requests where Snoop response can be SnpResp or SnpRespData or SnpRespDataPtl.

[b] If request.DoDWT is set to 0.

[c] If request.DoDWT us set to 1.

[d] SnpResp, SnpRespData, SnpRespDataPtl, SnpRespFwded, and SnpRespDataFwded.

3.3.3 Target ID determination for Snoop Request messages

A Snoop Request does not include a target ID. The protocol does not define an architectural mechanism to address and send a Snoop Request to a target. It is expected that this mechanism will be IMPLEMENTATION DEFINED and is outside the scope of this specification.

3.4 Network layer flow examples

This section shows transaction flows at the network layer. It contains the following sections:

- *Simple flow*.
- *Flow with interconnect based SAM* on page 3-158.
- *Flow with interconnect based SAM and Retry request* on page 3-158.

3.4.1 Simple flow

Figure 3-1 is an example of a simple transaction flow and shows how the TgtID is determined for the requests and responses:

1. RN0 sends a request with Target ID of HN0 using the SAM internal to RN0.
    - The interconnect does not remap the node ID.
2. HN0 looks up an internal SAM to determine the target SN.
3. SN0 receives the request and sends a data response.
    - The data response packet has the TgtID derived from the requests ReturnNID.
4. RN0 receives the data response from SN0.
5. RN0 sends, if required, a CompAck with TgtID of HN0 derived from the HomeNID in the Data Response packet to complete the transaction.

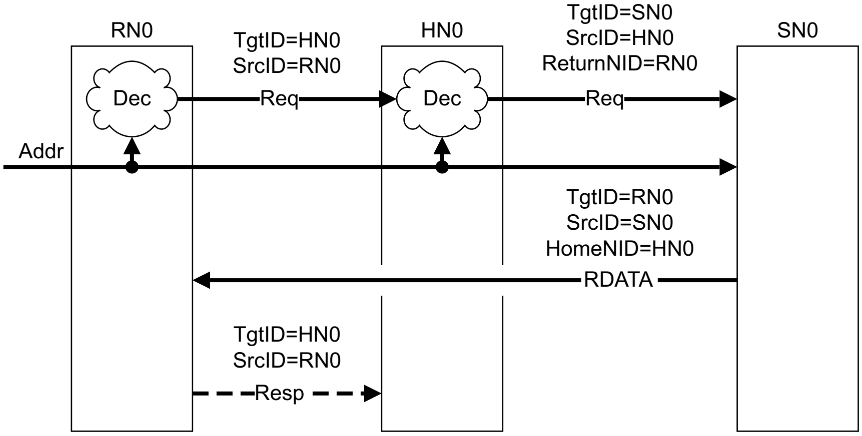


Figure 3-1 Target ID assignment without remapping

3.4.2 Flow with interconnect based SAM

Figure 3-2 shows a case where remapping of the target ID occurs in the interconnect.

Note

Only the target ID of the request from the RN is remapped. The TgtID in all other packets in the transaction flow is determined in a similar manner to *Simple flow* on page 3-157.

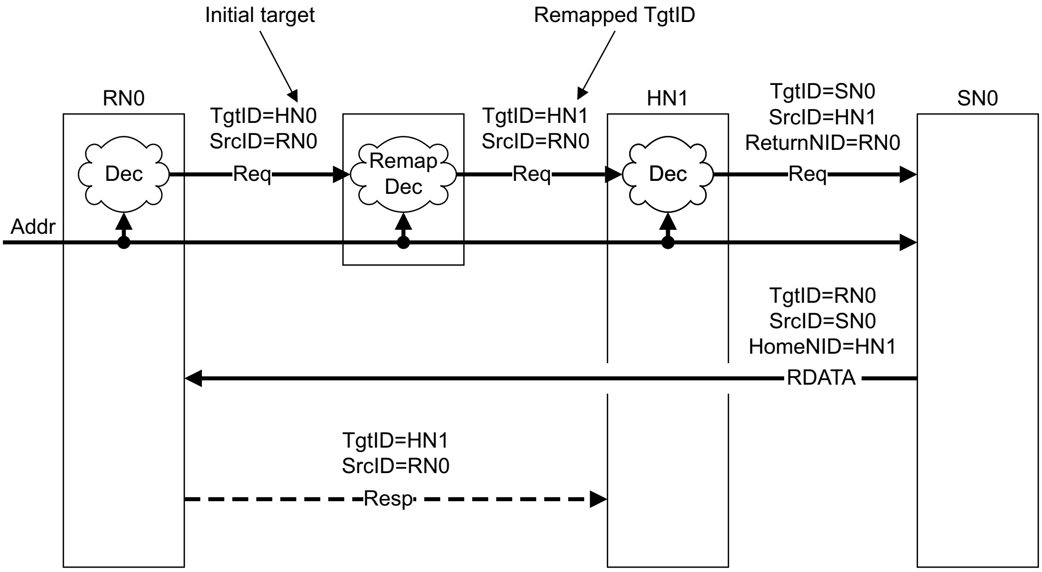


Figure 3-2 Target ID assignment with remapping logic

3.4.3 Flow with interconnect based SAM and Retry request

Figure 3-3 on page 3-159 shows a case of a request getting retried.

1. The interconnect remaps the TgtID provided by RN0 to HN1.
2. The request receives a RetryAck response.
   - The RetryAck and PCrdGrant responses get the TgtID information from the SrcID in the received request.
3. RN0 resends the request once both RetryAck and PCrdGrant responses are received.
   - The TgtID in the retried request is the same as the SrcID in the received RetryAck or the TgtID in the original request. The TgtID must pass through the remapping logic again.
4. The packets in the rest of the transaction flow get the TgtID in a similar manner to *Flow with interconnect based SAM*.

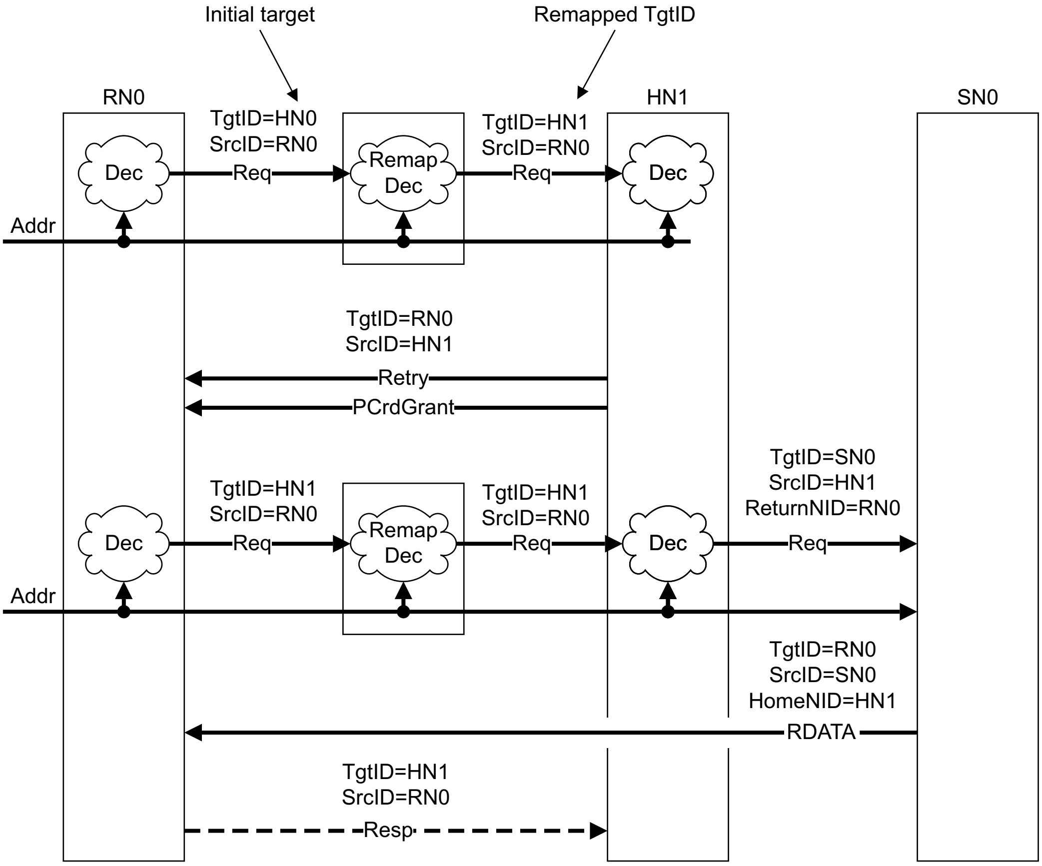


Figure 3-3 Remapping of TgtID and retried request

Chapter 4
Coherence Protocol

This chapter describes the coherence protocol and contains the following sections:

4.1 Cache line states

The action required when a protocol node accesses a cache line is determined by the cache line state. The protocol defines the following cache line states:

I Invalid:

- The cache line is not present in the cache.

UC Unique Clean:

- The cache line is present only in this cache.
- The cache line has not been modified with respect to memory.
- The cache line can be modified without notifying other caches.
- In response to a snoop that requests data, the cache line is permitted, but not required to be:
  - Returned to Home when requested.
  - Forwarded directly to the Requester when instructed by the snoop.

UCE Unique Clean Empty:

- The cache line is present only in this cache.
- The cache line is in a unique state but none of the data bytes are valid.
- The cache line can be modified without notifying other caches.
- In response to a snoop that requests data, the cache line must not be:
  - Returned to Home even when requested.
  - Forwarded directly to the Requester even when instructed by the snoop.

UD Unique Dirty:

- The cache line is present only in this cache.
- The cache line has been modified with respect to memory.
- The cache line must be written back to next level cache or memory on eviction.
- The cache line can be modified without notifying other caches.
- In response to a snoop that requests data, the cache line:
  - Must be returned to Home when requested.
  - Is expected to be forwarded directly to the Requester when instructed by the snoop.

UDP Unique Dirty Partial:

- The cache line is present only in this cache.
- The cache line is unique. Only a part of the cache line is Valid and Dirty.
- The cache line has been modified with respect to memory.
- When the cache line is evicted, it must be merged with data from next level cache or memory to form the complete Valid cache line.
- The cache line can be modified without notifying other caches.
- In response to a snoop that requests data, the cache line must:
  - Be returned to Home.
  - Not be forwarded directly to the Requester even when instructed by the snoop.

SC Shared Clean:

- Other caches might have a shared copy of the cache line.
- The cache line might have been modified with respect to memory.
- It is not the responsibility of this cache to write the cache line back to memory on eviction.
- The cache line cannot be modified without invalidating any shared copies and obtaining unique ownership of the cache line.
- In response to a snoop that requests data, the cache line:
    - Is required to not return data if RetToSrc bit is not set.
    - Is expected to return data if RetToSrc bit is set.
    - Is expected to be forwarded directly to the Requester when instructed by the snoop.

SD Shared Dirty:

- Other caches might have a shared copy of the cache line.
- The cache line has been modified with respect to memory.
- The cache line must be written back to next level cache or memory on eviction.
- The cache line cannot be modified without invalidating any shared copies and obtaining unique ownership of the cache line.
- In response to a snoop that requests data, the cache line:
    - Must be returned to Home when requested.
    - Is expected to be forwarded directly to the Requester when instructed by the snoop.

A cache is permitted to implement a subset of these states.

4.1.1 Empty cache line ownership

An empty cache line is a cache line that is held in a Unique state, so no other copies of the cache line exist, but none of the data bytes are Valid. This cache line state is UCE.

The following are examples of when empty cache line ownership can occur:

- A Requester can deliberately obtain an empty cache line:
    - Before starting a write, to save system bandwidth, a Requester that expects to write to a cache line can obtain an empty cache line with permission to store, instead of obtaining a Valid copy of the cache line.
- A Requester can transition into an empty state:
    - If the Requester has a copy of the cache line when it requests permission to store, and that copy of the cache line is invalidated before the Requester obtains permission to store, this results in the Requester having an empty cache line with permission to store.

4.1.2 Ownership of cache line with partial Dirty data

Once ownership of a cache line without data is obtained, the Requester is permitted to store to the cache line. If the Requester modifies part of the cache line, the cache line remains partially Unique Dirty. This cache line state is UDP.

4.2 Request types

Protocol requests are categorized as follows:

- Read requests:
    - A data response is provided to the Requester.
    - Can result in data movement among other agents in the system.
    - Can result in a cache state change at the Requester.
    - Can result in a cache state change at other Requesters in the system.
- Dataless requests:
    - No data response is provided to the Requester.
    - Can result in data movement among other agents in the system.
    - Can result in a cache state change at the Requester.
    - Can result in a cache state change at other Requesters in the system.
- Write requests and Combined Write requests:
    - Move data from the Requester.
    - Can result in data movement among other agents in the system.
    - Can result in a cache state change at the Requester.
    - Can result in a cache state change at other Requesters in the system.
- Atomic requests:
    - Move data from the Requester.
    - A data response is provided to the Requester in some Request types.
    - Can result in data movement among other agents in the system.
    - Can result in a cache state change at the Requester.
    - Can result in a cache state change at other Requesters in the system.
- Other requests:
    - Do not involve any data movement in the system.
    - Can be used to assist with *Distributed Virtual Memory* (DVM) maintenance.
    - Can be used to warm the memory controller for a following Read request.

The following subsections enumerate the resulting transactions and their characteristics.

Note

In *Read transactions* on page 4-165, *Dataless transactions* on page 4-170 and *Write transactions* on page 4-173, information is provided on the expected communicating node pairs. It is also legal for any transaction that is expected to target an HN-F, but not an HN-I, to target an HN-I. This can occur in the case of an incorrect assignment of memory type for a transaction. It is required that the HN-I responds to such a transaction in a protocol compliant manner. See Appendix B *Communicating Nodes* for complete information on communication node pairs.

4.2.1 Read transactions

See Chapter 12 *Memory Tagging* for a description of the Memory Tagging mechanism.

For information on the permitted MTE TagOp values for each Read request see Table 12-2 on page 12-378.

ReadNoSnp Read request by an RN to a Non-snoopable address region, or from HN to obtain a copy of the addressed data from the Slave:

- Data is included with the completion response.
- Data size is up to a cache line length, based on size attribute value in the request, irrespective of memory attributes.
- Data is not cached at the Requester in a system coherent manner.

  Note

  It is permitted to retain a copy of the data obtained in a local cache, or buffer, but this copy of the data will not remain coherent.

- Can have exclusive attribute asserted. See Chapter 6 *Exclusive Accesses* for details.
- Permitted, but not required, to assert ExpCompAck in the Request.
- Permitted to assert Order field in the Request.
- Data cannot be obtained directly from the Slave Node using DMT if the Exclusive bit is set.
- Permitted to use DMT if ExpCompAck is asserted in the Request.
- Permitted to use DMT if both ExpCompAck and Order are deasserted in the Request.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

ReadNoSnpSep A Read request to tell the Completer to send only a Data response:

- Data is returned in the Data-only response.
- Data size is up to a cache line length.
- Must not assert ExpCompAck in the Request.
- The Order field of the request must be set to `b01`.
- Communicating node pairs:
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I

ReadOnce Read request to a Snoopable address region to obtain a snapshot of the coherent data.

- Data is included with the completion response.
- Data size is a cache line length.
- Data will not be cached at the Requester.

  Note

  It is permitted to retain a copy of the data obtained in a local cache, or buffer, but this copy of the data will not remain coherent.

- Permitted, but not required, to assert ExpCompAck in the Request.
- Permitted to assert Order field in the Request.
- Permitted to use DMT if ExpCompAck is asserted in the Request.
- Permitted to use DMT if both ExpCompAck and Order are deasserted in the Request.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

ReadOnceCleanInvalid

Read request to a Snoopable address region to obtain a snapshot of the coherent data:

- Data is included with the completion response.
- Data size is a cache line length.
- Data will not be cached in a coherent state at the Requester.
- Permitted, but not required to assert ExpCompAck in the request.
- Permitted to assert Order field in the request.
- Permitted to use DMT if ExpCompAck is asserted in the Request.
- Permitted to use DMT if both ExpCompAck and Order field are deasserted in the Request.
- It is recommended, but not required that a snooped cached copy is invalidated.
- If a Dirty copy is invalidated, it must be written back to memory.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

——— Note ———

ReadOnceCleanInvalid is used instead of ReadOnce or ReadOnceMakeInvalid where the application determines that the data is still Valid, but will not be used in the near future.

Use of ReadOnceCleanInvalid by an application improves cache efficiency by reducing cache pollution.

The following should be considered when using ReadOnceCleanInvalid:

- The invalidation in the ReadOnceCleanInvalid transaction is a hint. Completion of the transaction does not guarantee removal of all cached copies, therefore it cannot be used as a replacement for a CMO.
- Use of the transaction can cause the deallocation of a cache line and therefore caution is needed if the transaction could target the same cache line that other agents in the system are using for Exclusive accesses.
- 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.

ReadOnceMakeInvalid

Read request to a Snoopable address region to obtain a snapshot of the coherent data:

- Data is included with the completion response.
- Data size is a cache line length.
- Data will not be cached in a coherent state at the Requester.
- Permitted, but not required, to assert ExpCompAck in the Request.
- Permitted to assert Order field in the Request.
- Permitted to use DMT if ExpCompAck is asserted in the Request.
- Permitted to use DMT if both ExpCompAck and Order field are deasserted in the Request.
- It is recommended, but not required, that all snooped cached copies are invalidated.
- If a Dirty copy is invalidated, it does not need to be written back to memory.
- Communicating node pairs:
    - RN-F, RN-D, RN-I to ICN(HN-F).

Note

ReadOnceMakeInvalid is used in preference to ReadOnce or ReadOnceCleanInvalid to obtain a snapshot of a data value when the application determines that the cached data is not going to be used again.

The application can free up the caches and also, by discarding Dirty data, avoid an unnecessary WriteBack to memory.

The following should be considered when using ReadOnceMakeInvalid:

- The invalidation in the ReadOnceMakeInvalid transaction is a hint. Completion of the transaction does not guarantee removal of all cached copies, therefore it cannot be used as a replacement for a CMO.
- Use of the transaction can cause the deallocation of a cache line and therefore caution is needed if the transactions could target the same cache line that other agents in the system are using for Exclusive accesses.
- The use of the ReadOnceMakeInvalid transaction can cause the loss of a Dirty cache line. Use of this transaction must be strictly limited to scenarios where it is known that the loss of a Dirty cache line is harmless.
- For a ReadOnceMakeInvalid transaction, it is required that the invalidation of the cache line is committed prior to the read data response for the transaction. The invalidation of the cache line is not required to have completed at this point, but it must be ensured that any later write transaction from any agent, which starts after this point, is guaranteed not to be invalidated by this transaction.

ReadClean

Read request to a Snoopable address region:

- Data is included with the completion response.
- Data size is a cache line length.
- Data must be provided to the Requester in clean state only:
    - UC, or SC.
- Can have exclusive attribute asserted. See Chapter 6 *Exclusive Accesses* for details.
    - Data cannot be obtained directly from the Slave Node using DMT if the Exclusive bit is set.
- Communicating node pairs:
    - RN-F to ICN(HN-F).

ReadNotSharedDirty

Read request to a Snoopable address region.

- Data is included with the completion response.
- Data size is a cache line length.
- Requester will accept the data in any valid state except SD:
  - UC, UD, SC.
- Can have exclusive attribute asserted. See Chapter 6 *Exclusive Accesses* for details.
  - Data cannot be obtained directly from the Slave Node using DMT if the Exclusive bit is set.
- Communicating node pairs:
  - RN-F to ICN(HN-F).
- Request is included in this specification for use by caches that do not support the SharedDirty state.

ReadShared

Read request to a Snoopable address region.

- Data is included with the completion response.
- Data size is a cache line length.
- Requester will accept the data in any valid state:
  - UC, UD, SC, or SD.
- Can have exclusive attribute asserted. See Chapter 6 *Exclusive Accesses* for details.
  - Data cannot be obtained directly from the Slave Node using DMT if the Exclusive bit is set.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

ReadUnique

Read request to a Snoopable address region to carry out a store to the cache line.

- All other cached copies must be invalidated.
- Data is included with the completion response.
- Data size is a cache line length.
- Data must be provided to the Requester in unique state only:
  - UC, or UD.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

ReadPreferUnique

Read request to a Snoopable address region requesting a unique copy of a cache line, Used when the Requester prefers, but does not require, the data to be returned in Unique state,

- Data is included with the completion response.
- Data size is a cache line length.
- Data is provided in Unique state unless another RN in the system is currently performing an exclusive sequence using the same address. In which case the data is provided in Shared state.
- It is permitted for the Home to always provide the data in Shared state.
- Responses can come from the Home, a Slave or a peer Request Node.
- Exclusive attribute in the request can take any value.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

Note

Request is included in this specification to improve the execution efficiency of an exclusive sequence.

MakeReadUnique Read request to a Snoopable address region requesting a unique copy of a cache line. Typical usage is when the Requester has a shared copy of the cache line and wants to obtain permission to store to the cache line.

Note

Because data return is guaranteed if the Requester receives an invalidating Snoop and retention of data is required otherwise, re-sending a Request to obtain a unique copy of the cache line is never required.

- LikelyShared must be deasserted in the request.
- Exclusive attribute in the request can take any value.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

4.2.2 Dataless transactions

See Chapter 12 *Memory Tagging* for a description of the Memory Tagging mechanism.

For information on the permitted MTE TagOp values for each Dataless request see Table 12-2 on page 12-378.

CleanUnique

Request to a Snoopable address region to change the state to Unique to carry out a store to the cache line. Typical usage is when the Requester has a shared copy of the cache line and wants to obtain permission to store to the cache line.

- Data is not included with the completion response.
- Any dirty copy of the cache line at a snooped cache must be written back to the next level cache or memory.
- Can have exclusive attribute asserted. See Chapter 6 *Exclusive Accesses* for details.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

MakeUnique

Request to Snoopable address region to obtain ownership of the cache line without a data response. This request is used only when the Requester guarantees that it will carry out a store to all bytes of the cache line.

- Data is not included with the completion response.
- Any dirty copy of the cache line at a snooped cache must be invalidated without carrying out a data transfer.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

Evict

Used to indicate that a Clean cache line is no longer cached by an RN.

- Data is not sent for this transaction.
- The cache line must not remain in the cache.
- Communicating node pairs:
  - RN-F to ICN(HN-F).

StashOnceUnique, StashOnceSepUnique

Request to a Snoopable address region. Request includes the Node ID of another RN and the Request can optionally include the ID of a Logical Processor within that node. It is recommended, but not required, that the other agent is snooped to indicate that it reads the addressed cache line and ensures that it is in a cache state suitable for writing to the cache line. The Data Pull request from the target RN is treated as a ReadUnique request. When a valid target is not specified, the addressed cache line can be fetched to be cached at the request Completer.

- Data is not included with the completion response.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

StashOnceShared, StashOnceSepShared

Request to a Snoopable address region. Request includes the Node ID of another RN and the Request can optionally include the ID of a Logical Processor within that node. It is recommended, but not required, that the other agent is snooped to indicate that it reads the addressed cache line. The Data Pull request from the target RN is treated as a ReadNotSharedDirty. When a valid target is not specified, then the addressed cache line can be fetched to be cached at the request Completer.

- Data is not included with the completion response.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

Cache Maintenance transactions

A *Cache Maintenance Operation* (CMO) assists with software cache management. The protocol includes the following five transactions to support a CMO:

CleanShared The completion response to a CleanShared request ensures that all cached copies are changed to a Non-dirty state and any Dirty copy is written back to memory.

CleanSharedPersist The completion response to a CleanSharedPersist request ensures that all cached copies are changed to a Non-dirty state and any Dirty cached copy is written back to the *Point of Persistence* (PoP) or final destination.

CleanSharedPersistSep

The Persist or combined CompPersist completion response to a CleanSharedPersistSep request ensures that all cached copies are changed to a Non-dirty state and any Dirty cached copy is written back to the *Point of Persistence* (PoP), or final destination.

Functionality of CleanSharedPersistSep is similar to CleanSharedPersist but requires two separate responses to the Requester:

- Comp response, indicating that the request has reached the Point of Coherency (PoC) and hazards can be removed at the Requester.
- Persist response, indicating that the request has reached the PoP, or the final destination.

It is expected, but not required, that a Requester, when sending a persistent CMO, uses a CleanSharedPersistSep transaction instead of CleanSharedPersist.

Such a Requester must support receiving both separate Comp and Persist responses as well as a combined CompPersist response.

CleanInvalid The completion response to a CleanInvalid request ensures that all cached copies are invalidated. The request requires that any cached Dirty copies must be written to memory.

MakeInvalid The completion response to a MakeInvalid request ensures that all cached copies are invalidated. The request permits that any cached Dirty copies are discarded.

The following characteristics are common to all five CMO transactions:

- Data is not included with the completion response. The Resp field value in the Comp, indicating cache state, must be ignored by both the Requester and the Home.
- Sending of a CMO transaction to the interconnect from an RN and from the interconnect to an SN is controlled by the **BROADCASTPERSIST** (**BP**) and **BROADCASTCACHEMAINTENANCE** (**BCM**) interface signals. See *Optional interface broadcast signals* on page 16-460.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

Note

Permitting CMOs to be forwarded downstream of the Home Node incorporates system topologies where some observers might directly access locations downstream of the Home Node and software cache maintenance is required to make cached data visible to such observers.

- ExpectCompAck must not be asserted.
- Cacheable and SnpAttr bit values must be observed.

- The permitted MemAttr values are:

Device	Allocate	Cacheable	EWA
1	0	0	0
	0	0	1
0	0	0	0
	0	0	1
	0	1	1
	1	1	1

  - — MemAttr values on a request received at Home must be preserved when the request is propagated to the Slave, except when it is known that the Slave has only Normal memory, in which case the MemAttr bit Device can be set to Normal.
- Snoopable attribute can take any value.
- Order field must not be asserted:
  - — A CMO intended for a particular address must not be sent to the interconnect before all previous transactions sent to the same address that can allocate the received data in the Requester caches have completed.
  - — A transaction, except Evict, WriteEvictFull, WriteEvictOrEvict, and PrefetchTgt, intended for a particular address, must not be sent to the interconnect before a previous CMO sent to the same address has completed.
  - — A Completer is permitted, but not required, to wait for WriteData to send a Persist response. Examples of when a Completer can send an early Persist is when the Completer does not support Persistence on that memory location or the request encounters a Non-data Error.
- A CMO is permitted to be combined with a Write transaction to the same address when the write is ahead of the CMO. See *Combined Write requests* on page 4-176.

Use of the Deep attribute in Persistent CMO

Systems with non-volatile memory to meet high availability expectations require guarantees that operation critical data is preserved on back-up battery failure as well as on power failure. The guarantee can be provided by a system by adding a mechanism to push previous writes to the Point of Deep Persistence. This specification supports this feature using an attribute, called Deep, on Persistent CMO transactions.

Deep attribute is applicable in the CleanSharedPersist and CleanSharedPersistSep transactions. See *Deep* on page 13-414.

4.2.3 Write transactions

Write transactions move data from a Requester to a Completer, this might be the next level cache, memory, or a peripheral. The data being transferred, depending on the transaction type, can be coherent or non-coherent. Each write transaction must assert appropriate byte enables with the data.

See Chapter 12 *Memory Tagging* for a description of the Memory Tagging mechanism.

For information on the permitted MTE TagOp values for each Write request see Table 12-2 on page 12-378.

WriteNoSnpFull Write a full cache line of data from an RN to a Non-snoopable address region, or a write for a full cache line of data from Home to Slave.

- Data size is a cache line length.
- All byte enables must be asserted.
- Can have the exclusive attribute asserted. See Chapter 6 *Exclusive Accesses*.
- Permitted to use DWT flow if the Order field is not set to OWO.
- Permitted to assert ExpCompAck only when Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

WriteNoSnpPtl Write a partial cache line of data from an RN to Non-snoopable address region or a write for a partial cache line of data from Home to Slave.

- Data size is up to a cache line length.
- Byte enables must be asserted for the appropriate byte lanes within the specified data size and deasserted in the rest of the data transfer.
- Can have the exclusive attribute asserted. See Chapter 6 *Exclusive Accesses*.
- Permitted to use DWT flow if the Order field is not set to OWO.
- Permitted to assert ExpCompAck only when Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

WriteNoSnpZero Write without data bytes when the data value is zero.

- Data size is a cache line length.
- Must not assert Exclusive in the request. See Chapter 6 *Exclusive Accesses*.
- Must not assert ExpCompAck in the request.
- The permitted Order field values in the request transaction are:
  - From RN to HN: No Order, Request Order and Endpoint Order only.
  - From HN-F to SN-F: No Order only.
  - From HN-I to SN-I: No Order and Endpoint Order only.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

WriteUniqueFull Write to a Snoopable address region. Write a full cache line of data to the next-level cache or memory when the cache line is Invalid at the Requester.

- Data size is a cache line length.
- All byte enables must be asserted.
- Permitted to assert ExpCompAck only when the Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

WriteUniquePtl Write to a Snoopable address region. Write up to a cache line of data to the next-level cache or memory when the cache line is Invalid at the Requester.

- Data size is up to a cache line length.
- Byte enables must be asserted for the appropriate byte lanes within the specified data size and deasserted in the rest of the data transfer.
- Permitted to assert ExpCompAck only when the Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

WriteUniqueZero Write without data bytes when the data value is zero.

- Size is a cache line length.
- Must not assert Exclusive in the request. See Chapter 6 *Exclusive Accesses*.
- Must not assert ExpCompAck in the request.
- The permitted Order field values in the request transaction are:
  - From RN to HN: No Order and Request Order only.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

WriteUniqueFullStash

Write to a Snoopable address region. Write a full cache line of data to the next-level cache or memory when the cache line is Invalid at the Requester. Also includes a request to the Stash target node to read the addressed cache line. The expected Read request is ReadUnique.

- Data size is a cache line length.
- All byte enables must be asserted.
- Permitted to assert ExpCompAck only when the Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

WriteUniquePtlStash

Write to a Snoopable address region. Write up to a cache line of data to the next-level cache or memory when the cache line is Invalid at the Requester. Also includes a request to the Stash target node to read the addressed cache line. The expected Read request type is ReadUnique.

- Data size is up to a cache line length.
- Byte enables must be asserted for the appropriate byte lanes within the specified data size and deasserted in the remainder of the data transfer.
- Permitted to assert ExpCompAck only when the Order field value is set to `0b10`.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F).

CopyBack transactions

CopyBack transactions are a subclass of Write transactions. CopyBack transactions move coherent data from a cache to the next level cache or memory. Each CopyBack transaction must assert the appropriate byte enables with the data. CopyBack transactions do not require the snooping of other agents in the system.

WriteBackFull Write-back a full cache line of Dirty data to the next level cache or memory.

- Data size is a cache line length.
- All byte enables must be asserted except when the write data is CopyBackWrData_I.
- The cache line must not remain in the cache.
- Communicating node pairs:
  - — RN-F to ICN(HN-F).

WriteBackPtl Write-back up to a cache line of Dirty data to the next level cache or memory.

- Data size is a cache line length.
- All appropriate byte enables, up to all 64, must be asserted.
- The cache line must not remain in the cache.
- Communicating node pairs:
  - — RN-F to ICN(HN-F).

WriteCleanFull Write-back a full cache line of Dirty data to the next level cache or memory and retain a Clean copy in the cache.

- Data size is a cache line length.
- All byte enables must be asserted except when the write data is CopyBackWrData_I.
- The cache line is expected to be in Clean state at completion of the transaction.
- Communicating node pairs:
  - — RN-F to ICN(HN-F).

WriteEvictFull Write-back of UniqueClean data to the next-level cache.

- Data size is a cache line length.
- All byte enables must be asserted except when the write data is CopyBackWrData_I.
- The cache line must not remain in the cache.
- The cache line must not propagate beyond its Snoop domain.
- Communicating node pairs:
  - — RN-F to ICN(HN-F).

WriteEvictOrEvict WriteBack of Clean data to the next-level cache.

- Merging of WriteEvictFull and Evict into one request.
- Data might not be sent if the Completer does not accept data.
- If data is sent then the Data size is a cache line length.
- LikelyShared value indicates the initial state of the cache line when the request is sent:

LikelyShared	Initial state
0	UC
1	SC

- Not permitted to assert Exclusive in the request.
- Communicating node pairs:
  - — RN-F to ICN(HN-F).

For WriteEvictOrEvict interaction with Memory Tagging see Chapter 12 *Memory Tagging*.

4.2.4 Combined Write requests

This specification supports the combining of Write transactions with Cache Maintenance transactions when both are to the same address. The ability to combine the two requests to the same address is useful when a CMO or PCMO transaction reaches a point in the system where a Write operation must be completed before the CMO or PCMO transaction can be initiated. The use of a single Combined Write transaction avoids the need to serialize the Write and CMO or PCMO transactions. The point where the two requests are combined can be an RN or an HN.

Table 4-1 and Table 4-2 show the Write, CMO, and PCMO combinations for which combining is permitted. Empty table cells indicate a combination that is not permitted or not applicable.

Note

Deep is an attribute on persistent CMO requests.

When a Persistent CMO is combined with a write, the Persistent CMO is treated as a CleanSharedPersistSep, that is, in addition to a Completion response, a separate Persist response is required for the request.

Table 4-1 Permitted combinations of Write and CMO for RN to HN requests

Write type	**Persistent CMO**	**Non-persistent CMO**		
	(With separate Persist response)	**CleanShared**	**CleanInvalid**	**MakeInvalid**
WriteBackFull	Yes	Yes	Yes	-
WriteBackPtl	-	-	-	-
WriteCleanFull	Yes	Yes	-	-
WriteEvictFull	-	-	-	-
WriteNoSnp	Yes	Yes	Yes	-
WriteUnique	Yes	Yes	-	-
WriteUniqueStash	-	-	-	-

Table 4-2 Permitted combinations of Write and CMO for HN to SN requests

Write type	**Persistent CMO**	**Non-persistent CMO**		
	(With separate Persist response)	**CleanShared**	**CleanInvalid**	**MakeInvalid**
WriteNoSnp	Yes	Yes	Yes	-

Examples of how combining of a Write request with Cache Maintenance is beneficial are:

- Combining WriteBack with CleanShared and CleanSharedPersist, is to support Request Nodes that transition the cache line to Invalid when cleaned by a CMO irrespective of the type of CMO.
- Combining WriteUnique with CleanShared is to support the case when the CleanSharedPersist target is known to not support Persistent transactions and therefore the CleanSharedPersist request needs to be converted to CleanShared by the Requester.

For each Write request the Combined Write requests are:

- WriteNoSnpFull:
    - — WriteNoSnpFullCleanSh.
    - — WriteNoSnpFullCleanInv.
    - — WriteNoSnpFullCleanShPerSep.
- WriteNoSnpPtl:
    - — WriteNoSnpPtlCleanSh.
    - — WriteNoSnpPtlCleanInv.
    - — WriteNoSnpPtlCleanShPerSep.
- WriteUniqueFull:
    - — WriteUniqueFullCleanSh.
    - — WriteUniqueFullCleanShPerSep.
- WriteUniquePtl:
    - — WriteUniquePtlCleanSh.
    - — WriteUniquePtlCleanShPerSep.
- WriteBackFull:
    - — WriteBackFullCleanSh.
    - — WriteBackFullCleanInv.
    - — WriteBackFullCleanShPerSep.
- WriteCleanFull:
    - — WriteCleanFullCleanSh.
    - — WriteCleanFullCleanShPerSep.

Characteristics

A Combined Write request is permitted for both CopyBack and Non-CopyBack writes.

All permitted behaviors of the Combined Write request are the same as if the write and CMO are sent separately, except that:

- WriteNoSnp(Excl) is not permitted to be combined with a CMO.
- TagOp setting of *Match* is not permitted in Write∗CMO.

A Home receiving a Combined Write request from an RN is permitted to forward the Combined Write to the SN if both the write and the CMO or PCMO in the request need to be sent downstream. The Home is also permitted to use DWT for such a Write request if the write from the RN is a Non-CopyBack write and ExpCompAck is set to zero.

Where an uncombined CMO from an RN results in the cache line being evicted from a cache at the Home or a Dirty copy is passed in a Snoop response, the CMO propagated to the Slave can be combined with the Write-Back to the Slave.

The receiver of a Write and CMO combined request is permitted to separate the write and the CMO request and process them separately. In such a case, the CMO request must be ordered behind the write. Once the requests are separated, the Completer is permitted to interleave requests to the same address between the write and the CMO transaction.

The Size field value in a combined request corresponds to the size of Data in the Write request. The Cache Maintenance Operation is always on a 64-byte granularity.

The MemAttr and SnpAttr field values in a Combined Write request correspond to the memory attributes of the Write request. When the write and the CMO transactions are separated, the Write transaction inherits the MemAttr and SnpAttr values of the original combined request. The SnpAttr and Cacheable bit values of the separated CMO transaction must be set to the most pervasive.

A Completer that responds with a DBIDRespOrd response guarantees to order all subsequent requests to the same address from the same Requester behind both the write and the CMO transactions.

The value of the DoDWT field is used to select the TgtID and TxnID values for the DBIDResp response. The Persist response, if required to complete the transaction, always uses the ReturnNID value of the Request for the TgtID of the Persist response. This is irrespective of the DoDWT field value. See *DoDWT* on page 13-417.

4.2.5 Atomic transactions

An Atomic transaction permits a Requester to send to the interconnect a transaction with a memory address and an operation to be performed on that memory location. This transaction type moves the operation closer to where the data resides and is useful for atomically executing an operation and updating the memory location in a performance efficient manner.

Without an Atomic transaction, an atomic operation has to be executed using a sequence of memory accesses. These accesses might rely on Exclusive reads and writes.

Using an Atomic transaction:

- A more deterministic latency can be estimated for atomic operations.
- The blocking period of access to the memory location being modified is reduced, which then reduces the impact on the forward progress of memory accesses by other agents.
- Providing fairness among different Requesters accessing a memory location becomes simpler, because accessing of that memory location by an atomic operation is arbitrated at the PoS or PoC.

This specification defines the following terms relating to atomic operations and Atomic transactions:

Atomic operation	The execution of a function involving multiple data values such that, the loading of the original value, the execution of the function, and the storing of the updated value, occurs in an atomic manner so that no other agent has access to the location during the entire operation.
Atomic transaction	A transaction that is used to pass an atomic operation, along with the data values required for the execution of the atomic operation, from one agent in a system to another, so that the atomic operation can be carried out by a different component in the system than the component that requires the operation to be performed.

Atomic transaction types

This specification defines four Atomic transaction types:

- AtomicStore.
- AtomicLoad.
- AtomicSwap.
- AtomicCompare.

See Chapter 12 *Memory Tagging* for a description of the Memory Tagging mechanism.

For information on the permitted MTE TagOp values for each Atomic request see Table 12-2 on page 12-378.

The following terminology is used to refer to the different data elements in the execution of an atomic operation:

TxnData	The write data in the AtomicLoad, and AtomicStore transactions.
CompareData	The compare value in the AtomicCompare transaction.
SwapData	The swap value in the AtomicCompare, and AtomicSwap transactions.
InitialData	The content of the addressed location before the atomic operation.

Enumeration of the four Atomic transaction types is as follows:

AtomicStore

- Sends a single data value with an address and the atomic operation to be performed.
- The target, an HN or an SN, performs the required operation on the address location specified with the data supplied in the Atomic transaction.
- The target returns a completion response without data.
- Outbound data size is 1, 2, 4, or 8 byte.
- Only appropriate byte enables must be asserted.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.
- Number of operations supported is 8.

Table 4-3 shows the eight operations supported by the AtomicStore transaction.

Table 4-3 AtomicStore operations

STADD	• Update location with (TxnData + InitialData). • InitialData is not returned to the Requester.
STCLR	• Update location with (InitialData AND (NOT TxnData)). Bitwise. • InitialData is not returned to the Requester.
STEOR	• Update location with (InitialData XOR TxnData). Bitwise. • InitialData is not returned to the Requester.
STSET	• Update location with (InitialData OR TxnData). Bitwise. • InitialData is not returned to the Requester.
STSMAX	• Update location with TxnData if: — (((Signed INT) TxnData – (Signed INT) InitialData) > 0). • InitialData is not returned to the Requester.
STSMIN	• Update location with TxnData if: — (((Signed INT) TxnData – (Signed INT) InitialData) < 0). • InitialData is not returned to the Requester.
STUMAX	• Update location with TxnData if: — (((Unsigned INT) TxnData – (Unsigned INT) InitialData) > 0). • InitialData is not returned to the Requester.
STUMIN	• Update location with TxnData if: — (((Unsigned INT) TxnData – (Unsigned INT) InitialData) < 0). • InitialData is not returned to the Requester.

Each of the AtomicStore operations apply to 1, 2, 4, or 8 byte data sizes.

AtomicLoad

- Sends a single data value with an address and the atomic operation to be performed.
- The target, an HN or an SN, performs the required operation on the address location specified with the data value supplied in the Atomic transaction.
- The target returns the completion response with data. The data value is the original value at the addressed location.
- Data will not be cached at the Requester.
- Outbound data size is 1, 2, 4, or 8 byte.
- Only appropriate byte enables must be asserted.
- Inbound data size is the same as the outbound data size.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.
- Number of operations supported is 8.

Table 4-4 shows the eight operations supported by the AtomicLoad transaction.

Table 4-4 AtomicLoad operations

LDADD	• Update location with (TxnData + InitialData). • Return InitialData to the Requester.
LDCLR	• Update location with (InitialData AND (NOT TxnData)). Bitwise. • Return InitialData to the Requester.
LDEOR	• Update location with (InitialData XOR TxnData). Bitwise. • Return InitialData to the Requester.
LDSET	• Update location with (InitialData OR TxnData). Bitwise. • Return InitialData to the Requester.
LDSMAX	• Update location with TxnData if: — (((Signed INT) TxnData – (Signed INT) InitialData) > 0). • Return InitialData to the Requester.
LDSMIN	• Update location with TxnData if: — (((Signed INT) TxnData – (Signed INT) InitialData) < 0). • Return InitialData to the Requester.
LDUSMAX	• Update location with TxnData if: — (((Unsigned INT) TxnData – (Unsigned INT) InitialData) > 0). • Return InitialData to the Requester.
LDUMIN	• Update location with TxnData if: — (((Unsigned INT) TxnData – (Unsigned INT) InitialData) < 0). • Return InitialData to the Requester.

Each of the AtomicLoad operations apply to 1, 2, 4, or 8 byte data sizes.

AtomicSwap

- Sends a single data value, the swap value, together with the address of the location to be operated on.
- The target, an HN or an SN, swaps the value at the address location with the data value supplied in the transaction.
- The target returns the completion response with data. The data value is the original value at the addressed location.
- Data will not be cached at the Requester.
- Outbound data size is 1, 2, 4, or 8 byte.
- Only appropriate byte enables must be asserted.
- Inbound data size is the same as the outbound data size.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.
- Number of operations supported is 1.

AtomicCompare

- Sends two data values, the compare value and the swap value, with the address of the location to be operated on.
- The target, an HN or an SN, compares the value at the addressed location with the compare value:
  - If the values match, the target writes the swap value to the addressed location.
  - If the values do not match, the target does not write the swap value to the addressed location.
- The target returns the completion response with data. The data value is the original value at the addressed location.
- Data will not be cached at the Requester.
- Outbound data size is 2, 4, 8, 16, or 32 byte.
- Only appropriate byte enables must be asserted.
- Inbound data size is half of the outbound data size.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.
- Number of operations supported is 1.

Other common characteristics of Atomic transactions are:

- Transaction ordering is supported for Atomic transactions:
  - In Atomic transactions to a Normal memory region, only Request Order is permitted to be asserted, Endpoint Order must not be asserted.
  - In Atomic transactions to a Device region, both Request Order and Endpoint Order are permitted to be asserted.
- The Completer must wait for all snoop responses before sending the Comp or CompData response.

When the Slave Node supports the execution of atomic operations, the Home is permitted to forward Atomic transactions to the Slave Node. Home cannot use DMT for Non-store atomics that are forwarded to SN. The rules governing such forwarding are:

- The Atomic transaction must be sent to the Slave Node only after all the required Snoop transactions are completed and any Dirty cached data is written back to the Slave Node.
- The Slave Node can either send a separate Comp and DBIDResp or a combined CompDBIDResp as a response to the Atomic Store transaction. For AtomicLoad, AtomicSwap, and AtomicCompare transactions the Slave node must send DBIDResp and Comp with Data as CompData. CompData must use the value of ReturnNID and ReturnTxnID from the request as the value of TgtID and TxnID respectively.

  Note

  By separating the Comp and DBIDResp responses, the Slave Node has an opportunity to signal an error in the received data, or an error during execution of the atomic operation.

- Home must send Atomic transaction data after receiving DBIDResp without waiting for completion.
- Home is permitted to send the completion response to the Requester without waiting for the initiation or completion of the Atomic transaction at the Slave Node.

A Requester with a cache can handle an Atomic transaction request to a Snoopable memory region as follows:

- If the cache line is Unique, then it can perform the atomic operation locally without generating an Atomic transaction.
- If the cache line is Shared but not Dirty, it can either:
  - Generate a ReadUnique or CleanUnique to gain ownership of the cache line and perform the atomic operation locally.
  - Invalidate the local copy and send the Atomic transaction to the interconnect.
- If the cache line is Shared Dirty, it can either:
  - Generate a CleanUnique or ReadUnique, gain ownership of the cache line, and perform the operation locally.
  - WriteBack and Invalidate the local copy and then send the Atomic transaction to the interconnect.
- Optionally, in all the above cases, the Requester is permitted to send the Atomic transaction with the SnoopMe bit set to direct the interconnect to send a Snoop request to the Requester to invalidate, and if required, extract the cached copy. See *SnoopMe* on page 13-419.

4.2.6 Other transactions

This section describes the protocol transactions that carry out miscellaneous actions.

See Chapter 12 *Memory Tagging* for a description of the Memory Tagging mechanism.

For information on the permitted MTE TagOp values for each miscellaneous request see Table 12-2 on page 12-378.

DVM transactions

DVM transactions are used for virtual memory system maintenance.

DVMOp DVM Operation. Actions include the passing of messages between components within a distributed virtual memory system. See Chapter 8 *DVM Operations* for details.

- Communicating node pairs:
  - RN-F, RN-D to ICN(MN).

Prefetch transaction

The prefetch target transaction is used to speculatively fetch data from main memory.

PrefetchTgt Prefetch Target. A Request to a Snoopable memory address, sent from a Request Node directly to a Slave Node:

- The PrefetchTgt transaction does not include a response.
- The request can be used by the Slave Node to fetch the data from off-chip memory and buffer it in anticipation of a subsequent Read request to the same location.
- The request does not include a response, therefore the Requester can deallocate the request as soon as the request is sent.
- The following fields are inapplicable and can take any value:
  - TxnID.
  - Order.
  - Endian.
  - Size.
  - MemAttr.
  - SnpAttr.
  - Excl.
  - LikelyShared.
- The Receiver must not send any response, including RetryAck.
- The Receiver is permitted to initiate an internal action or discard the request without any further action.
- Data read from off-chip memory using PrefetchTgt must not hold Slave Node resources waiting indefinitely for a future Read request to the same address.
- Communicating node pairs:
  - RN-F, RN-D, RN-I to SN-F.

The receiver must accept the request without any dependency on receiving of a subsequent Read request to the same address.

4.3 Snoop request types

For details of Snoop transaction interaction with the *Memory Tagging Extension* (MTE) see Chapter 12 *Memory Tagging*.

The ICN generates a snoop request either in response to a request from an RN or due to an internal cache or snoop filter maintenance operation. A snoop transaction, except for SnpDVMOp, operates on the cached data at the RN-F. A SnpDVMOp transaction carries out a DVM maintenance operation at the target node.

This specification permits snoops to Non-snoopable address locations.

SnpOnceFwd, SnpOnce

Snoop request to obtain the latest copy of the cache line, preferably without changing the state of the cache line at the Snoopee:

- SnpOnceFwd is permitted to be sent only to one RN-F.
- See *Forwarding Snoop transactions* on page 4-222 for the permitted responses to SnpOnceFwd.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for the permitted responses to SnpOnce.
- Expected not to change cache state.

SnpStashUnique

Snoop request recommending that the Snoopee obtains a copy of the cache line in a Unique state:

- Permitted to be sent to only one RN-F.
- This specification recommends not sending the snoop for a StashOnceUnique request if the cache line is cached in Unique state at the Stash target.
- Permitted to send the snoop to the Stash target for WriteUniqueFullStash and WriteUniquePtlStash only if the Snoopee does not have a cached copy of the cache line.
- The Snoopee must not return data with the Snoop response.
- Permits the Snoop response to include a Data Pull.
- Data Pull request in the Snoop response is treated as a ReadUnique.
- Must not change the cache line state at the Snoopee.

SnpStashShared

Snoop request recommending that the Snoopee obtains a copy of the cache line in a Shared state:

- Permitted to be sent to only one RN-F.
- This specification recommends not sending the snoop if the cache line is cached at the target.
- The Snoopee must not return data with the Snoop response.
- Permits the Snoop response to include a Data Pull.
- Must not change the cache line state at the Snoopee.

SnpCleanFwd, SnpClean

Snoop request to obtain a copy of the cache line in Clean state while leaving any cache copy in Shared state:

- SnpCleanFwd is permitted to be only sent to one RN-F.
- See *Forwarding Snoop transactions* on page 4-222 for permitted responses to SnpCleanFwd.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted responses to SnpClean.
- Must not leave the cache line in Unique state.

SnpNotSharedDirtyFwd, SnpNotSharedDirty

Snoop request to obtain a copy of the cache line in SharedClean state while leaving any cached copy in a Shared state:

- SnpNotSharedDirtyFwd is permitted to be sent only to one RN-F.
- See *Forwarding Snoop transactions* on page 4-222 for permitted responses to SnoopNotSharedDirtyFwd.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted responses to SnpNotSharedDirty.
- Must not leave the cache line in Unique state.

SnpSharedFwd, SnpShared

Snoop request to obtain a copy of the cache line in Shared state while leaving any cached copy in Shared state:

- SnpSharedFwd is permitted to be only sent to one RN-F.
- See *Forwarding Snoop transactions* on page 4-222 for permitted responses to SnpSharedFwd.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted responses to SnpShared.
- Must not leave the cache line in Unique state.

SnpUniqueFwd, SnpUnique

Snoop request to obtain a copy of the cache line in Unique state while invalidating any cached copies:

- SnpUniqueFwd is permitted to be sent to only one RN-F.
- See *Forwarding Snoop transactions* on page 4-222 for permitted responses to SnpUniqueFwd.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted responses to SnpUnique.
- Must change the cache line to Invalid state.

SnpPreferUniqueFwd, SnpPreferUnique

Snoop request to obtain a copy of the cache line in Unique state while invalidating any cached copies:

- SnpPreferUniqueFwd is permitted to be sent to only one RN-F.
- Home is expected to use SnpPreferUniqueFwd or SnpPreferUnique in response to ReadPreferUnique.
- The behavior of the Snoopee is dependent on whether it is executing an exclusive sequence. See *SnpPreferUnique and SnpPreferUniqueFwd* on page 4-229.

SnpUniqueStash

Snoop request to invalidate the cached copy at the Snoopee and recommends that the Snoopee obtains a copy of the cache line in Unique state:

- Permitted to be sent to only one RN-F.
- See *Stash snoop transactions* on page 4-220 for responses to SnpUniqueStash.
- Permits the Snoop response to include a Data Pull.
- Data Pull request in the Snoop response is treated as a ReadUnique.

SnpCleanShared Snoop request to remove any Dirty copy of the cache line at the Snoopee:

- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted SnpCleanShared responses.
- Must not leave the cache line in a Dirty state.

SnpCleanInvalid

Snoop request to Invalidate the cache line at the Snoopee and obtain any Dirty copy. Might also be generated by the ICN without a corresponding request:

- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted SnpCleanInvalid responses.
- Must change the cache line to Invalid state.

SnpMakeInvalid

Snoop request to Invalidate the cache line at the Snoopee and discard any Dirty copy:

- Does not return data with the Snoop response, Dirty data is discarded.
- See *Non-Forwarding and Non-stash Snoop transactions* on page 4-214 for permitted SnpMakeInvalid responses.
- Must change the cache line to Invalid state.

SnpMakeInvalidStash

Snoop request to invalidate the copy of the cache line and recommends that the Snoopee obtains a copy of the cache line in Unique state:

- Permitted to be sent to only one RN-F.
- Snoopee must not return data with the Snoop response, Dirty data must be discarded.
- See *Stash snoop transactions* on page 4-220 for the permitted SnpMakeInvalidStash responses.
- Permits the Snoop response to include a Data Pull.
- Data Pull request in the Snoop response is treated as a ReadUnique.

SnpQuery

SnpQuery probes the state of a cache line at a Request Node:

- Home can send a SnpQuery snoop without any corresponding request from a Requester.
- The Snoop response must include the precise state of the cache line at the targeted Snoopee.
- Snoopee must not return data with the Snoop response.
- See *SnpQuery* on page 4-219 for the permitted SnpQuery responses.
- The SnpQuery snoop must not change the state of the cache line at the Snoopee.

See *MakeReadUnique transaction* on page 4-206 and *MakeReadUnique(Excl)* on page 6-279 to see how SnpQuery can be used in the efficient handling of exclusive request flows.

SnpDVMOp

Generated at the ICN, initiated by the DVMOp request:

- A single DVMOp request generates two snoop requests.
- Returns a single Snoop response for the two snoop requests.

  See *Non-sync type DVM transaction flow* on page 8-294.

4.4 Request types and corresponding Snoop requests

Table 4-5 shows the Request transactions and the corresponding Snoop transactions that are generated by the interconnect. A Requester can implement a subset of the Request transactions, but must be able to respond to all Snoop transactions.

Table 4-5 Request types and the corresponding snoop requests

Request type	Request	Snoop		
		Expected	Alternative snoop	Snoop to non-target
Read	ReadNoSnp	n/a	SnpOnceFwd	n/a
	ReadNoSnpSep	n/a	n/a	n/a
	ReadOnce	SnpOnceFwd	SnpOnce	n/a
	ReadOnceCleanInvalid	SnpUnique	SnpOnceFwd	n/a
	ReadOnceMakeInvalid	SnpUnique	SnpOnceFwd	n/a
	ReadClean	SnpCleanFwd	SnpClean	n/a
	ReadNotSharedDirty	SnpNotSharedDirtyFwd	SnpNotSharedDirty	n/a
	ReadShared	SnpSharedFwd	SnpShared	n/a
	ReadUnique	SnpUniqueFwd	SnpUnique	n/a
	ReadPreferUnique	SnpPreferUniqueFwd	SnpPreferUnique	n/a
	MakeReadUnique	SnpCleanInvalid	SnpUnique SnpUniqueFwd[a] SnpMakeInvalid	n/a
Dataless	CleanUnique	SnpCleanInvalid	n/a	n/a
	MakeUnique	SnpMakeInvalid	SnpCleanInvalid	n/a
	Evict	n/a	n/a	n/a
	CleanShared	SnpCleanShared	n/a	n/a
	CleanSharedPersist*	SnpCleanShared	n/a	n/a
	CleanInvalid	SnpCleanInvalid	n/a	n/a
	MakeInvalid	SnpMakeInvalid	SnpCleanInvalid	n/a
Dataless-Stash	StashOnceUnique StashOnceSepUnique	SnpStashUnique	n/a	n/a
	StashOnceShared StashOnceSepShared	SnpStashShared	n/a	n/a

Table 4-5 Request types and the corresponding snoop requests (continued)

Request type	Request	Snoop		
		Expected	Alternative snoop	Snoop to non-target
Write	WriteNoSnp	n/a	n/a	n/a
	WriteNoSnpZero	n/a	n/a	n/a
	WriteUniqueFull	SnpMakeInvalid	SnpCleanInvalid SnpUnique	n/a
	WriteUniquePtl	SnpCleanInvalid or SnpUnique	n/a	n/a
	WriteUniqueZero	SnpMakeInvalid	SnpCleanInvalid	n/a
Write-Stash	WriteUniqueFullStash	SnpMakeInvalidStash	SnpUniqueStash	SnpMakeInvalid SnpCleanInvalid SnpUnique
	WriteUniquePtlStash	SnpUniqueStash	n/a	SnpUnique
Write - CopyBack	WriteBack	n/a	n/a	n/a
	WriteClean	n/a	n/a	n/a
	WriteEvictFull	n/a	n/a	n/a
	WriteEvictOrEvict	n/a	n/a	n/a
Atomic	AtomicStore	SnpUnique	n/a	n/a
	AtomicLoad	SnpUnique	n/a	n/a
	AtomicSwap	SnpUnique	n/a	n/a
	AtomicCompare	SnpUnique	n/a	n/a
Others	DVMOp	SnpDVMOp	n/a	n/a
	PCrdReturn	n/a	n/a	n/a
	PrefetchTgt	n/a	n/a	n/a

a. Home is permitted to use SnpUniqueFwd if it determines that the Requester has lost its copy of the cache line.

The interconnect has the following behavior when generating a snoop request on receipt of a request from an RN:

- This specification supports a snoop filter or directory within the interconnect to track the state of cache lines present in RN-F caches. The tracking can be as detailed as knowing each RN-F that has a copy of the cache line, or as nonspecific as knowing that a cache line is present in one of the RN-F caches. Such tracking permits the ICN to filter unnecessary snooping of an RN-F, for example:
    - If the snoop filter indicates that the cache line is not present in any of the RN-F caches, then the interconnect does not send a snoop request.
    - If the cache line in the RN-F caches is already in the required state, for example the received request is ReadShared and all cached copies of the cache line are in SharedClean (SC) state, then the interconnect does not send a snoop request.
- The interconnect in response to a WriteUniqueFull, WriteUniqueFullStash, MakeUnique and MakeInvalid must not use the SnpMakeInvalid snoop request unless either:
    - The transaction TagOp value is *Update*.
    - The interconnect can determine that the Snoopee does not hold Dirty tags.
- The interconnect in response to a MakeReadUnique must not use the SnpMakeInvalid Snoop request unless the interconnect can determine either that:
    - The Requester still has a cached copy of the cache line and the Snoopee does not have Dirty tags.
    - The Requester has lost its cached copy and the Snoopee does not have a Dirty copy of the cache line.
- It is permitted for the interconnect to generate a snoop request spontaneously without a corresponding request from an RN. For example, the interconnect can send a SnpUnique or SnpCleanInvalid request as a result of a backward invalidation from a snoop filter or interconnect cache.
- This specification permits the interconnect to select which snoop request to send. For example:
    - For a WriteUniquePtl request, either a SnpCleanInvalid or SnpUnique snoop request can be sent. Both of these snoop transactions invalidate the cache line and if the cache line is dirty then data is returned with the response. The write data is written to memory once all Snoop responses are received and the partial data has been merged with any dirty data received with the Snoop response.

      The only difference in the behavior between the SnpCleanInvalid and SnpUnique snoop requests is that SnpUnique can return data from the UniqueClean (UC) state but SnpCleanInvalid does not. Using SnpUnique therefore might result in an unnecessary data transfer. This example shows the disadvantage of using SnpUnique instead of SnpCleanInvalid in certain circumstances.
- This specification permits the interconnect to:
    - Use SnpNotSharedDirty or SnpShared or SnpClean for ReadNotSharedDirty, ReadShared and ReadClean transactions.
    - Use SnpNotSharedDirtyFwd or SnpSharedFwd or SnpCleanFwd for ReadShared transactions.
    - Use SnpNotSharedDirtyFwd or SnpCleanFwd for ReadNotSharedDirty and ReadClean transactions.
    - Use any non-Forwarding, non-invalidating snoop types for the ReadOnce transactions.
    - Use any non-Forwarding snoop types except SnpMakeInvalid for the ReadOnceCleanInvalid and ReadOnceMakeInvalid transactions.
    - Use Forwarding snoop type SnpOnceFwd for the ReadOnce, ReadOnceCleanInvalid and ReadOnceMakeInvalid transactions.
    - Send SnpStashUnique or SnpMakeInvalidStash to the target RN for WriteUniqueFullStash and WriteUniquePtlStash if the target RN does not have the cache line.
    - Replace any invalidating Snoop request by the SnpUnique or SnpCleanInvalid request.
    - Replace any Forwarding snoop with a corresponding non-Forwarding type. The receiver is permitted to treat the forward indication as a hint and respond to the snoop with a corresponding non-Forwarding version in a protocol compliant manner. This is permitted irrespective of the use of MTE.
    - Use of SnpMakeInvalid for MakeInvalid and WriteUniqueZero is permitted only when the Home knows that the Snoopee does not have Dirty tags.

4.5 Response types

Each request can generate one or more responses. Some responses can also include data. A Response is classified as follows:

- *Completion response*.
- *WriteData response* on page 4-192.
- *Snoop response* on page 4-194.
- *Miscellaneous response* on page 4-200.

4.5.1 Completion response

A completion response is required for all transactions except PCrdReturn and PrefetchTgt. It is typically the last message sent, from the Completer, to conclude a request transaction. The Requester might, however, still need to send a CompAck response to conclude the transaction. A completion guarantees that the request has reached a PoS or a PoC, where it will be ordered with respect to requests to the same address from any Requester in the system. See *Ordering* on page 2-116 for details on the Ordering guarantees.

Read and Atomic transaction completion

A Read Completion is either in the form of a single response on the RDAT channel using the CompData opcode, or two separate responses, one on the RSP channel using the RespSepData opcode, and a second one on the RDAT channel using the DataSepResp opcode. See *Dataless transaction completion* on page 4-191 for completion without data for the MakeReadUnique transaction.

AtomicLoad, AtomicSwap, and AtomicCompare Completion is sent on the RDAT channel and uses the CompData opcode.

The CompData and DataSepResp completion responses include the Resp field that indicates the following:

Cache state The final permitted state of the cache line at the Requester for all reads except ReadNoSnp and ReadOnce*.

Pass Dirty Indicates if the responsibility for updating memory is passed to the Requester. The assertion of the Pass Dirty bit is shown by _PD in the response name.

When using separate Comp and Data responses, RespSepData also includes the Resp field with Cache state and Pass Dirty indications. The Resp field value in RespSepData must be either inapplicable and set to zero or the same as in the corresponding DataSepResp.

Table 4-6 shows the permitted Read transaction completion, the encoding of the Resp field, and the meaning of the response. An SN can send DataSepResp only in response to ReadNoSnpSep, and only CompData in response to ReadNoSnp.

Table 4-6 Permitted Read transaction completion and Resp field encodings

Response	Resp[2:0]	Description
CompData_I DataSepResp_I	`0b000`	Indicates that a coherent copy of the cache line cannot be kept.
RespSepData_I	`0b000`	Cache state in this response is not applicable. Cache state must be determined from DataSepResp response.
CompData_UC DataSepResp_UC RespSepData_UC	`0b010`	The final state of the cache line can be UC, UCE, SC or I, when the cache state in the response is applicable. This response is also permitted for ReadNoSnp and ReadOnce* transactions but the cache line will not be coherent. Responsibility for a Dirty cache line is not being passed.

Table 4-6 Permitted Read transaction completion and Resp field encodings (continued)

Response	Resp[2:0]	Description
CompData_SC DataSepResp_SC RespSepData_SC	`0b001`	The final state of the cache line can be SC or I. Responsibility for a Dirty cache line is not being passed.
CompData_UD_PD DataSepResp_UD_PD RespSepData_UD_PD	`0b110`	The final state of the cache line can be UD or SD. Responsibility for a Dirty cache line is being passed.
CompData_SD_PD	`0b111`	The final state of the cache line must be SD. Responsibility for a Dirty cache line is being passed.

In a response with an error indication, the cache state is permitted to be any value, including reserved values. See *Errors and transaction structure* on page 9-326.

Dataless transaction completion

A completion for Dataless transactions and the MakeReadUnique transaction without data is sent on the CRSP channel and uses the Comp, CompPersist, CompCMO or CompStashDone opcode CompCMO is the completion message for the CMO and PCMO in a Combined Write transaction. It must be used only in Combined Write transactions.

The Comp response includes the Resp field that indicates the following:

Cache state	The final state the cache line is permitted to be in at the Requester, except for CMO transactions. For CMO transactions, the cache state field value is ignored and the cache state remains unchanged.

Table 4-7 shows the permitted Dataless transaction completion, the encoding of the Resp field, and the meaning of the response.

Table 4-7 Permitted Dataless transaction completion and Resp field encodings

Response	Resp[2:0]	Description
Comp_I	`0b000`	The final state of the cache line must be I
Comp_UC	`0b010`	The final state of the cache line can be UC, UCE, SC or I
Comp_SC	`0b001`	The final state of the cache line can be SC or I
Comp_UD_PD	`0b110`	The final state of the cache line must be UD or SD. Responsibility for a Dirty cache line is being passed.

In a response with an error indication, the cache state is permitted to be any value, including reserved values. See *Errors and transaction structure* on page 9-326. See *CompCMO and Comp responses in Combined Write transactions* on page 2-76 for CompCMO and CompPersist. See *Independent Stash request* on page 7-289 for CompStashDone.

Write and Atomic transaction completion

A Write and AtomicStore Completion is sent on the CRSP channel and uses the Comp or CompDBIDResp opcode.

No cache state information, or responsibility for a Dirty cache line, is communicated using the Write transaction completion. The Resp field of a Comp or CompDBIDResp response must be set to zero for a Write transaction completion. All cache state information and responsibility for a Dirty cache line are communicated with the WriteData, See *WriteData response*.

The permitted Write transaction completion responses are:

Comp	Used when the Completion response is separate from the DBIDResp response.
CompDBIDResp	Used when the Completion response is combined with the DBIDResp response. All CopyBack requests must use the CompDBIDResp completion response. Non-CopyBack writes and AtomicStore, can either send Comp and DBIDResp responses separately or can opportunistically combine the Comp and DBIDResp responses and send CompDBIDResp if both are ready to be sent to the Requester.

Miscellaneous transaction completion

A Comp response, with the Resp field set to zero, is always used for DVM transaction completion.

4.5.2 WriteData response

The WriteData response is part of Write request and DVMOp transactions. The Requester sends WriteData to the Completer after receiving a guarantee that a buffer is available to accept the data. Buffer availability is signaled through a DBIDResp response sent from the Completer.

The WriteData response is sent on the WDAT channel and uses the following opcodes.

CopyBackWrData	• Used for WriteBack, WriteClean, WriteEvictFull, and WriteEvictOrEvict, and CopyBack Combined Write transactions. • Transfers coherent data from the cache at the Requester to the interconnect. • Includes an indication of the cache line state prior to sending the WriteData response.
NonCopyBackWrData	• Used for WriteUnique and WriteNoSnp, and Non-Copyback Combined Write transactions. • Also used for a DVMOp transaction. • The cache state in the response must be I.
NCBWrDataCompAck	• Used for WriteUnique and WriteNoSnp transactions. • Combined NonCopyBackWrData and CompAck. • The cache state in the response must be I.
WriteDataCancel	• Used to inform the Completer that a Write request is canceled before write data is sent. • Used for WriteUniquePtl, WriteUniquePtlStash and WriteNoSnpPtl transactions from RN to HN with Normal memory attribute. • Used for WriteNoSnp and WriteNoSnpPtl from HN-F to SN-F with Normal memory attribute. • Must not be used in Write requests to Device memory. • All data packets originally intended to be transferred must be sent. • BE field value in the WriteDataCancel message must be set to all zeroes. • Cache state in the response must be I.

The response includes the Resp field, which indicates the following:

Cache state	Indicates the state of the cache line before sending the WriteData response. This state can differ from the state of the cache line when the original transaction request was sent if a snoop request, to the same address, is received by the Requester after sending the original transaction request, but before sending the corresponding WriteData response.
Pass Dirty	Indicates if the responsibility for updating memory is passed by the Requester. The assertion of the Pass Dirty bit is shown by _PD in the response name.

Table 4-8 shows the permitted WriteData responses, the Opcode and Resp field encodings, and the meaning of the response.

Table 4-8 Permitted WriteData responses and Opcode and Resp field encodings

Response	DAT Opcode	Resp[2:0]	Description
CopyBackWrData_I	`0x2`	`0b000`	Data corresponding to a CopyBack request. Cache line state when data was sent is I.
CopyBackWrData_UC	`0x2`	`0b010`	Data corresponding to a CopyBack request. Cache line state when data was sent is UC.
CopyBackWrData_SC	`0x2`	`0b001`	Data corresponding to a CopyBack request. Cache line state when data was sent is SC.
CopyBackWrData_UD_PD	`0x2`	`0b110`	Data corresponding to a CopyBack request. Cache line state when data was sent is UD or UDP. Responsibility for updating the memory is passed.
CopyBackWrData_SD_PD	`0x2`	`0b111`	Data corresponding to a CopyBack request. Cache line state when data was sent is SD. Responsibility for updating the memory is passed.
NonCopyBackWrData	`0x3`	`0b000`	Data corresponding to a Non-CopyBack Write request.
NCBWrDataCompAck	`0xC`	`0b000`	Data corresponding to a Non-CopyBack Write request and combined CompAck to indicate that the transaction has completed.
WriteDataCancel	`0x7`	`0b000`	Data corresponding to a canceled Non-CopyBack Write request.

Note

The cache line state at the Requester after the Write transaction has completed is not determined from the cache state information in the WriteData response. It can be determined if the cache line remains Valid or not after the transaction by the opcode of the transaction:

- A WriteBack or WriteEvictFull transaction must be in I state.
- A WriteClean transaction can remain allocated and be in a Clean state.

4.5.3 Snoop response

A Snoop transaction includes a Snoop response. A Snoop response can be with or without data. The forms of Snoop response are:

Snoop response without data

- This Snoop response is used when no data transfer is required.
- It is sent on the SRSP channel and uses the SnpResp opcode.
- It can include a Data Pull request for Stashing snoops.
- Snoop response without data is always used for the response to a SnpDVMOp transaction.

Snoop response without Data to Home and *Direct Cache Transfer* (DCT)

- This Snoop response is used when the Snoopee sends Data to the Requester and a data transfer to the Home is not required.
- It is sent on the SRSP channel and uses the SnpRespFwded opcode.

Snoop response with data

- This Snoop response is used when a full cache line of data is transferred to Home.
- It is sent on the WDAT channel and uses the SnpRespData opcode.
- It can include a Data Pull request for Stashing snoops.

Snoop response with partial data

- This Snoop response is used when a partial cache line of data is transferred to the Home.
- It is sent on the WDAT channel and uses the SnpRespDataPtl opcode.
- It can include a Data Pull request for Stashing snoops.
- It is sent when the combination of the Snoop request and cache line state is:
    - Any Snoop request except SnpMakeInvalid, and the cache line state is UDP.

Snoop response with Data to Home and DCT

- This Snoop response is used when the Snoopee sends Data to the Requester and a data transfer to the Home is also required.
- It is sent on the DAT channel and uses the SnpRespDataFwded opcode.

The Snoop response includes the Resp field, which indicates the following:

Cache state The final state of the cache line at the snooped node after sending the Snoop response.

Pass Dirty Indicates that the responsibility for updating memory is passed to the Requester or ICN.

Pass Dirty must only be asserted for a Snoop response with data. The assertion of the Pass Dirty bit is shown by _PD in the response name.

The Snoop response also includes the FwdState field that is applicable in Snoop responses with DCT and indicates the cache state and pass dirty value in the CompData response sent to the Requester.

These attributes convey sufficient information for the interconnect to determine the appropriate response to the initial Requester, and to determine if data must be written back to memory. It is also sufficient to support snoop filter or directory maintenance in the interconnect.

See *Snoop requests* on page 12-373 for details on Snoop responses and MTE interaction.

Note

The Snoop response cache state information provides the state of the cache line after the Snoop response is sent. This is different from:

- A WriteData response, where the cache state information provides the state of the cache line at the point the write data is sent.
- A read data response, where the cache state information indicates the permitted state of the cache line after the transaction completes.

Table 4-9 shows the permitted Non-forward type snoop responses without data, the RSP Opcode and Resp field encodings, and the meaning of the response.

Table 4-9 Permitted Non-forward type snoop responses without data

Response	RSP Opcode	Resp[2:0]	Description
SnpResp_I	`0x1`	`0b000`	Snoop response without data. Cache line state is I.
SnpResp_SC	`0x1`	`0b001`	Snoop response without data. Cache line state is SC, or I.
SnpResp_UC	`0x1`	`0b010`	Snoop response without data. Cache line state is UC, UCE, SC, or I.
SnpResp_UD	`0x1`	`0b010`	Snoop response without data. Cache line state is UD.
SnpResp_SD	`0x1`	`0b011`	Snoop response without data. Cache line state is SD.

Table 4-10 shows the permitted Forward type snoop responses without data, the RSP Opcode, Resp, and FwdState field encodings, and the meaning of the response.

Table 4-10 Permitted Forward type snoop responses without data

Response	RSP Opcode	Resp[2:0]	FwdState[2:0]	Description
SnpResp_I_Fwded_I	`0x9`	`0b000`	`0b000`	Snoop response without data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is I.
SnpResp_I_Fwded_SC	`0x9`	`0b000`	`0b001`	Snoop response without data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is SC.
SnpResp_I_Fwded_UC	`0x9`	`0b000`	`0b010`	Snoop response without data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is UC.
SnpResp_I_Fwded_UD_PD	`0x9`	`0b000`	`0b110`	Snoop response without data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is UD. Responsibility for updating the memory is passed.
SnpResp_I_Fwded_SD_PD	`0x9`	`0b000`	`0b111`	Snoop response without data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is SD. Responsibility for updating the memory is passed.

Table 4-10 Permitted Forward type snoop responses without data (continued)

Response	RSP Opcode	Resp[2:0]	FwdState[2:0]	Description
SnpResp_SC_Fwded_I	`0x9`	`0b001`	`0b000`	Snoop response without data. Cache line state is SC. Copy of data forwarded to the Requester. Forward State is I.
SnpResp_SC_Fwded_SC	`0x9`	`0b001`	`0b001`	Snoop response without data. Cache line state is SC. Copy of data forwarded to the Requester. Forward State is SC.
SnpResp_SC_Fwded_SD_PD	`0x9`	`0b001`	`0b111`	Snoop response without data. Cache line state is SC. Copy of data forwarded to the Requester. Forward State is SD. Responsibility for updating the memory is passed.
SnpResp_UC_Fwded_I SnpResp_UD_Fwded_I	`0x9`	`0b010`	`0b000`	Snoop response without data. Cache line state is UC or UD. Copy of data forwarded to the Requester. Forward State is I. **Note** A single encoding is used to indicate that the cache line is unique. This encoding is used for UC and UD.
SnpResp_SD_Fwded_I	`0x9`	`0b011`	`0b000`	Snoop response without data. Cache line state is SD. Copy of data forwarded to the Requester. Forward State is I.
SnpResp_SD_Fwded_SC	`0x9`	`0b011`	`0b001`	Snoop response without data. Cache line state is SD. Copy of data forwarded to the Requester. Forward State is SC.

Table 4-11 shows the permitted Non-forward type snoop responses with data, the DAT Opcode and Resp field encodings, and the meaning of the response.

Table 4-11 Permitted Non-forward type snoop responses with data

Response	DAT Opcode	Resp[2:0]	Description
SnpRespData_I	`0x1`	`0b000`	Snoop response with data. Cache line state is I.
SnpRespData_UC SnpRespData_UD	`0x1`	`0b010`	Snoop response with data. Cache line state is UC or UD. **Note** A single encoding is used to indicate that the cache line is unique. This encoding is used for UC and UD.
SnpRespData_SC	`0x1`	`0b001`	Snoop response with data. Cache line state is SC.
SnpRespData_SD	`0x1`	`0b011`	Snoop response with data. Cache line state is SD.
SnpRespData_I_PD	`0x1`	`0b100`	Snoop response with data. Cache line state is I. Responsibility for updating the memory is passed to the Home.
SnpRespData_UC_PD	`0x1`	`0b110`	Snoop response with data. Cache line state is UC. Responsibility for updating the memory is passed to the Home.
SnpRespData_SC_PD	`0x1`	`0b101`	Snoop response with data. Cache line state is SC. Responsibility for updating the memory is passed to the Home.
SnpRespDataPtl_I_PD	`0x5`	`0b100`	Snoop response with partial data. Cache line state is I. Responsibility for updating the memory is passed to the Home.
SnpRespDataPtl_UD	`0x5`	`0b010`	Snoop response with partial data. Cache line state is UDP.

Table 4-12 shows the permitted Forward type snoop responses with data, the DAT Opcode, Resp, and FwdState field encodings, and the meaning of the response.

Table 4-12 Permitted Forward type snoop responses with data

Response	DAT Opcode	Resp[2:0]	FwdState[2:0]	Description
SnpRespData_I_Fwded_SC	`0x6`	`0b000`	`0b001`	Snoop response with data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is SC.
SnpRespData_I_Fwded_SD_PD	`0x6`	`0b000`	`0b111`	Snoop response with data. Cache line state is I. Copy of data forwarded to the Requester. Forward State is SD. Responsibility of updating the memory is passed to the Requester.
SnpRespData_SC_Fwded_SC	`0x6`	`0b001`	`0b001`	Snoop response with data. Cache line state is SC. Copy of data forwarded to the Requester. Forward State is SC.
SnpRespData_SC_Fwded_SD_PD	`0x6`	`0b001`	`0b111`	Snoop response with data. Cache line state is SC. Copy of data forwarded to the Requester. Forward State is SD. Responsibility for updating the memory is passed to the Requester.
SnpRespData_SD_Fwded_SC	`0x6`	`0b011`	`0b001`	Snoop response with data. Cache line state is SD. Copy of data forwarded to the Requester. Forward State is SC.
SnpRespData_I_PD_Fwded_I	`0x6`	`0b100`	`0b000`	Snoop response with data. Cache line state is I. Responsibility for updating the memory is passed to the Home. Copy of data forwarded to the Requester. Forward State is I.

Table 4-12 Permitted Forward type snoop responses with data (continued)

Response	DAT Opcode	Resp[2:0]	FwdState[2:0]	Description
SnpRespData_I_PD_Fwded_SC	`0x6`	`0b100`	`0b001`	Snoop response with data. Cache line state is I. Responsibility for updating the memory is passed to the Home. Copy of data forwarded to the Requester. Forward State is SC.
SnpRespData_SC_PD_Fwded_I	`0x6`	`0b101`	`0b000`	Snoop response with data. Cache line state is SC. Responsibility for updating the memory is passed to the Home. Copy of data forwarded to the Requester. Forward State is I.
SnpRespData_SC_PD_Fwded_SC	`0x6`	`0b101`	`0b001`	Snoop response with data. Cache line state is SC. Responsibility for updating the memory is passed to the Home. Copy of data forwarded to the Requester. Forward State is SC.

The cache line state associated with a Snoop response with data must be a legal value, even if the RespErr field indicates there is a Data Error. A Snoop response with data is not permitted to have a Non-data Error. See *Snoop transactions* on page 9-335.

The cache line state associated with a Snoop response without data must be Invalid if the response has a Non-data Error. The responder must invalidate locally cached copies of the cache line. In addition, when the response to a Forwarding snoop results in a Non-data Error the Snoopee must not forward data to the Requester. As a consequence, if the CompData message has already been sent to the Requester then the Snoop response to the Home must not include a Non-data Error.

In responses to Stashing snoops, the Snoopee can send a Read request combined with the Snoop response (SnpResp_X_Read), by setting the DataPull bit. The permitted Snoop responses with Data Pull are:

- For SnpUniqueStash:
  - SnpResp_I_Read.
  - SnpRespData_I_Read.
  - SnpRespData_I_PD_Read.
  - SnpRespDataPtl_I_PD_Read.
- For SnpMakeInvalidStash:
  - SnpResp_I_Read.
- For SnpStashUnique:
  - SnpResp_I_Read.
  - SnpResp_UC_Read.
  - Snp_Resp_SC_Read.
  - SnpResp_SD_Read.
- For SnpStashShared:
  - SnpResp_I_Read.
  - SnpResp_UC_Read.

4.5.4 Miscellaneous response

This section describes responses that cannot be classified as a Completion, WriteData or Snoop response.

For all responses in this section the Resp and RespErr fields have no meaning and must be set to zero.

The miscellaneous responses are:

CompAck

- Sent by the Requester on receipt of the Completion response.
- Used by Read, Dataless, and WriteUnique transactions.

See *Transaction structure* on page 2-42.

RetryAck

- Sent by a Completer to a Requester if the request is not accepted at the Completer due to lack of appropriate resources.
- Response is permitted for any request transaction except PCrdReturn or PrefetchTgt.

See *Transaction Retry sequence* on page 2-82.

PCrdGrant

- Grants a Protocol Credit. A subsequent request, sent using the Protocol Credit, is guaranteed to be accepted by the target.

See *Transaction Retry sequence* on page 2-82.

ReadReceipt

- Sent for a request that requires Request Order in the interconnect with respect to other ordered requests from the same Requester.
- Sent by a Slave Node to indicate it has accepted a Read request and will not send a RetryAck response.
- See *Ordering requirements* on page 2-121 for how the ReadReceipt is used in an ordered request.
- See *The following requirements apply to this type of lifetime reduction transaction:* on page 2-49 for how the ReadReceipt helps in reducing the life time of a Read request at a Home Node.
- Applies to ReadNoSnp, ReadNoSnpSep, and ReadOnce* request transactions.

See *ReadNoSnp, ReadOnce, ReadOnceCleanInvalid, ReadOnceMakeInvalid* on page 2-47.

DBIDResp

- Response sent to signal to the Requester that resources are available to accept the WriteData response.
- DBIDResp response also indicates that the Completer provides certain transaction ordering guarantees. See *Transaction ordering* on page 2-120.
- Applies to Write, Combined Write, and Atomic request transactions.
- The response is permitted from Home Node to Request Node and Slave Node to both a Home Node and Request Node.

See *Transaction structure* on page 2-42.

DBIDRespOrd

- Response sent to signal to the Requester that resources are available to accept the WriteData response.
- DBIDRespOrd response also indicates that the Completer provides certain transaction ordering guarantees. See *Transaction ordering* on page 2-120.
- Applies to Write, Combined Write, and Atomic request transactions.
- DBIDRespOrd is not permitted in DVM transactions.
- The response is permitted from Home Node to Request Node and Slave Node to both a Home Node and Request Node.

See *Transaction ordering* on page 2-120.

Persist

Sent by a Completer for CleanSharedPersistSep transaction to indicate that any data written earlier to the same memory location is made persistent.

See *CMO transactions* on page 2-61.

StashDone

Sent by a Completer for StashOnceSep to signal the ordering of the request at the Completer.

See *Independent Stash request* on page 7-289.

TagMatch

Sent by the Completer for a Write transaction with TagOp of *Match* to signal the completion of the Tag Match operation.

See *Home to Slave transactions* on page 12-375.

4.6 Silent cache state transitions

A cache can change state due to an internal event without notifying the rest of the system.

The legal silent cache state transitions are shown in Table 4-13. In some cases it is possible, but not required, to issue a transaction to indicate that the transition has occurred. If such a transaction is issued then the cache state transition is visible to the interconnect and is not classified as a silent transition.

The RN-F action described in Table 4-13 as *Local sharing*, describes the case where an RN-F specifies a Unique cache line as Shared, effectively disregarding the fact that the cache line remains Unique to the RN-F. For example, this can happen when the RN-F contains multiple internal agents and the cache line becomes shared between them.

For silent cache state transitions:

- Cache eviction and Local sharing transitions can occur at any point and are IMPLEMENTATION DEFINED.
- Store and Cache Invalidate transitions can only occur as the result a deliberate action, which in the case of a core is caused by the execution of a particular program instruction.

The Notes column in Table 4-13 indicates how a silent cache transition can be made non-silent or visible at the interface.

Table 4-13 Legal silent cache state transitions

RN-F action	RN-F Cache state		Notes
	Present	Next	
Cache eviction	UC	I	Can use Evict, WriteEvictFull, or WriteEvictOrEvict transaction
	UCE	I	Can use Evict transaction
	SC	I	Can use Evict transaction
Local sharing	UC	SC	-
	UD	SD	-
Store	UC	UD	Full or partial cache line store
	UCE	UDP	Partial cache line store
	UCE	UD	Full cache line store
	UDP	UD	Store that fills the cache line
Cache Invalidate	UD	I	Can use Evict transaction
	UDP	I	Can use Evict transaction

A cache state change from UC to UCE is not permitted.

Note

Sequences of silent transitions can also occur. Any silent transition that results in the cache line being in UD, UDP, or SC state can undergo a further silent transition.

4.7 Cache state transitions at a Requester

This section specifies the cache state transitions and completion responses for the following request transactions:

- *Read request transactions*.
- *Dataless request transactions* on page 4-211.
- *Write request transactions* on page 4-212.
- *Atomic transactions* on page 4-213.
- *Other request transactions* on page 4-213.

4.7.1 Read request transactions

Table 4-14 shows the cache state transitions at the Requester, and the completion responses, for Read request transactions except for the MakeReadUnique transaction.

For details of the permitted completion responses and cache state transitions at the Requester for the MakeReadUnique transaction, both non-Exclusive and Exclusive, see *MakeReadUnique transaction* on page 4-206.

The cache state in the Data response to the Requester from the Slave Node is UC, that is, CompData_UC irrespective of the original request type. The Requester must ignore the cache state in the CompData response to ReadNoSnp, ReadOnce, ReadOnceCleanInvalid and ReadOnceMakeInvalid and implicitly assume the cache state value to be I.

Note

In a non-DMT data transfer, where the CompData response is sent from the Slave to Home, the cache state in the response can be either I or UC, but it is expected that typically a slave design can be simplified by always using UC. Home then sends CompData to the Requester with the appropriate cache state value.

Table 4-14 Cache state transitions at the Requester for Read request transactions

Request type	Cache state			Comp response	Separate responses
	Initial		Final		
	Expected	Others Permitted			
ReadNoSnp	I	-	I	CompData_UC, CompData_I	RespSepData + DataSepResp_UC
ReadOnce	I, UCE[a]	-	I	CompData_UC, CompData_I	RespSepData + DataSepResp_UC
ReadOnceCleanInvalid	I, UCE[a]	-	I	CompData_UC, CompData_I	RespSepData + DataSepResp_UC
ReadOnceMakeInvalid	I, UCE[a]	-	I	CompData_UC, CompData_I	RespSepData + DataSepResp_UC

Table 4-14 Cache state transitions at the Requester for Read request transactions (continued)

Request type	Cache state				
	Initial		Final	Comp response	Separate responses
	Expected	Others Permitted			
ReadClean TagOp = *Transfer*	I	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
	UC, UCE		UC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
	UD, UDP	-	UD	CompData_SC	RespSepData + DataSepResp_SC
			UD	CompData_UC	RespSepData + DataSepResp_UC
	SC	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
	SD[b]	-	SD	CompData_SC	RespSepData + DataSepResp_SC
			UD	CompData_UC	RespSepData + DataSepResp_UC
ReadClean TagOp != *Transfer*	I	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
	UCE	-	UC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
ReadNotSharedDirty	I, UCE[a]	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
			UD	CompData_UD_PD	RespSepData + DataSepResp_UD_PD

Table 4-14 Cache state transitions at the Requester for Read request transactions (continued)

Request type	Cache state				
	Initial		Final	Comp response	Separate responses
	Expected	Others Permitted			
ReadShared	I, UCE[a]	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
			SD	CompData_SD_PD	-
			UD	CompData_UD_PD	RespSepData + DataSepResp_UD_PD
ReadUnique	I, SC	UC, UCE	UC	CompData_UC	RespSepData + DataSepResp_UC
			UD	CompData_UD_PD	RespSepData + DataSepRespUD_PD
	SD	UD, UDP[c]	UD	CompData_UC CompData_UD_PD	RespSepData + DataSepResp_UC, RespSepData + DataSepResp_UD_PD
ReadPreferUnique	I, SC	-	SC	CompData_SC	RespSepData + DataSepResp_SC
			UC	CompData_UC	RespSepData + DataSepResp_UC
			UD	CompData_UD_PD	RespSepData + DataSepResp_UD_PD
	SD[b]	-	SD	CompData_SC	RespSepData + DataSepResp_SC
			UD	CompData_UC CompData_UD_PD	RespSepData + DataSepResp_UC, RespSepData + DataSepResp_UD_PD
MakeReadUnique	See Table 4-17 on page 4-209 and Table 4-18 on page 4-210.				

[a] For ReadOnce∗, ReadNotSharedDirty and ReadShared transactions the Requester with an initial state of UCE must not upgrade the cache line to UDP nor UD while the request is outstanding.

[b] A Requester in initial state of SD that receives a CompData_SC or DataSepResp_SC response must stay in SD state. Similarly, a Home that uses a Snoop filter to track the cached state at the Requester, must not downgrade the state of the cache line in the Snoop filter based on the state in the response to the Requester.

[c] Data received from memory must be dropped if the cache state is UD or SD, or merged if the cache state is UDP. Data received from memory must be the same as the cached data when the cache state is SC or UC.

MakeReadUnique transaction

This section describes the permitted responses, and the cache state transitions at the Requester, for the MakeReadUnique and MakeReadUnique(Excl) transactions. The additional MakeReadUnique(Excl) behavioral requirements are described in *MakeReadUnique(Excl)* on page 6-279.

Permitted responses

Table 4-15 shows the permitted responses in a MakeReadUnique transaction. Table 4-16 on page 4-207 shows the additional responses that are permitted only when the Exclusive bit in the request is set to one. This attribute is indicated by adding the suffix (Excl) to the request.

Some key features of the permitted responses are:

- A response without data, Comp_UD_PD, indicates that the Requester is being passed responsibility for a Dirty cache line.

  This can occur when the Requester holds a SharedClean copy of the cache line and another agent holds a SharedDirty copy. It is permitted for the Home to invalidate the SharedDirty copy, for example using a SnpMakeInvalid transaction, and then pass the responsibility for the Dirty cache line to the Requester that issued the MakeReadUnique transaction.

- A response with a cache state of SC is only permitted in response to an Exclusive version of MakeReadUnique.

  Comp_SC, an example of a response with a cache state of SC, is sent when the Home determines that the Exclusive Store has failed but the snoop filter, or response to a SnpQuery snoop to the Requester, indicates that the Requester still holds a copy of the cache line while there is another shared copy in the system.

  Note

  The above situation can occur with a non-full address PoC exclusive monitor. The monitor bit for an LP can be reset by another LP performing an Exclusive store to a different address location. This store to a different address does not invalidate the cached copy of the address location being used for Exclusive access by the first Requester.

Table 4-15 shows the permitted responses in both the non-Exclusive and Exclusive MakeReadUnique transaction.

Table 4-15 Permitted responses in the MakeReadUnique transaction

Response Received	Description
Comp_UC	Requester retains its copy of the cache line. All other cached copies have been invalidated. Cache line might be Clean or Dirty at the Requester.
Comp_UD_PD	Requester retains its copy of the cache line. Dirty copy of the cache line held elsewhere has been invalidated. Cache line must become Dirty at the Requester.
CompData_UC	Requester lost the cache line while the transaction was in progress. Clean copy of the cache line is given to the Requester. Combined data and completion responses are given.

Table 4-15 Permitted responses in the MakeReadUnique transaction (continued)

Response Received	Description
CompData_UD_PD	Requester lost the cache line while the transaction was in progress. Dirty copy of the cache line is given to the Requester. Combined data and completion responses are given.
RespSepData, DataSepResp_UC	Requester lost the cache line while the transaction was in progress. Clean copy of the cache line is given to the Requester. Separate data and completion responses are given.
RespSepData, DataSepResp_UD_PD	Requester lost the cache line while the transaction was in progress. Dirty copy of the cache line is given to the Requester. Separate data and completion responses given by Home.

Table 4-16 shows the additional permitted responses in the Exclusive MakeReadUnique transaction.

Table 4-16 Additional permitted responses in the MakeReadUnique (Excl) transaction

Response received	Description
Comp_SC	Global Exclusive check fails. Shared cached copy exists in another cache. Requester retains its copy of the cache line.
CompData_SC	Global Exclusive check fails. Shared cached copy exists in another cache. Requester might have lost its copy of the cache line.
RespSepData DataSepResp_SC	Global Exclusive check fails. Shared cached copy exists in another cache. Requester might have lost its copy of the cache line.

Expected snoops

The use of snoops by the Home to invalidate a cache line at the Snoopee in response to a non-Exclusive MakeReadUnique, or an Exclusive MakeReadUnique that passes an Exclusive check, are as follows:

- The Home is expected to use a SnpCleanInvalid snoop:
  - The Home is permitted to use SnpUnique instead of SnpCleanInvalid.
  - The Home is permitted to use SnpUniqueFwd if it determines that the Requester has lost the cached copy of the cache line.
  - The Home is also permitted to use SnpMakeInvalid instead of SnpCleanInvalid if it determines that the Snoopee does not have a Dirty copy of the cache line.

See *Home behavior* on page 6-280 for the types of snoops the Home is expected and permitted to use when an Exclusive MakeReadUnique request fails the Exclusive check.

Cache line state transitions

The final state of the cache line after a MakeReadUnique transaction is dependent on the state of the cache line immediately before the transaction response is received. This can be different from the state of the cache line at the point the transaction is issued.

If the Home does not have a snoop filter, or the snoop filter is imprecise, and the Home cannot determine if the Requester still has a copy of the cache line at the point that the transaction completes, then the Home must assume the cache line is lost at the Requester and provide data with the response.

A Requester that receives a response with data, while still holding the line in SD state, must use its own copy of the cache line, rather than the copy returned with the response.

Note

A Requester that receives a response with data, while still holding the line in SD state implies there is no snoop filter present or that the snoop filter is imprecise. In this case, it is possible that the data returned with the response is Stale.

If the Requester is aware that there is no snoop filter, then it can use a CleanUnique transaction instead of MakeReadUnique to avoid an unnecessary memory read when the cache line is not cached at any other agent.

Note

Use of the CleanUnique transaction can avoid an unnecessary memory read in the absence of a snoop filter. The unnecessary memory read occurs when the cache line remains cached at the Requester, but a snoop filter is not present in the system or a SnpQuery snoop is not used and a cached copy is not available from another agent in the system. However, using a CleanUnique transaction will result in the Requester needing to issue another transaction in the case where the cache line is lost due to a snoop while the transaction is in progress.

Table 4-15 on page 4-206 shows the state transitions and responses that are applicable to both non-Exclusive and Exclusive versions of MakeReadUnique. Table 4-16 on page 4-207 shows the additional state transitions and responses that apply only to the Exclusive version of the request.

The response rules are:

- The cache state in the response to a non-Exclusive MakeReadUnique must be Unique.
- The cache state in the response to an Exclusive MakeReadUnique is permitted to be Unique or Shared.
- The cache state in the response to an Exclusive and to a Non-exclusive MakeReadUnique must not include Shared Dirty.
- For each permitted combined completion and data response, a corresponding response with separate completion and data is permitted.

Table 4-17 on page 4-209 and Table 4-18 on page 4-210 include columns for:

- The initial cache state.
- The cache state immediately before the transaction response is received.
- The final state for each of the possible response combinations.

The handling of a MakeReadUnique transaction at the Home depends on the availability or not of a precise snoop filter. Table 4-17 on page 4-209 and Table 4-18 on page 4-210 also include the responses that are permitted with and without a precise snoop filter:

- The table column labeled: SF Precise, covers the cases when the precise state of the cache line at the Requester, at the response time, is known. The Home obtains the knowledge of precise state from a snoop filter, or by sending a SnpQuery snoop, or in some other IMPLEMENTATION DEFINED manner.
- The column labeled: SF imprecise or Absent, covers the cases when the precise state of the cache line at the Requester, at the response time, is not known. This also includes the case when there is no snoop filter, or the Home might decide to ignore the available precise information, or not attempt to obtain the information.

Table 4-17 Cache state transitions at the Requester for the MakeReadUnique request (non-Excl and Excl)

Initial State	State at time of response	Final State	Comp Resp	SF Precise	SF Imprecise or Absent	Notes
SD	SD	UD	Comp_UC	Y	-	
			CompData_UC	-	Y	Returned data might be stale
			RespSepData, DataSepResp_UC	-	Y	
SC, SD	SC	UC	Comp_UC	Y	-	
			CompData_UC	-	Y	Returned data might be stale
			RespSepData, DataSepResp_UC	-	Y	
		UD	Comp_UD_PD	Y	-	
			CompData_UD_PD	-	Y	Data in response is identical to the Requester copy
			RespSepData, DataSepResp_UD_PD	-	Y	
	I	UC	CompData_UC	Y	Y	Line lost to the snoop
			RespSepData, DataSepResp_UC	Y	Y	
		UD	CompData_UD_PD	Y	Y	
			RespSepData, DataSepResp_UD_PD	Y	Y	
I,UC,UD	Not permitted					

Table 4-18 Additional cache state transitions at the Requester for the MakeReadUnique(Excl) request

Initial State	State at time of response	Final State	Comp Resp	SF Precise	SF Imprecise or Absent
SC, SD	SC	SC	Comp_SC	Y	-
	I	SC	CompData_SC	Y	-
			RespSepData DataSepResp_SC	Y	-
	SC, I	SC	CompData_SC	-	Y
			RespSepData DataSepResp_SC	-	Y
SD	SD	SD	Comp_SC	Y	-
			CompData_SC	-	Y
			RespSepData DataSepResp_SC	-	Y

See *MakeReadUnique(Excl)* on page 6-279 for the additional MakeReadUnique(Excl) behavioral requirements.

4.7.2 Dataless request transactions

Table 4-19 shows the cache state transitions at the Requester, and the completion responses, for Dataless request transactions.

Table 4-19 Cache state transitions at the Requester for Dataless request transactions

Request type	Cache state			Comp Response
	Initial		Final	
	Expected	Others permitted		
CleanUnique	I	UC, UCE	UCE	Comp_UC
	SC	UC	UC	Comp_UC
	SD	UD	UD	Comp_UC
MakeUnique	I, SC, SD	UC, UCE	UD	Comp_UC
Evict	I	-	I	Comp_I
StashOnceUnique	I	-	I	Comp
StashOnceSepUnique	I	-	I	Comp + StashDone or CompStashDone
StashOnceShared	I	-	I	Comp
StashOnceSepShared	I	-	I	Comp + StashDone or CompStashDone
CleanShared CleanSharedPersist	I, SC, UC	-	No Change	Comp_UC
				Comp_SC
				Comp_I
CleanSharedPersistSep	I, SC,UC	-	No Change	Comp_UC + Persist or CompPersist_UC
				Comp_SC + Persist or CompPersist_SC
				Comp_I + Persist or CompPersist_I
CleanInvalid	I	-	I	Comp_I
MakeInvalid	I	-	I	Comp_I

Before a CleanInvalid, MakeInvalid or Evict transaction it is permitted for the cache state to be UC, UCE or SC. However, it is required that the cache state transitions to the I state before the transaction is issued. Therefore Table 4-19 shows I state as the only initial state.

4.7.3 Write request transactions

Table 4-20 shows the cache state transitions at the Requester, the WriteData response, and the combined or separate completion and DBIDResp response for Write and corresponding Combined Write request transactions. Combined Write request transactions are not listed in Table 4-20. See *Combined Write requests* on page 4-176.

Table 4-20 Requester cache state transitions for Write request transactions

Request Type	Cache state at Requester			WriteData response	Comp response
	Initial	Before WriteData response[a]	Final		
WriteNoSnpPtl	I	-	I	NCBWrData or NCBWrDataCompAck or WriteDataCancel	DBIDResp* + Comp or CompDBIDResp
WriteNoSnpFull	I	-	I	NCBWrData or NCBWrDataCompAck	DBIDResp* + Comp or CompDBIDResp
WriteNoSnpZero	I	-	I	None	DBIDResp* + Comp or CompDBIDResp
WriteUniquePtl WriteUniquePtlStash	I	I	I	NCBWrData or WriteDataCancel or NCBWrDataCompAck	DBIDResp* + Comp or CompDBIDResp
WriteUniqueZero	I	I	I	None	DBIDResp* + Comp or CompDBIDResp
WriteUniqueFull WriteUniqueFullStash	I	I	I	NCBWrData or NCBWrDataCompAck	DBIDResp* + Comp or CompDBIDResp
WriteBackFull	UD	UD	I	CBWrData_UD_PD	CompDBIDResp
		UC	I	CBWrData_UC	CompDBIDResp
	UD, SD	SD	I	CBWrData_SD_PD	CompDBIDResp
		SC	I	CBWrData_SC	CompDBIDResp
		I	I	CBWrData_I	CompDBIDResp
WriteBackPtl	UDP	UDP	I	CBWrData_UD_PD	CompDBIDResp
		I	I	CBWrData_I	CompDBIDResp
WriteCleanFull	UD	UD	UC	CBWrData_UD_PD	CompDBIDResp
		UC	UC	CBWrData_UC	CompDBIDResp
	UD, SD	SD	SC	CBWrData_SD_PD	CompDBIDResp
		SC	SC	CBWrData_SC	CompDBIDResp
		I	I	CBWrData_I	CompDBIDResp
WriteEvictFull	UC	UC	I	CBWrData_UC	CompDBIDResp
		SC	I	CBWrData_SC	CompDBIDResp
		I	I	CBWrData_I	CompDBIDResp

Table 4-20 Requester cache state transitions for Write request transactions (continued)

Request Type	Cache state at Requester			WriteData response	Comp response
	Initial	Before WriteData response[a]	Final		
WriteEvictOrEvict	UC	UC[b]	I	CBWrData_UC	CompDBIDResp
			I	None	Comp[c]
	UC, SC	SC	I	CBWrData_SC	CompDBIDResp
			I	None	Comp[c]
		I	I	CBWrData_I	CompDBIDResp
			I	None	Comp[c]

a. A snoop might be received while a write is pending and results in a cache line state change before the WriteData response.

b. Once the request is sent the Requester is permitted to keep the cache state in UC but must not modify the cache line.

c. Comp is sent if the Home decides not to request data.

When the Requester receives a RetryAck response whose data is invalidated by a snoop, the Requester is permitted to send the Write request with AllowRetry set to zero, knowing that the subsequent Data response would be CopyBackWrData_I

Note

After completion of a WriteClean transaction, it is possible for the cache line in a Unique state to immediately transition to a Dirty state.

4.7.4 Atomic transactions

Table 4-21 shows the cache state transitions at the Requester, and the completion and response for Atomic transactions.

Table 4-21 Requester cache state transitions for Atomic request transactions

Atomic request	Cache state			WriteData response	Comp response
	Initial		Final		
	Expected	Others permitted			
AtomicStore	I, SC UCE, SD	UC, UD, UDP	I	NCBWrData	DBIDResp* + Comp_I or CompDBIDResp
AtomicLoad	I, SC UCE, SD	UC, UD, UDP	I	NCBWrData	DBIDResp* + CompData_I
AtomicSwap	I, SC UCE, SD	UC, UD, UDP	I	NCBWrData	DBIDResp* + CompData_I
AtomicCompare	I, SC UCE, SD	UC, UD, UDP	I	NCBWrData	DBIDResp* + CompData_I

4.7.5 Other request transactions

DVMOp and PrefetchTgt requests do not have any cache state transitions associated with them.

4.8 Cache state transitions at a Snoopee

This section specifies the cache state transitions and completion responses for the following Snoop transactions:

- *Non-Forwarding and Non-stash Snoop transactions*.
- *Stash snoop transactions* on page 4-220.
- *Forwarding Snoop transactions* on page 4-222.

A Snoopee, an RN-F that receives a snoop, performs two actions. One action is a state change of the cached line and the second action is sending a response message either to the Home, or to both the Home and the Requester.

The cache state change depends on the snoop type, the initial state of the cache line and the value of DoNotGoToSD in the snoop. See *Do not transition to SD* on page 4-233.

The Snoopee must send a response to Home either with Data or without Data. In addition, for Forwarding snoops the Snoopee can also forward a Data response to the Requester.

The type of response sent is determined by the snoop type, initial cache state, cache state change, and the value of RetToSrc. See *Returning Data with Snoop response* on page 4-232.

4.8.1 Non-Forwarding and Non-stash Snoop transactions

The non-Forwarding and Non-stash Snoop transactions are:

- SnpOnce.
- SnpClean, SnpShared, and SnpNotSharedDirty.
- SnpUnique, SnpPreferUnique.
- SnpCleanShared, SnpCleanInvalid, and SnpMakeInvalid.
- SnpQuery.

SnpOnce

Table 4-22 on page 4-215 shows for SnpOnce, the initial, expected final, and other permitted final cache states at the snooped Requester, the RetToSrc field value, and the valid completion response from a snooped RN-F.

Table 4-22 Cache state transitions, RetToSrc value, and valid completion responses

Snoop request type	Initial cache state	Final cache state		RetToSrc[a]	Snoop response
		Expected	Others permitted		
SnpOnce	I	I	-	X	SnpResp_I
	UC	UC	I, SC	0	SnpResp_UC
					SnpRespData_UC
				1	SnpResp_UC
					SnpRespData_UC
		SC	I	0	SnpResp_SC
					SnpRespData_SC
				1	SnpResp_SC
					SnpRespData_SC
		I	-	0	SnpResp_I
					SnpRespData_I
				1	SnpResp_I
					SnpRespData_I
	UCE	UCE	I	X	SnpResp_UC
		I	-	X	SnpResp_I
	UD	UD	SD	X	SnpRespData_UD
		SD	-	X	SnpRespData_SD
		SC	I	X	SnpRespData_SC_PD
		I	-	X	SnpRespData_I_PD
	UDP	I	-	X	SnpRespDataPtl_I_PD
		UDP	-	X	SnpRespDataPtl_UD
	SC	SC	I	0	SnpResp_SC
				1	SnpRespData_SC
		I	-	0	SnpResp_I
				1	SnpRespData_I
	SD	SD	-	X	SnpRespData_SD
		SC	I	X	SnpRespData_SC_PD
		I	-	X	SnpRespData_I_PD

a. X indicates that the protocol requirements apply for both states of RetToSrc.

SnpClean, SnpShared, SnpNotSharedDirty and SnpPreferUnique

Table 4-23 shows for SnpClean, SnpShared, SnpNotSharedDirty and SnpPreferUnique the initial, expected final, and other permitted final cache states at the snooped Requester, the RetToSrc field value, and the valid completion response from a snooped RN-F. Table 4-23 must be used to determine SnpPreferUnique cache state transitions when the Snoopee is in the middle of executing an Exclusive access sequence. A Snoopee that is not in the middle of executing an Exclusive access is permitted, but not expected, to use Table 4-23 to determine SnpPreferUnique cache state transitions.

Table 4-23 Cache state transitions, RetToSrc value, and valid completion responses

Snoop request type	Initial cache state	Final cache state		RetToSrc[a]	Snoop response
		Expected	Others permitted		
SnpClean, SnpShared, SnpNotSharedDirty, SnpPreferUnique[b]	I	I	-	X	SnpResp_I
	UC	SC	I	0	SnpResp_SC
					SnpRespData_SC
				1	SnpResp_SC
					SnpRespData_SC
		I	-	0	SnpResp_I
					SnpRespData_I
				1	SnpResp_I
					SnpRespData_I
	UCE	I	-	X	SnpResp_I
	UD	SD[c]	-	X	SnpRespData_SD
		SC	I	X	SnpRespData_SC_PD
		I	-	X	SnpRespData_I_PD
	UDP	I	-	X	SnpRespDataPtl_I_PD
	SC	SC	I	0	SnpResp_SC
				1	SnpRespData_SC
		I	-	0	SnpResp_I
				1	SnpRespData_I
	SD	SD[c]	-	X	SnpRespData_SD
		SC	I	X	SnpRespData_SC_PD
		I	-	X	SnpRespData_I_PD

a. X indicates that the protocol requirements apply for both states of RetToSrc.

b. Only applicable when the Snoopee is in the middle of executing an Exclusive sequence.

c. This state transition is not permitted if DoNotGoToSD is set.

SnpUnique and SnpPreferUnique

Table 4-24 shows for SnpUnique and SnpPreferUnique the initial, expected final, and other permitted final cache states at the snooped Requester, the RetToSrc field value, and the valid completion response from a snooped RN-F for SnpUnique. Table 4-24 is expected to be used to determine SnpPreferUnique cache state transitions when the Snoopee is not in the middle of executing an Exclusive access sequence.

Table 4-24 Cache state transitions, RetToSrc value, and valid completion responses

Snoop request type	Initial cache state	Final cache state		RetToSrc[a]	Snoop response
		Expected	Others permitted		
SnpUnique, SnpPreferUnique[b]	I	I	-	X	SnpResp_I
	UC	I	-	0	SnpResp_I
					SnpRespData_I
				1	SnpResp_I
					SnpRespData_I
	UCE	I	-	X	SnpResp_I
	UD	I	-	X	SnpRespData_I_PD
	UDP	I	-	X	SnpRespDataPtl_I_PD
	SC	I	-	0	SnpResp_I
				1	SnpRespData_I
	SD	I	-	X	SnpRespData_I_PD

[a] X indicates that the protocol requirements apply for both states of RetToSrc.

[b] Only applicable when the Snoopee is not in the middle of executing an Exclusive sequence.

SnpCleanShared, SnpCleanInvalid and SnpMakeInvalid

Table 4-25 shows for SnpCleanShared, SnpCleanInvalid, and SnpMakeInvalid the initial, expected final, and other permitted final cache states at the snooped Requester, the RetToSrc field value, and the valid completion response from a snooped RN-F.

Table 4-25 Cache state transitions, RetToSrc value, and valid completion responses

Snoop request type	Initial cache state	Final cache state		RetToSrc	Snoop response
		Expected	Others permitted		
SnpCleanShared	I	I	-	0	SnpResp_I
	UC	UC	I, SC	0	SnpResp_UC
		SC	I	0	SnpResp_SC
		I	-	0	SnpResp_I
	UCE	I	-	0	SnpResp_I
	UD	UC	I, SC	0	SnpRespData_UC_PD
		SC	I	0	SnpRespData_SC_PD
		I	-	0	SnpRespData_I_PD
	UDP	I	-	0	SnpRespDataPtl_I_PD
	SC	SC	I	0	SnpResp_SC
		I	-	0	SnpResp_I
	SD	SC	I	0	SnpRespData_SC_PD
		I	-	0	SnpRespData_I_PD
SnpCleanInvalid	I	I	-	0	SnpResp_I
	UC	I	-	0	SnpResp_I
	UCE	I	-	0	SnpResp_I
	UD	I	-	0	SnpRespData_I_PD
	UDP	I	-	0	SnpRespDataPtl_I_PD
	SC	I	-	0	SnpResp_I
	SD	I	-	0	SnpRespData_I_PD
SnpMakeInvalid	I	I	-	0	SnpResp_I
	UC	I	-	0	SnpResp_I
	UCE	I	-	0	SnpResp_I
	UD	I	-	0	SnpResp_I
	UDP	I	-	0	SnpResp_I
	SC	I	-	0	SnpResp_I
	SD	I	-	0	SnpResp_I

SnpQuery

Table 4-26 shows for SnpQuery the initial, expected final, and other permitted final cache states at the snooped Requester, the RetToSrc field value, and the valid completion response from a snooped RN-F.

Table 4-26 Cache state transitions, RetToSrc value, and valid completion responses

Snoop request type	Initial cache state	Final cache state		RetToSrc	Snoop response
		Expected	Others permitted		
SnpQuery	I	I	-	0	SnpResp_I
	UC	UC	-	0	SnpResp_UC
	UCE	UCE	-	0	SnpResp_UC
	UD	UD	-	0	SnpResp_UD
	UDP	UDP	-	0	SnpResp_UD
	SC	SC	-	0	SnpResp_SC
	SD	SD	-	0	SnpResp_SD

4.8.2 Stash snoop transactions

The following sub-sections show the permitted responses for the Stash type snoops.

SnpUniqueStash and SnpMakeInvalidStash

The permitted responses to SnpUniqueStash and SnpMakeInvalidStash are the same as the responses to SnpUnique and SnpMakeInvalid respectively.

The RetToSrc bit value must not be set to 1 in SnpUniqueStash and SnpMakeInvalidStash.

Any Snoop response to SnpUniqueStash and SnpMakeInvalidStash can include a Data Pull request. A Data Pull request in a Snoop response to SnpUniqueStash and SnpMakeInvalidStash must be treated as a ReadUnique request.

Table 4-27 shows the Snoopee cache state transitions and required Snoop responses. The Snoop responses do not include the Data Pull options. Data Pull is permitted with any Snoop response.

Table 4-27 Snoop response to SnpUniqueStash and SnpMakeInvalidStash

Snoop request type	Cache state			RetToSrc	Snoop response
	Initial	Final			
		Expected	Others permitted		
SnpUniqueStash	I	I	-	0	SnpResp_I
	UC	I	-	0	SnpRespData_I
					SnpResp_I
	UCE	I	-	0	SnpResp_I
	UD	I	-	0	SnpRespData_I_PD
	UDP	I	-	0	SnpRespDataPtl_I_PD
	SC	I	-	0	SnpResp_I
					SnpRespData_I
	SD	I	-	0	SnpRespData_I_PD
SnpMakeInvalidStash	Any	I	-	0	SnpResp_I

SnpStashUnique and SnpStashShared

For SnpStashUnique and SnpStashShared the Snoopee must not change cache state.

The Snoopee is permitted to not perform a cache lookup before responding, in which case the Snoop response must be SnpResp_I.

The Snoopee is permitted to include the precise cache state in the response.

A Snoop response can include Data Pull only if the cache state in the response is precise.

The Snoopee can include a Data Pull in response to a SnpStashUnique only if the cache-data is not present, or present in a Shared state. The Snoopee can include Data Pull in response to SnpStashShared only if the cache cache-data is not present.

A Data Pull request in a Snoop response to SnpStashUnique must be treated as a ReadUnique request. A Data Pull request in a Snoop response to SnpStashShared must be treated as a ReadNotSharedDirty request.

The inclusion of Data Pull in the Snoop response must ensure that the initial state must not violate the initial state conditions permitted for the corresponding independent Read requests. See *Read transactions* on page 4-165.

Table 4-28 shows the Snoopee cache state transitions, the required Snoop responses, and Data Pull options for SnpStashUnique.

Table 4-28 Snoop response to SnpStashUnique

Snoop request type	Cache state			RetToSrc	Snoop response
	Initial	Final			
		Expected	Others permitted		
SnpStashUnique	I	I	-	0	SnpResp_I
					SnpResp_I_Read
	UC	UC	-	0	SnpResp_UC
					SnpResp_I
	UCE	UCE	-	0	SnpResp_UC
					SnpResp_UC_Read
					SnpResp_I
	UD	UD	-	0	SnpResp_UD
					SnpResp_I
	UDP	UDP	-	0	SnpResp_UD
					SnpResp_I
	SC	SC	-	0	SnpResp_SC
					SnpResp_SC_Read
					SnpResp_I
	SD	SD	-	0	SnpResp_SD
					SnpResp_SD_Read
					SnpResp_I

Table 4-29 shows the Snoopee cache state transitions, the required Snoop responses, and Data Pull options for SnpStashShared.

Table 4-29 Snoop response to SnpStashShared

Snoop request type	Cache state			RetToSrc	Snoop response
	Initial	Final			
		Expected	Others permitted		
SnpStashShared	I	I	-	0	SnpResp_I_Read
					SnpResp_I
	UC	UC	-	0	SnpResp_UC
					SnpResp_I
	UCE	UCE	-	0	SnpResp_UC
					SnpResp_UC_Read
					SnpResp_I
	UD	UD	-	0	SnpResp_UD
					SnpResp_I
	UDP	UDP	-	0	SnpResp_UD
					SnpResp_I
	SC	SC	-	0	SnpResp_SC
					SnpResp_I
	SD	SD	-	0	SnpResp_SD
					SnpResp_I

4.8.3 Forwarding Snoop transactions

Forwarding (Fwd) type snoops are used by Home to support DCT. The rules, common to all Forwarding snoops at the Snoopee are:

- Expected, but not required, to forward a copy of the cache line to the Requester if the cache line is in one of the following states:
  - UD.
  - UC.
  - SD.
  - SC.
- The Snoopee is permitted, but not expected, to convert the Snoop to its corresponding non-Forwarding type.
- Must not forward data in Unique state in response to a Non-invalidating type snoop.
- Snoopee receiving a Snoop request with the DoNotGoToSD bit set, except when the Snoop is SnpOnceFwd, must not transition to SD, even if the coherency conditions permit it.

- In certain cases, based on the Snoop type, the state of the cache line at the Snoopee, and the RetToSrc value in the Snoop request, the Snoopee forwards a copy to Home along with a copy to the Requester. Forwarding snoops must not be used if the original request requests tags.

  If tags are available, Clean tags are permitted to be forwarded to the Requester along with the data.

- Home is not permitted to send a Forwarding type snoop for:
  - Atomic transactions.
  - Passing Exclusive read transactions.

    Note

    Exclusive read transactions that fail due to Non-exclusive support for the address range being accessed are treated as corresponding Non-exclusive reads, Home can therefore use Forwarding type snoops in these cases.

For the rules that are specific to a particular Fwd type snoop see the individual sub-sections in *Forwarding Snoop transactions* on page 4-222.

The tables in the following individual sub-sections show the Snoopee state transitions and the corresponding responses to the Requester and the Home.

The first column in the tables shows the initial cache state, and is the combined data and tag state. Table 4-30 shows the combined state used and the corresponding data and possible tag state combinations.

Table 4-30 Combined state and the corresponding data and tag states

Combined state	Data state	Tag state
I	I	I
UC	UC	I Clean
UCE	UCE	I
UD	UD	I Clean Dirty
UDP	UDP	I
SC	SC	I Clean
SD	SD	I Clean Dirty

The last three columns in the tables correspond to the TagOp value in the Data responses to Home. They are organized, based on the initial state of the tags:

Dirty Two columns:

- First column: Indicates if the transition itself is permitted or not. This column is only relevant when the tag initial state is Dirty.

  Table 4-31 shows the conventions used, P and NP are only relevant when the data initial state is UD or SD and the tag initial state is Dirty.

Table 4-31 Key to table conventions

Symbol	Description
P	The transition is permitted
NP	The transition is not permitted
-	TagOp is not applicable

- Second column: The response TagOp value when the transition is permitted. The TagOp value in the response:
  - Must be *Update* if the response state includes Pass Dirty.
  - Must be *Transfer* if the response does not include Pass Dirty.

Invalid, Clean

One column:

- TagOp in the response can be:
  - *Invalid* if the initial tag state is Invalid or Clean.
  - *Transfer* when the initial tag state is Clean.

In Snoop responses to Home without data, TagOp is inapplicable.

SnpOnceFwd

The rules, in addition to the common rules, to be followed by a Snoopee that receives a SnpOnceFwd are:

- Snoopee must forward the cache line in I state.
  - As a consequence, the Snoopee must not forward Pass Dirty to the Requester.
- Snoopee must return data to Home only when Dirty state is changed to Clean or Invalid.
- RetToSrc bit in the snoop must be set to zero.
- Snoopee can ignore the DoNotGoToSD value in the snoop.

Table 4-32 on page 4-225 shows the Snoopee cache state transitions and the required Snoop responses.

See Table 4-31 for the symbol key.

Table 4-32 SnpOnceFwd Snoopee state transitions with Clean and Dirty tags

Snoopee cache state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other permitted				Dirty		I. Clean
I	I	-	0	No Fwd	SnpResp_I	-	-	-
UC	UC	-	0	CompData_I	SnpResp_UC_Fwded_I	-	-	-
	SC	I	0	CompData_I	SnpResp_SC_Fwded_I	-	-	-
	I	-	0	CompData_I	SnpResp_I_Fwded_I	-	-	-
UCE	UCE	-	0	No Fwd	SnpResp_UC	-	-	-
	I	-	0	No Fwd	SnpResp_I	-	-	-
UD	UD	-	0	CompData_I	SnpResp_UD_Fwded_I	P[a]	-	-
	SD	-	0	CompData_I	SnpResp_SD_Fwded_I	P[a]	-	-
	SC	I	0	CompData_I	SnpRespData_SC_PD_Fwded_I	P	Update[b]	I, Transfer[c]
	I	-	0	CompData_I	SnpRespData_I_PD_Fwded_I	P	Update	I, Transfer
UDP	UDP	-	0	No Fwd	SnpRespDataPtl_UD	-	-	I
	I	-	0	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	SC	I	0	No Fwd	SnpResp_SC	-	-	-
				CompData_I	SnpResp_SC_Fwded_I	-	-	-
	I	-	0	No Fwd	SnpResp_I	-	-	-
				CompData_I	SnpResp_I_Fwded_I	-	-	-
SD	SD	-	0	CompData_I	SnpResp_SD_Fwded_I	P[a]	-	-
	SC	I	0	CompData_I	SnpRespData_SC_PD_Fwded_I	P	Update	I. Transfer
	I	-	0	CompData_I	SnpRespData_I_PD_Fwded_I	P	Update	I. Transfer

a. This transition is permitted, even though the Snoop response does not include data and the tag state is Dirty, because Dirty tags are retained by the Snoopee.

b. This transition from tag state of Dirty is permitted because a Dirty copy of tags and data are passed to the Home.

c. This transaction from an initial state of Dirty data, and Invalid or Clean tags, includes the Snoop response of data with Clean tags when available.

SnpCleanFwd, SnpNotSharedDirtyFwd

The rules, in addition to the common rules, to be followed by a Snoopee that receives a SnpCleanFwd or a SnpNotSharedDirtyFwd are:

- Snoopee must forward the cache line in SC state.
- Snoopee must transition to either SD, SC or I state.
- For behavior related to the RetToSrc bit see *Returning Data with Snoop response* on page 4-232.

Table 4-33 shows the Snoopee cache state transitions and the required Snoop responses.

See Table 4-31 on page 4-224 for the symbol key.

Table 4-33 SnpCleanFwd and SnpNotSharedDirtyFwd Snoopee state transitions with Clean and Dirty tags

Snoopee cache (data+tag) state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other permitted				Dirty		I, Clean
I	I	-	X[a]	No Fwd	SnpResp_I	-	-	-
UC	SC	I	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
	I	-	0	CompData_SC	SnpResp_I_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_I_Fwded_SC	-	-	I, Transfer
UCE	I	-	X[a]	No Fwd	SnpResp_I	-	-	-
UD	SD[b]	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I, Transfer
	SC	I	X[a]	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer
	I	-	X[a]	CompData_SC	SnpRespData_I_PD_Fwded_SC	P	Update	I, Transfer
UDP	I	-	X[a]	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	SC	I	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
	I	-	0	CompData_SC	SnpResp_I_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_I_Fwded_SC	-	-	I, Transfer
SD	SD[b]	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I, Transfer
	SC	I	X[a]	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer
	I	-	X[a]	CompData_SC	SnpRespData_I_PD_Fwded_SC	P	Update	I, Transfer

a. The protocol requirements apply for both states of RetToSrc.

b. This state transition is not permitted if DoNotGoToSD is asserted.

SnpSharedFwd

The rules, in addition to the common rules, to be followed by a Snoopee that receives a SnpSharedFwd are:

- Snoopee is permitted to forward the cache line in either SD or SC state.
- Snoopee must transition to either SD, SC or I state.
- For behavior related to the RetToSrc bit see *Returning Data with Snoop response* on page 4-232.

Table 4-34 shows the Snoopee cache state transition and the required Snoop responses.

See Table 4-31 on page 4-224 for the symbol key.

Table 4-34 SnpSharedFwd Snoopee state transitions with Clean and Dirty tags

Snoopee cache state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other permitted				Dirty		I, Clean
I	I	-	X[a]	No Fwd	SnpResp_I	-	-	-
UC	SC	I	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
	I	-	0	CompData_SC	SnpResp_I_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_I_Fwded_SC	-	-	I, Transfer
UCE	I	-	X[a]	No Fwd	SnpResp_I	-	-	-
UD	SD[b]	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I, Transfer
	SC	I	0	CompData_SD_PD	SnpResp_SC_Fwded_SD_PD	NP	-	-
			1	CompData_SD_PD	SnpRespData_SC_Fwded_SD_PD	NP	-	I, Transfer
			X[a]	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer
	I	-	0	CompData_SD_PD	SnpResp_I_Fwded_SD_PD	NP	-	-
			1	CompData_SD_PD	SnpRespData_I_Fwded_SD_PD	NP	-	I, Transfer
			X[a]	CompData_SC	SnpRespData_I_PD_Fwded_SC	P	Update	I, Transfer
UDP	I	-	X[a]	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	SC	I	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
	I	-	0	CompData_SC	SnpResp_I_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_I_Fwded_SC	-	-	I, Transfer

Table 4-34 SnpSharedFwd Snoopee state transitions with Clean and Dirty tags (continued)

Snoopee cache state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other permitted				Dirty		I, Clean
SD	SD[b]	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I. Transfer
	SC	I	0	CompData_SD_PD	SnpResp_SC_Fwded_SD_PD	NP	-	-
			1	CompData_SD_PD	SnpRespData_SC_Fwded_SD_PD	NP	-	I, Transfer
			X[a]	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer
	I	-	0	CompData_SD_PD	SnpResp_I_Fwded_SD_PD	NP	-	-
			1	CompData_SD_PD	SnpRespData_I_Fwded_SD_PD	NP	-	I, Transfer
			X[a]	CompData_SC	SnpRespData_I_PD_Fwded_SC	P	Update	I, Transfer

a. The protocol requirements apply for both states of RetToSrc.
b. This state transition is not permitted if DoNotGoToSD is asserted.

SnpUniqueFwd

Use of the SnpUniqueFwd snoop is only permitted if the cache line is cached at a single RN-F:

- Home is permitted to send the SnpUniqueFwd snoop to an RN-F in Shared state if Home determines that the invalidating snoop needs to be sent to only one cache.

The rules, in addition to the common rules, to be followed by a Snoopee that receives a SnpUniqueFwd are:

- Snoopee must forward the cache line in Unique state.
- Snoopee that has the cache line in Dirty state must Pass Dirty to the Requester not to Home.
- Snoopee must transition to I state.
- Snoopee must not return data to Home.

RetToSrc bit in the snoop must be set to zero.

Table 4-35 on page 4-229 shows the Snoopee cache state transitions and the required Snoop responses.

See Table 4-31 on page 4-224 for the symbol key.

Table 4-35 SnpUniqueFwd Snoopee state transitions with Clean and Dirty tags

Snoopee cache (data+tags} state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other permitted				Dirty		I, Clean
I	I	-	0	No Fwd	SnpResp_I	-	-	-
UC	I	-	0	CompData_UC	SnpResp_I_Fwded_UC	-	-	-
UCE	I	-	0	No Fwd	SnpResp_I	-	-	-
UD	I	-	0	CompData_UD_PD	SnpResp_I_Fwded_UD_PD	NP[a]	-	-
				No Fwd	SnpRespData_I_PD	P	Update	I, Transfer
UDP	I	-	0	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	I	-	0	CompData_UC	SnpResp_I_Fwded_UC	-	-	-
SD	I	-	0	CompData_UD_PD	SnpResp_I_Fwded_UD_PD	NP[a]	-	-
				No Fwd	SnpRespData_I_PD	P	Update	I, Transfer

a. This transition is not permitted because Dirty tags are lost. The Dirty tags are lost because the Snoop response to the Home does not include Data with which to pass the Dirty tags to the Home.

SnpPreferUnique and SnpPreferUniqueFwd

Use of the SnpPreferUniqueFwd snoop is only permitted if the cache line is cached at a single RN-F.

The rules, in addition to the common rules, to be followed by a Snoopee that receives a SnpPreferUniqueFwd and does not treat it as the non-Forwarding snoop are:

- When the Snoopee is in the process of executing an Exclusive access sequence, using the same address:
  - Snoopee must forward the cache line in SC state.
  - Snoopee must transition to either SD or SC state.
  - Snoopee must not transition to I state.
  - For behavior related to the RetToSrc bit see *Returning Data with Snoop response* on page 4-232 related to SnpNotSharedDirtyFwd.
  - A Snoopee that is not in the middle of executing an Exclusive access sequence is permitted, but not expected, to treat the snoop as a non-invalidating snoop.
- When the Snoopee is not in the process of executing an Exclusive access sequence, using the same address, and treats the snoop as an invalidating snoop:
  - Snoopee must forward the cache line in Unique state.
  - Snoopee that has the cache line in Dirty state must Pass Dirty to the Requester not to Home.
  - Snoopee must transition to I state.
  - Snoopee must not return data to Home.
  - RetToSrc bit value in the snoop is ignored and treated as zero.

Table 4-36 and Table 4-37 on page 4-231 show the Snoopee cache state transitions and the required Snoop responses.

See Table 4-31 on page 4-224 for the symbol key.

Table 4-36 State transitions for SnpPreferUniqueFwd when Snoopee is executing an exclusive sequence

Snoopee cache (Data+tag) state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other Permitted				Dirty		I, Clean
I	I	-	X	No Fwd	SnpResp_I	-	-	-
UC	SC	-	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
UCE	I	-	X	No Fwd	SnpResp_I	-	-	-
UD	SD*	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I, Transfer
	SC	-	X	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer
UDP	I	-	X	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	SC	-	0	CompData_SC	SnpResp_SC_Fwded_SC	-	-	-
			1	CompData_SC	SnpRespData_SC_Fwded_SC	-	-	I, Transfer
SD	SD*	-	0	CompData_SC	SnpResp_SD_Fwded_SC	P	-	-
			1	CompData_SC	SnpRespData_SD_Fwded_SC	P	Transfer	I, Transfer
	SC	-	X	CompData_SC	SnpRespData_SC_PD_Fwded_SC	P	Update	I, Transfer

Table 4-37 shows the expected and permitted Snoopee cache state transitions and responses to the SnpPreferUniqueFwd Snoop request when the Snoopee is not executing an exclusive sequence.

See Table 4-31 on page 4-224 for the symbol key.

Table 4-37 State transitions for SnpPreferUniqueFwd when Snoopee is not executing an exclusive sequence

Snoopee cache (Data+tag) state			RetToSrc	Response to		TagOp value in response to Home		
Initial	Final			Requester	Home	Initial tag state		
	Expected	Other Permitted				Dirty		I, Clean
I	I	-	X	No Fwd	SnpResp_I	-	-	-
UC	I	-	X	CompData_UC	SnpResp_I_Fwded_UC	-	-	-
UCE	I	-	X	No Fwd	SnpResp_I	-	-	-
UD	I	-	X	CompData_UD_PD	SnpResp_I_Fwded_UD_PD	NP	-	-
			X	No Fwd	SnpRespData_I_PD	P	Update	I, Transfer
UDP	I	-	X	No Fwd	SnpRespDataPtl_I_PD	-	-	I
SC	I	-	X	CompData_UC	SnpResp_I_Fwded_UC	-	-	-
SD	I	-	X	CompData_UD_PD	SnpResp_I_Fwded_UD_PD	NP		
			X	No Fwd	SnpRespData_I_PD	P	Update	I, Transfer

4.9 Returning Data with Snoop response

The rules for returning a copy of the cache line with the Snoop response are detailed below:

For non-Forwarding snoops, except SnpMakeInvalid, the rules for returning a copy of the cache line to the Home are:

- Irrespective of the value of RetToSrc, must return a copy if the cache line is Dirty.
- Irrespective of the value of RetToSrc, optionally can return a copy if the cache line is Unique Clean.
- If the RetToSrc value is 1, must return a copy if the cache line is Shared Clean and the Snoopee retains a copy of the cache line.
- If the RetToSrc value is 0, must not return a copy if the cache line is Shared Clean.

For Forwarding snoops, the rules for returning a copy of the cache line to the Home are:

- Irrespective of the value of RetToSrc, must return a copy if a Dirty cache line cannot be forwarded or kept.
- If the RetToSrc value is 1, must return a copy if the cache line is Dirty or Clean.
- If the RetToSrc value is 0, must not return a copy if the cache line is Clean.

RetToSrc is applicable and must be set to zero in:

- Stash snoops.
- SnpCleanShared, SnpCleanInvalid, and SnpMakeInvalid,
- SnpOnceFwd and SnpUniqueFwd.

RetToSrc is applicable and can take any value in all other snoops except SnpDVMOp.

RetToSrc is inapplicable and must be set to zero in SnpDVMOp.

Home must only set RetToSrc on the Snoop request to a single Request Node.

4.10 Do not transition to SD

Do not transition to SD is a modifier on non-invalidating snoops.

It specifies when a Snoopee must not transition to SD state as a result of the Snoop request.

The field name is DoNotGotoSD.

Note

- A non-forced change or silent change from UD to SD is permitted irrespective of the value of DoNotGoToSD.
- Any forced change from Unique to Shared must obey DoNotGoToSD.

See *DoNotGoToSD* on page 13-420 for the field applicability and field value encoding.

4.11 Hazard conditions

This section lists the responsibilities of the RN-F and HN-F to handle address hazards and race conditions among Snoopable transactions. Ordering among Non-snoopable transactions and among Snoopable transactions is described in *Ordering* on page 2-116.

In addition to many Requesters issuing transactions at the same time, the protocol also permits each Requester to make multiple outstanding requests, and to receive multiple outstanding snoop requests. It is the responsibility of the interconnect, that is, ICN(HN-F, HN-I and MN), 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.

4.11.1 At the RN-F node

An RN-F node must respond to received Snoop requests, except for SnpDVMOp(Sync), in a timely manner without creating any Protocol layer dependency on completion of outstanding requests.

If a pending request to the same cache line is present at the RN-F and the pending request has not received any Data response packets:

- The snoop request must be processed normally.
- The cache state must transition as applicable for each snoop request type.
- The cached data or CopyBack request data must be returned with the snoop response, or forwarded to the Requester, if required by the Snoop request type, Snoop request attributes, and cache state.

If a pending request to the same cache line is present at the RN-F and the pending request has received at least one Data response packet:

- The RN-F must wait to receive all Data response packets before responding to the Snoop request.
- Once all the Data response packets are received by RN-F:
  - — The Snoop request must be processed normally.
  - — The cache state must transition as applicable for each Snoop request type.
  - — The cached data must be returned with the Snoop response, or forwarded to the Requester, if required by the Snoop request type, Snoop request attributes, and cache state.

If the pending request is a CopyBack request then the following additional requirements apply:

- Request transaction flow must be completed after receiving the CompDBIDResp.
- The cache state in the WriteData response must be the state of the cache line after the snoop request is processed, not the state at the time of sending the CopyBack request.
- An RN is permitted to not send valid CopyBack Data, if the cache line state after the Snoop response is sent is I or SC. The cache state in the WriteData response, after CopyBack Data is taken away by the snoop, must be I and all byte enables must be deasserted and the corresponding data must be set to zero.
- If data is included with WriteData it must be the same data that as sent with the Snoop response or more up to date data.

  Note

  More recent data than that sent with the snoop response can only be provided if the snoop was a SnpOnce, SnpOnceFwd, or SnpCleanShared and the Snoop response indicates that the cache line can be further modified.

For non-ordered transactions, the RN can send CompAck without waiting for DataSepResp. For ordered transactions, the RN can send CompAck as soon as the first packet of DataSepResp is received. In both cases, an RN must not respond to a Snoop request before receiving all data packets.

The RN-F might receive multiple snoop requests before it receives a response for a pending CopyBack request for the same cache line, in which case the data response carries the cache line state after completion of the response to the last snoop request. Such a scenario is possible because the CopyBack request can be queued behind multiple Read and Dataless requests at the HN-F.

4.11.2 At the ICN(HN-F) node

An HN-F orders transactions to the same cache line by sequencing transaction responses and snoop transactions to the Requesters. As the interconnect is not required to be ordered, the arrival order of these messages, in certain cases, might not be the same as the order in which they were issued at the HN-F.

If a Response message that includes data requires multiple packets or beats of transfers over the interconnect, then receiving or sending the message by Home implies sending or receiving all the packets corresponding to that message. That is, when a Home starts sending the message, it must send all packets of the message without dependence on completion of any other Request or Response message.

Similarly, a Home, when it accepts part of the Data message, must accept the remaining packets of that message without any dependence on forward progress of any other Request or Response message.

When a subsequent forwarding of data depends upon receiving a Data message, the forwarding of data action can occur after receiving the first Data packet. A subsequent non-data forwarding action that is processing of a subsequent request at Home as a consequence of sending or receiving of data by Home, must wait until all data is sent or received.

While a Snoop transaction response is pending, the only transaction responses that are permitted to be sent to the same address are:

- RetryAck for a CopyBack.
- RetryAck and DBIDResp for a WriteUnique and Atomics.
- RetryAck and, if applicable, a ReadReceipt for a Read request type.
- RetryAck for a Dataless request type.

Once a completion is sent for a transaction, the HN-F must not send a snoop request to the same cache line until it receives:

- A CompAck for any Read and Dataless requests except for ReadOnce* and ReadNoSnp.
- A WriteData response for CopyBack and Atomic requests.
- For WriteUnique, a WriteData response and, if applicable, CompAck.

Chapter 5

Interconnect Protocol Flows

This chapter shows interconnect protocol flows for different transaction types, and interconnect hazard conditions. The protocol flows are illustrated using Time-Space diagrams. It contains the following sections:

See *Time-Space diagrams* on page xv for details of the conventions used to illustrate protocol flow in this specification.

In the transaction flow diagrams that follow:

- There are multiple coherent RNs, an HN-F and a SN-F.
- If the HN-F receives multiple data responses, that is, one response from a snooped RN-F and another from a SN-F, then the data being forwarded to the Requester is highlighted in bold.
- There is no ICN cache at the HN-F, this results in all requests to the HN-F initiating a request to the SN-F.

5.1 Read transaction flows

This section gives examples of the interconnect protocol flow for Read transactions.

5.1.1 Read transactions with DMT and without snoops

For Read transactions without snoops, this specification recommends the use of *Direct Memory Transfer* (DMT).

Figure 5-1 shows an example DMT transaction flow using the ReadShared transaction.

In this example a response from SN-F to HN-F is not required because CompAck from the Requester is used to deallocate the request at Home.

The steps in the ReadShared transaction flow are:

1. RN-F sends a Read request to HN-F.
2. HN-F sends a Read request to SN-F.
    - The ID field values in the Read request are based on where the Data response is to be sent. Data can be sent to the Requester or to the HN-F. See Figure 2-29 on page 2-95 that shows an example of how the ID field values are derived.
3. SN-F sends a Data response directly to RN-F.
4. RN-F sends CompAck to HN-F as the Request is ReadShared and requires CompAck to complete the transaction.

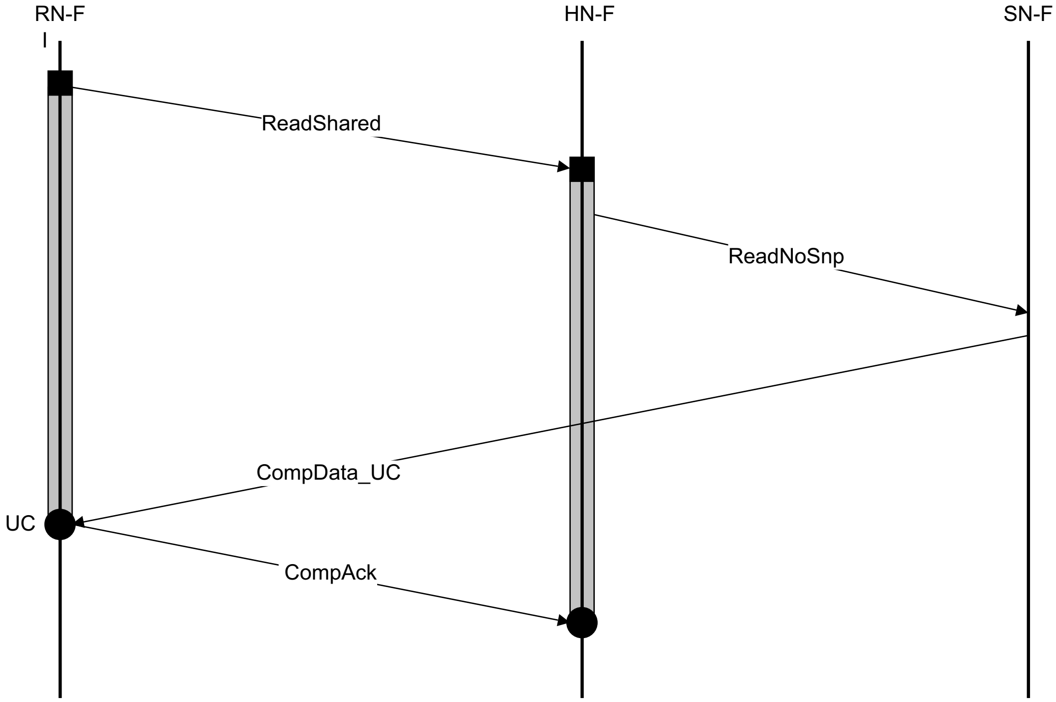


Figure 5-1 DMT Read transaction example without snoops

5.1.2 Read transaction with DMT and with snoops

For Read transactions with snoops and data from memory this specification recommends the use of DMT.

Figure 5-2 shows an example DMT transaction flow using the ReadShared transaction.

In this example a response from SN-F to HN-F is not required because CompAck from the Requester is used to deallocate the request at Home.

The steps in the ReadShared transaction flow are:

1. RN-F0 sends a Read request to HN-F.
2. HN-F sends a Snoop request to RN-F1.
3. HN-F sends a Read request to SN-F after receiving the Snoop response from RN-F1, which guarantees that RN-F1 has not responded with data.
    - The ID field values in the Read request are based on where the Data response is to be sent. Data can be sent to the Requester or to the HN-F. See Figure 2-29 on page 2-95 that shows an example of how the ID field values are derived.
4. SN-F sends a Data response directly to RN-F0.
5. RN-F0 sends CompAck to HN-F as the Request is ReadShared and requires CompAck to complete the transaction.

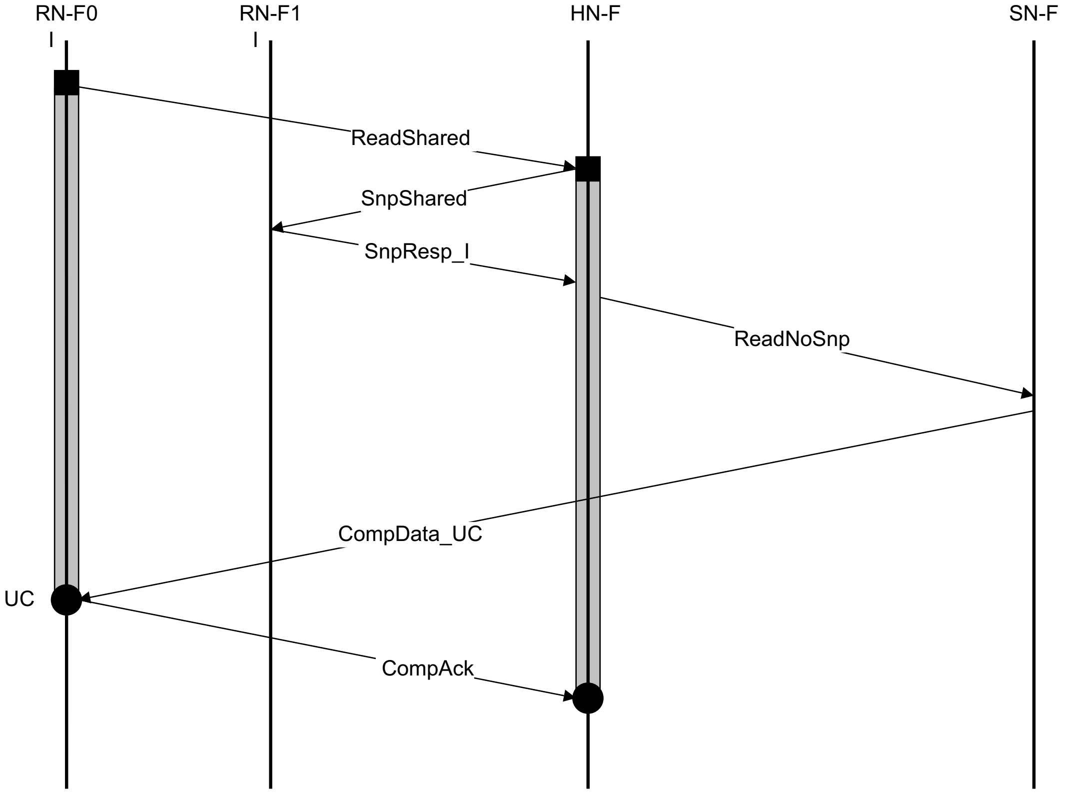


Figure 5-2 DMT Read transaction example with snoops and data from memory

5.1.3 Read transaction with DCT

For Read transactions with snoops and data from cache memory this specification recommends use of *Direct Cache Transfer* (DCT).

DCT from cache line in UC state

Figure 5-3 on page 5-241 shows an example flow for a DCT transaction. The Requester is RN-F0 and the forwarding cache is located at RN-F1.

The steps in the DCT transaction flow are:

1. RN-F0 sends a ReadShared request to HN-F.
2. HN-F sends a SnpSharedFwd, a Forwarding Snoop request to RN-F1.
3. RN-F1 cache line state transitions from UC to SC.
4. RN-F1 forwards CompData_SC response to RN-F0.
5. RN-F1 also sends a SnpResp_SC_Fwded_SC Snoop response to HN-F that indicates:
    - The data was forwarded to the Requester.
    - The final state of the cache line in the snooped cache is SC.
    - The state in which the cache line can be cached at the Requester is SC.
6. After receiving the CompData response RN-F0 sends a CompAck response to HN-F to conclude the transaction.

Note

Steps 4 and 5 in the DCT transaction flow can occur in any order as CompData and SnpResp are sent on different channels.

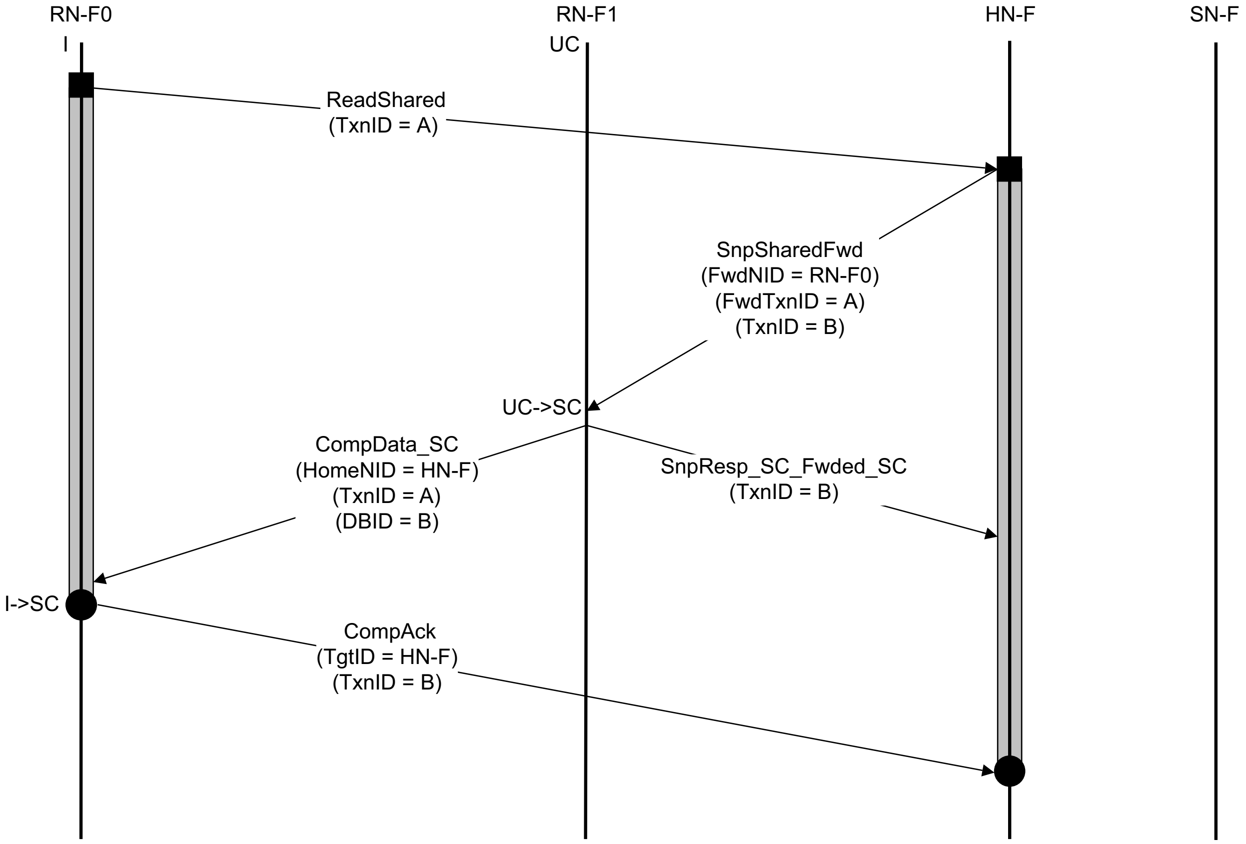


Figure 5-3 Direct Cache Transfer from cache line in UC state

Double data return in a DCT transaction

Figure 5-4 on page 5-242 shows an example DCT transaction flow that sends the data to HN-F as well as forwarding the data to the RN-F0.

The steps in the DCT transaction flow are:

1. RN-F0 sends a ReadShared request to HN-F.
2. HN-F sends a SnpSharedFwd Snoop request to RN-F1.
3. RN-F1 cache line state transitions from UD to SC.
4. RN-F1 sends CompData_SC response to RN-F0.
5. RN-F1 also sends a SnpRespData_SC_PD_Fwded_SC Snoop response to HN-F that includes a copy of the cache line and passes responsibility for the Dirty cache line to HN-F:
   - The data was forwarded to the Requester.
   - The final state of the cache line in the snooped cache is SC.
   - The state in which the cache line can be cached at the Requester is SC.
6. The RN-F0 sends CompAck after it receives the Data response to conclude the transaction.

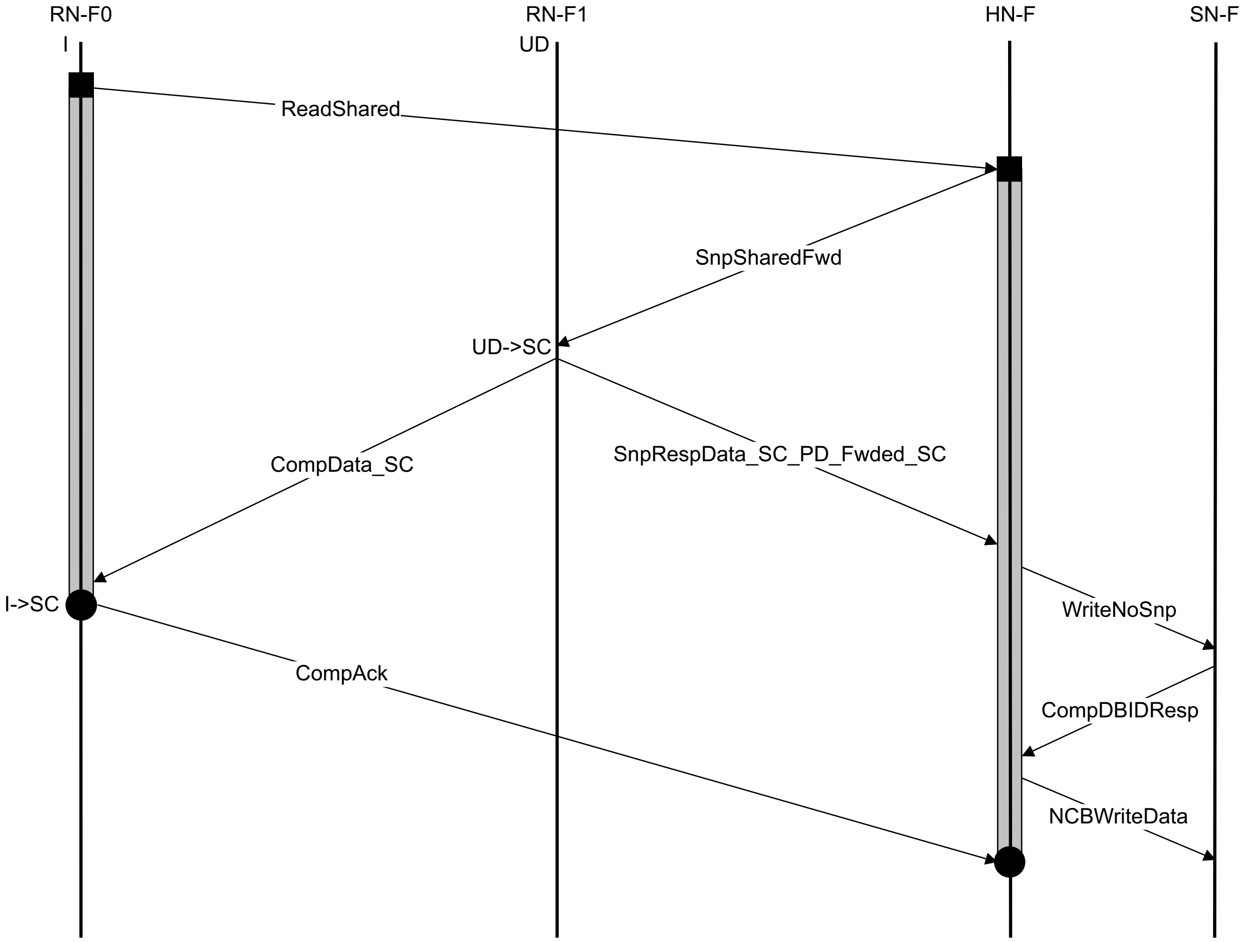


Figure 5-4 Double data return in a DCT transaction

5.1.4 Read transaction with neither DMT nor DCT

Figure 5-5 shows an example of the flow without DMT using the ReadNoSnp transaction. In this example, the ReadNoSnp has the ExpCompAck set in the original request.

The request does not generate any snoops and receives the data from a response to a memory read by the HN-F. The steps in the ReadNoSnp transaction flow are:

1. RN-F0 issues a ReadNoSnp transaction.
2. HN-F receives and allocates the request.

   Note

   HN-F does not send snoops as the request is recognized as a Non-snoopable request type.

3. HN-F sends a ReadNoSnp to SN-F.
4. SN-F returns data response to HN-F.
5. HN-F in turn returns the data to RN-F0. If ExpCompAck was not asserted in the ReadNoSnp request, then HN-F deallocates the request.
6. If ExpCompAck was asserted in the ReadNoSnp request, RN-F0 sends a CompAck response to HN-F.
7. RN-F0 deallocates the request.
8. HN-F receives the CompAck response and deallocates the request.

Figure 5-5 shows the transaction flow, the copy of data being transferred is marked in bold.

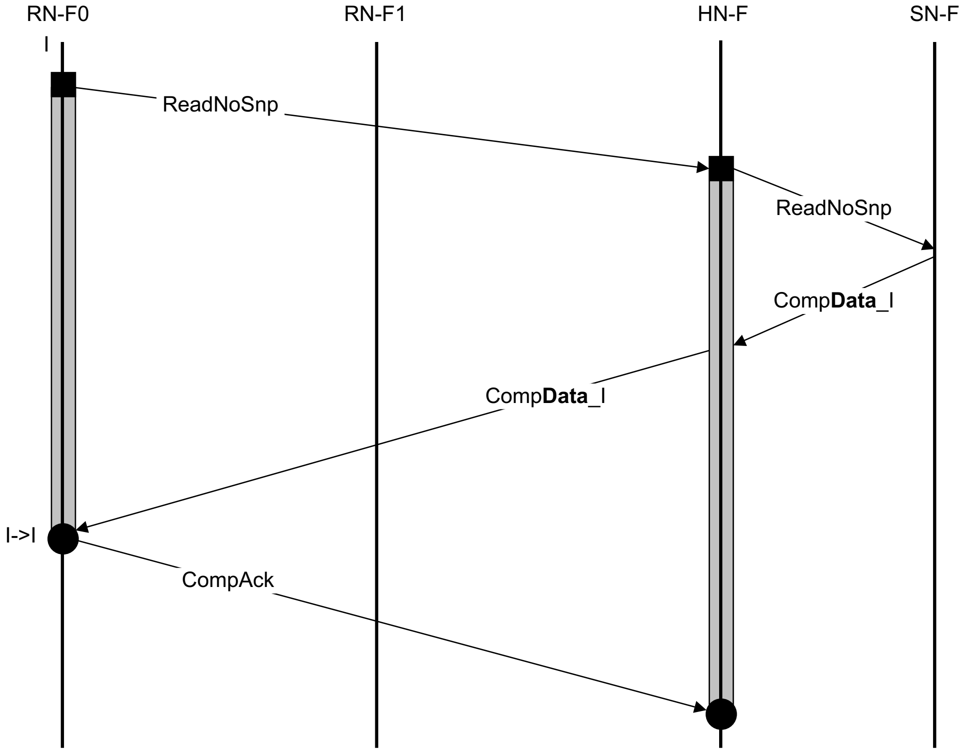


Figure 5-5 ReadNoSnp transaction flow

5.1.5 Read transaction with snoop response with partial data and no memory update

An example of this type of flow is a ReadUnique transaction.

RN-F1 has the cache line in UDP state. RN-F1 responds to the snoop with a snoop response with partial cache line data and passes responsibility for updating memory.

HN-F waits for the data response from memory, merges the partial snoop response data with the data response from memory, and sends the resultant data to the Requester.

HN-F does not update memory because responsibility for updating memory is passed on to the Requester.

Figure 5-6 shows the transaction flow, the copy of data being transferred is marked in bold.

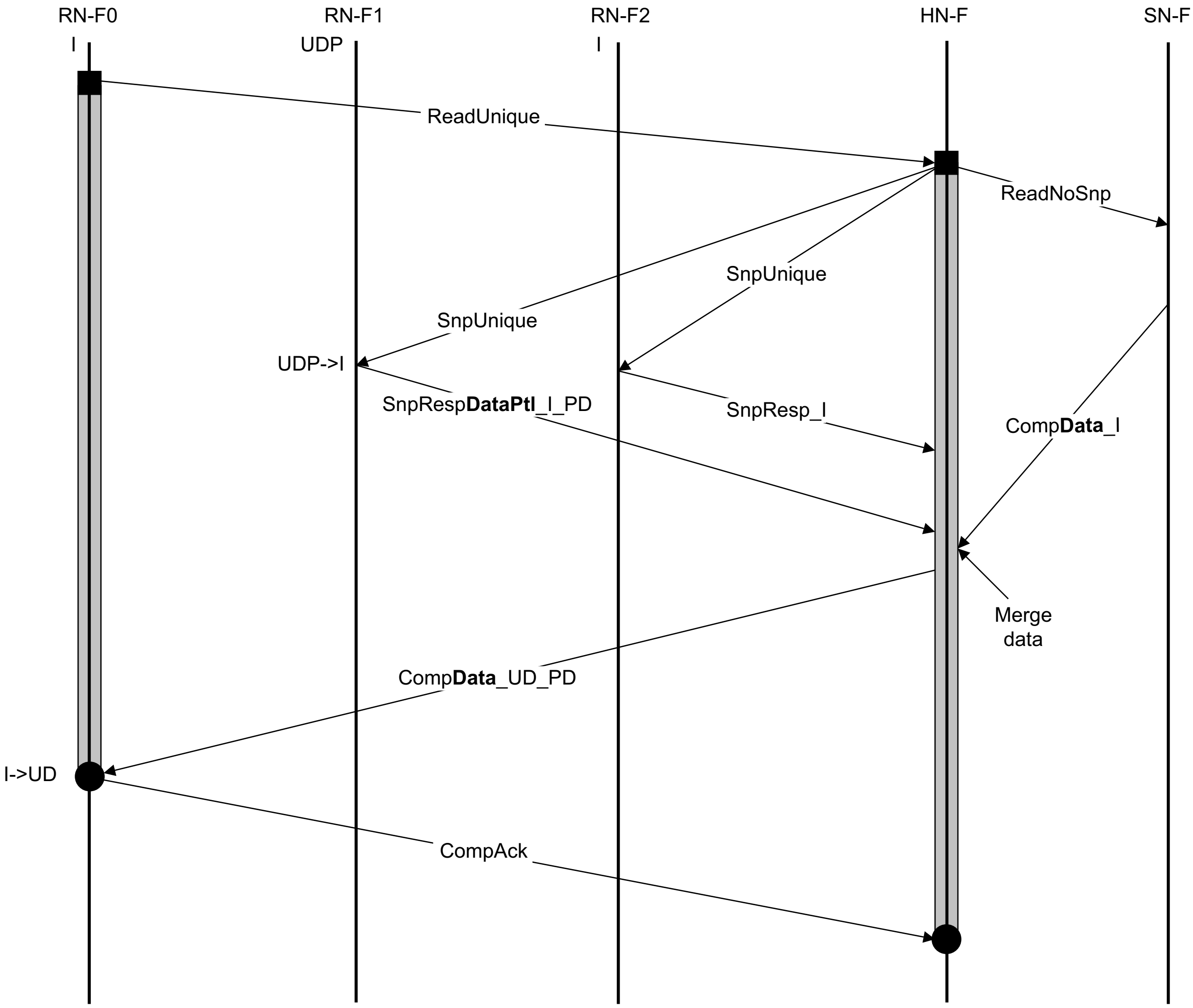


Figure 5-6 ReadUnique with partial data snoop response

5.1.6 Read transaction with snoop response with partial data and memory update.

An example of this type of flow is a ReadClean transaction.

RN-F1 has the cache line in UDP state. RN-F1 responds to the snoop with a snoop response with partial cache line data and passes responsibility for updating memory.

HN-F waits for the data response from memory, merges the partial snoop response data with the data response from memory, and sends the resultant data to the Requester.

HN-F updates memory as the responsibility for updating memory is not passed on to the Requester.

Figure 5-7 shows the transaction flow, the copy of data being transferred is marked in bold.

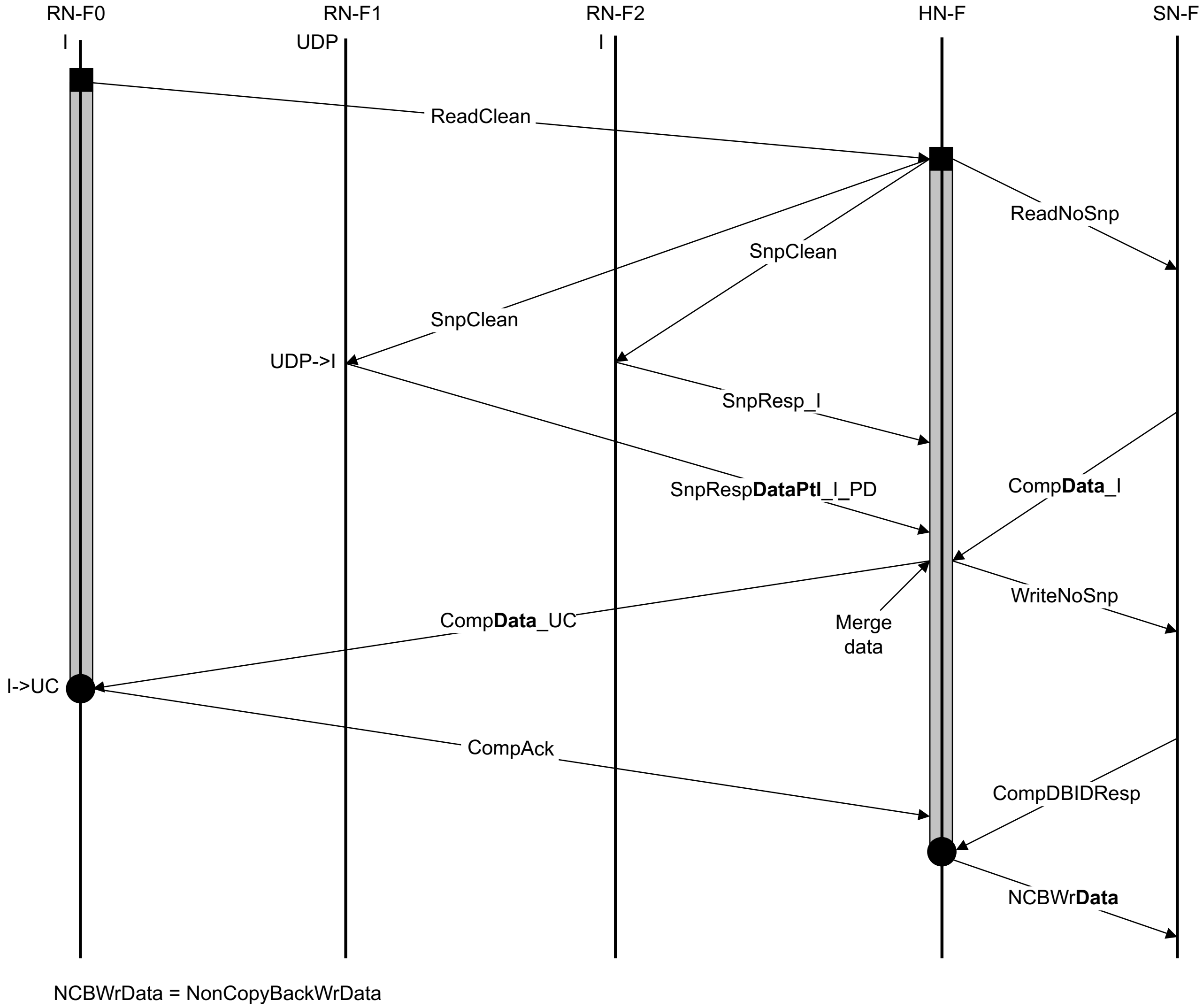


Figure 5-7 ReadClean with partial data snoop response

5.1.7 ReadOnce* and ReadNoSnp with early Home deallocation

Figure 5-8 shows the optimized flow for an unordered ReadOnce request.

The steps in the optimized ReadOnce transaction flow are:

1. RN-F0 sends an unordered ReadOnce request to HN-F with Order[1:0] set to `0b00`.
2. HN-F sends a DMT ReadNoSnp request to SN-F with the Order[1:0] set to `0b01`.
3. SN-F sends ReadReceipt to Home.
4. HN-F deallocates the request after receiving the ReadReceipt response.
5. SN-F sends CompData_UC directly to RN-F0.

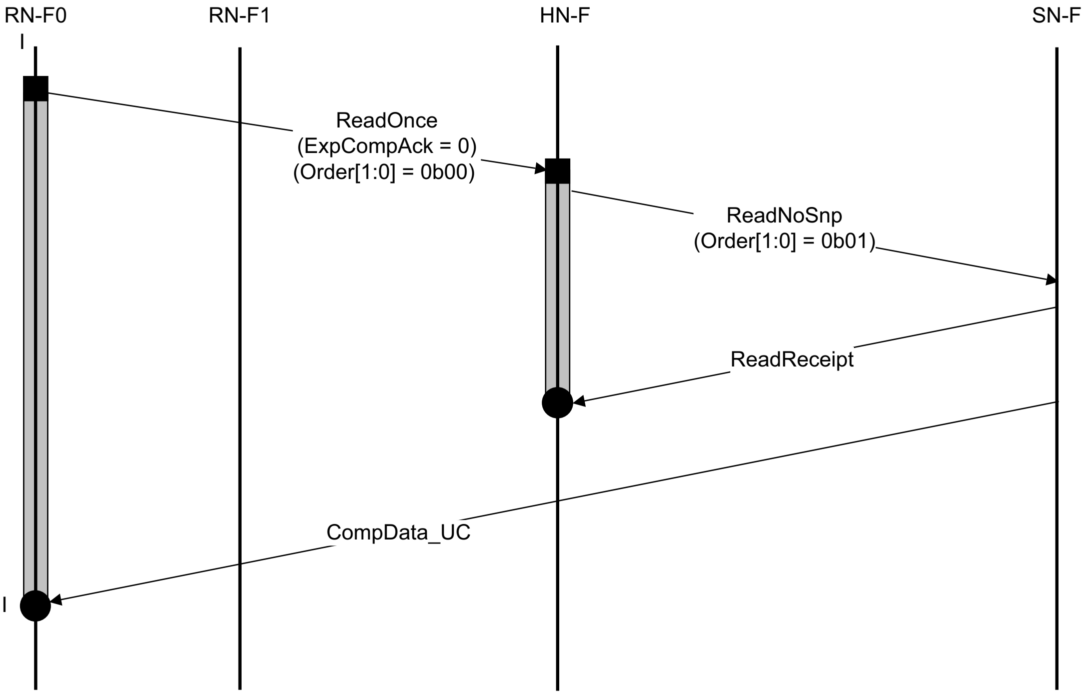


Figure 5-8 DMT optimization for unordered ReadOnce

Note

Use of a ReadNoSnp transaction from Home to Slave, in the case where CompAck is not required, avoids the need to send a RespSepData response from Home to Requester.

5.1.8 ReadNoSnp transaction with DMT and separate Non-data and Data-only response

Figure 5-9 shows an example DMT transaction flow with separate Non-data and Data-only response.

In this example there is no ordering requirement and RN-F can send CompAck to HN-F to deallocate the request at Home without waiting for DataSepResp.

The steps in the ReadNoSnp transaction are:

1. RN-F sends a ReadNoSnp request to HN-F.
2. HN-F sends a ReadNoSnpSep request to SN-F.
    - This tells SN-F that a data only response is required.
3. HN-F sends a RespSepData response to RN-F.
    - This tells RN-F that the request has been allocated at HN-F and that there will be a separate Data response.
4. SN-F sends a ReadReceipt to HN-F.
    - This tells HN-F that the request to the Slave is completed and ensures that the SN will not respond with RetryAck for that request. As the request is a ReadNoSnp with Order field set to zero, the request at HN-F can be deallocated on receipt of ReadReceipt.
5. RN-F sends CompAck after receiving RespSepData.
    - For ReadNoSnp, CompAck is not functionally required, HN-F that deallocates the request after receiving the ReadReceipt from SN-F can just drop the CompAck.
6. SN-F, as Completer, sends DataSepResp to RN-F returning the read data.

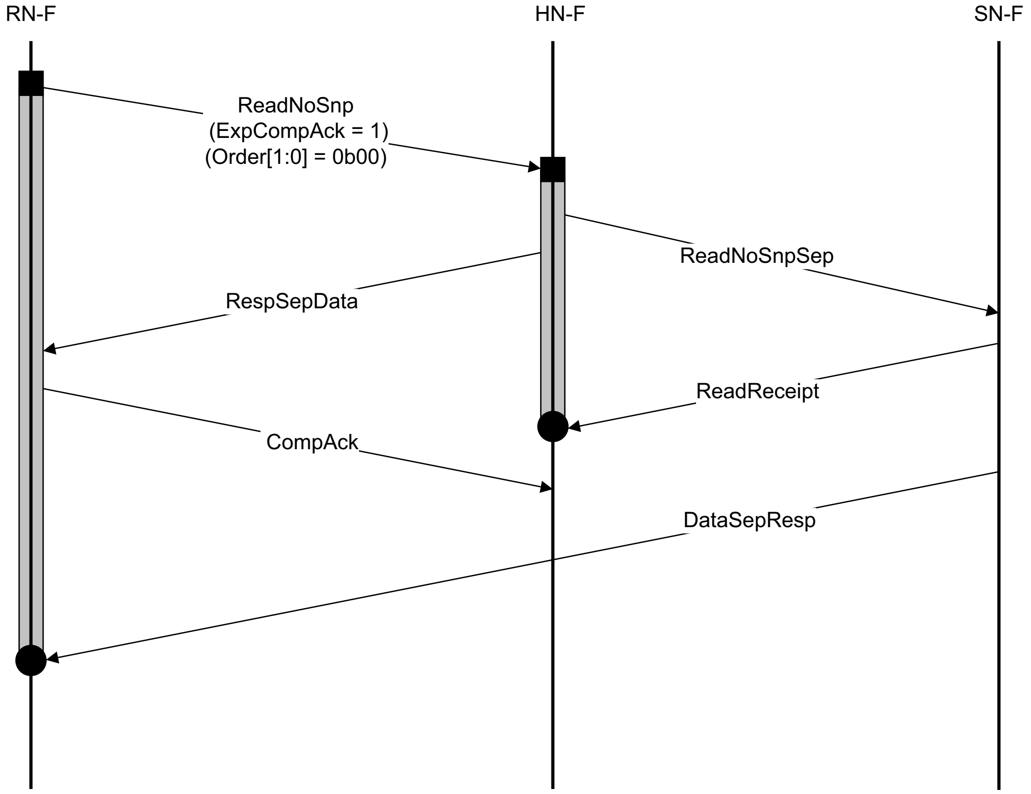


Figure 5-9 DMT Read transaction example with separate Non-data and Data-only

5.1.9 ReadNoSnp transaction with DMT with ordering and separate Non-data and Data-only

Figure 5-10 shows an example DMT transaction flow with ordering and separate Non-data and Data-only.

This example, which has ReadNoSnp with non-zero Order field requires that:

- Next ordered request can be sent only after receiving of RespSepData.
- RN-F must wait for RespSepData and at least one packet of DataSepResp before sending CompAck.
- HN-F must not send next ordered request to SN-F until it receives CompAck.

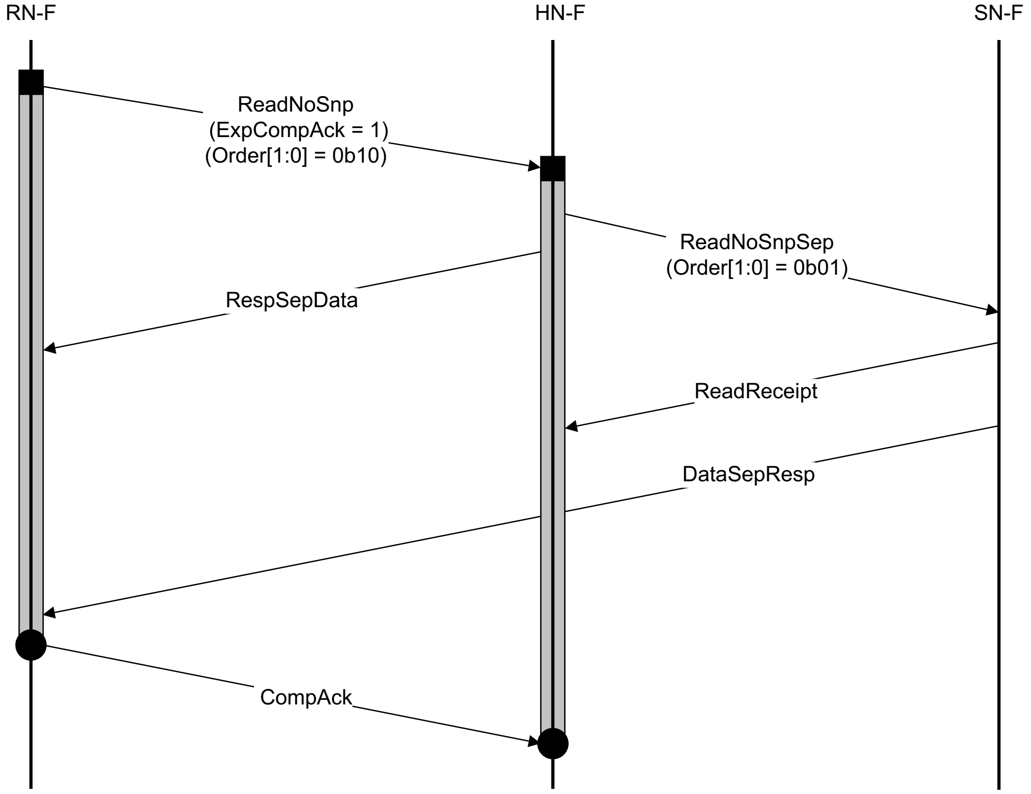


Figure 5-10 DMT Read transaction example with ordering and separate Non-data and Data-only

5.2 Dataless transaction flows

This section gives examples of the interconnect protocol flow for Dataless transactions.

5.2.1 Dataless transaction without memory update

An example of this type of flow is a MakeUnique transaction.

RN-F1 has the cache line in UC state. RN-F1 responds to the snoop with a snoop response without data and changes the cache line state to I.

HN-F waits for all snoop responses and then sends a Comp_UC response to the Requester.

HN-F does not send a Read request to SN-F because the request is a Dataless transaction.

Figure 5-11 shows the transaction flow.

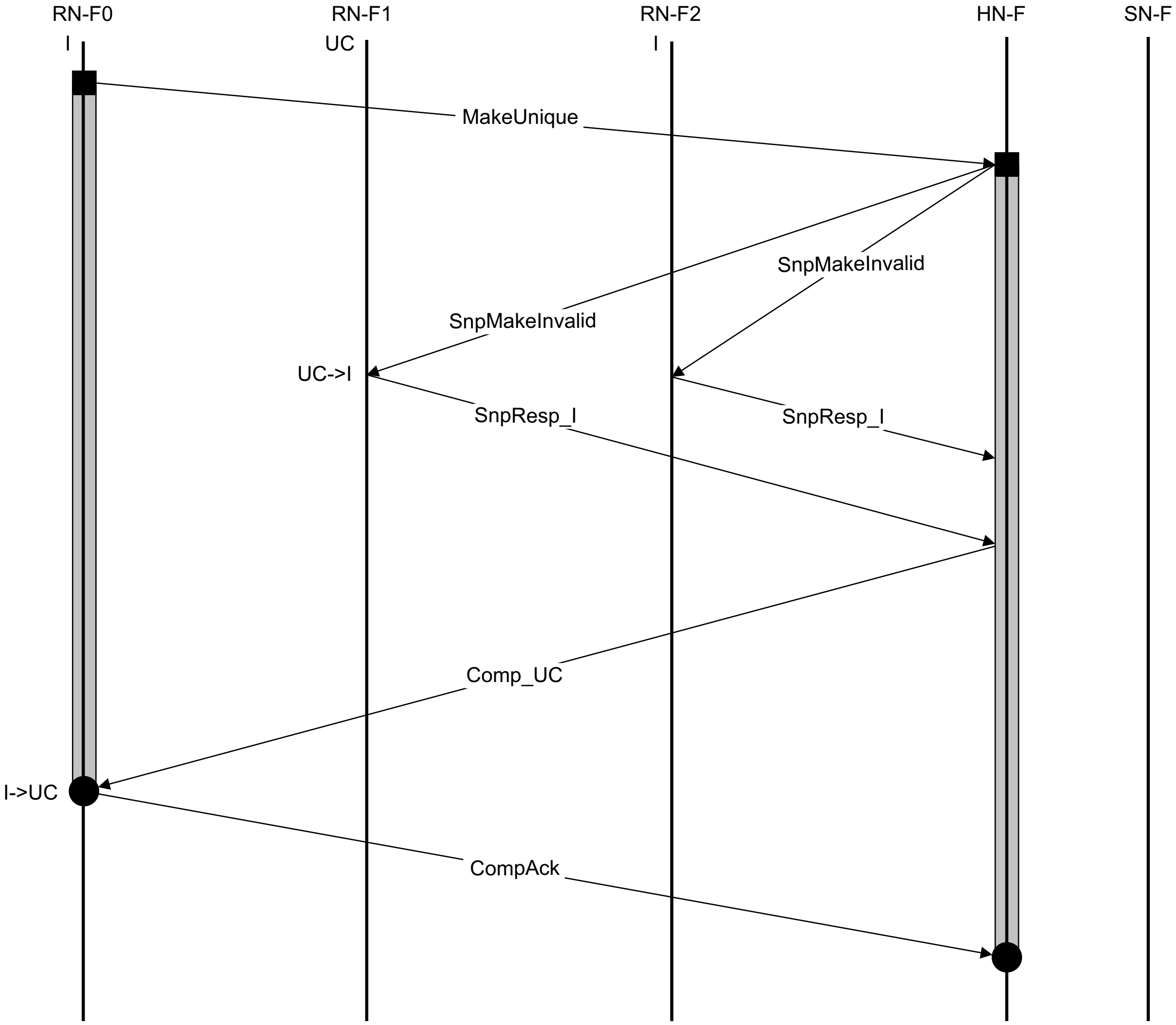


Figure 5-11 MakeUnique without memory update

5.2.2 Dataless transaction with memory update

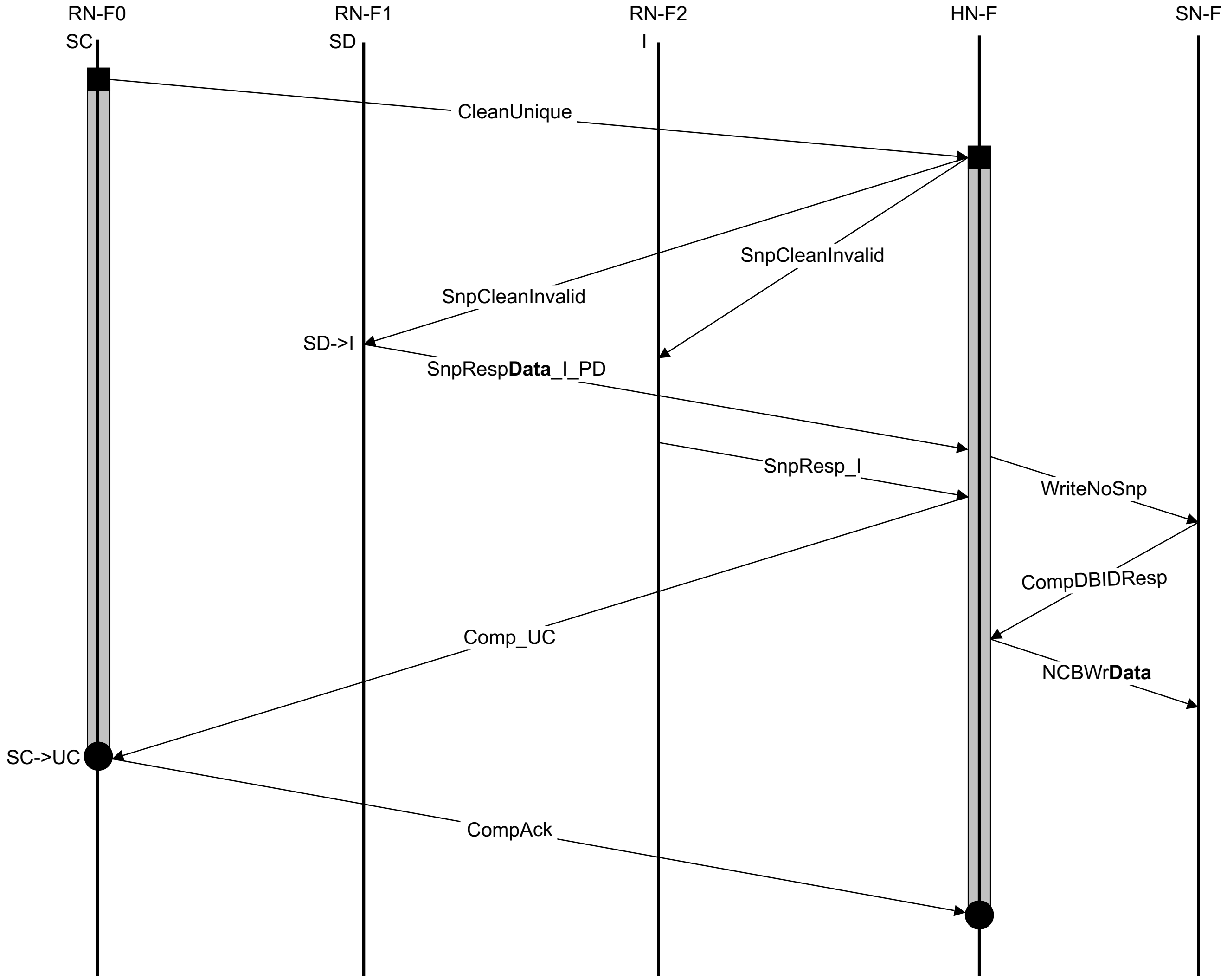
An example of this type of flow is a CleanUnique transaction.

RN-F1 has the cache line in SD state and responds to the snoop with a snoop response with data and PD asserted.

HN-F waits for all snoop responses and then sends a Comp_UC response to the Requester.

HN-F sends a Write request to update memory with the data received from RN-F1.

Figure 5-12 shows the transaction flow.



NCBWrData = NonCopyBackWrData

Figure 5-12 CleanUnique with memory update

5.2.3 Persistent CMO with snoop and separate Comp and Persist

In this example of CleanSharedPersistSep transaction flow, the *Point of Persistence* (PoP) is at the SN-F.

RN-F1 has the cache line in SC state. RN-F1 responds to the snoop with a snoop response without data.

HN-F waits for all snoop responses and then sends a Comp_SC response to the Requester.

HN-F sends a CleanSharedPersistSep request to SN-F, only after completing the writing back of all snooped Dirty data, if any, to the SN-F. SN-F responds to the request with Comp.

SNF sends a Persist response to RN-F0 to indicate that the request has reached the PoP, and data from any prior writes to the same location is pushed to the PoP.

Figure 5-13 shows the transaction flow.

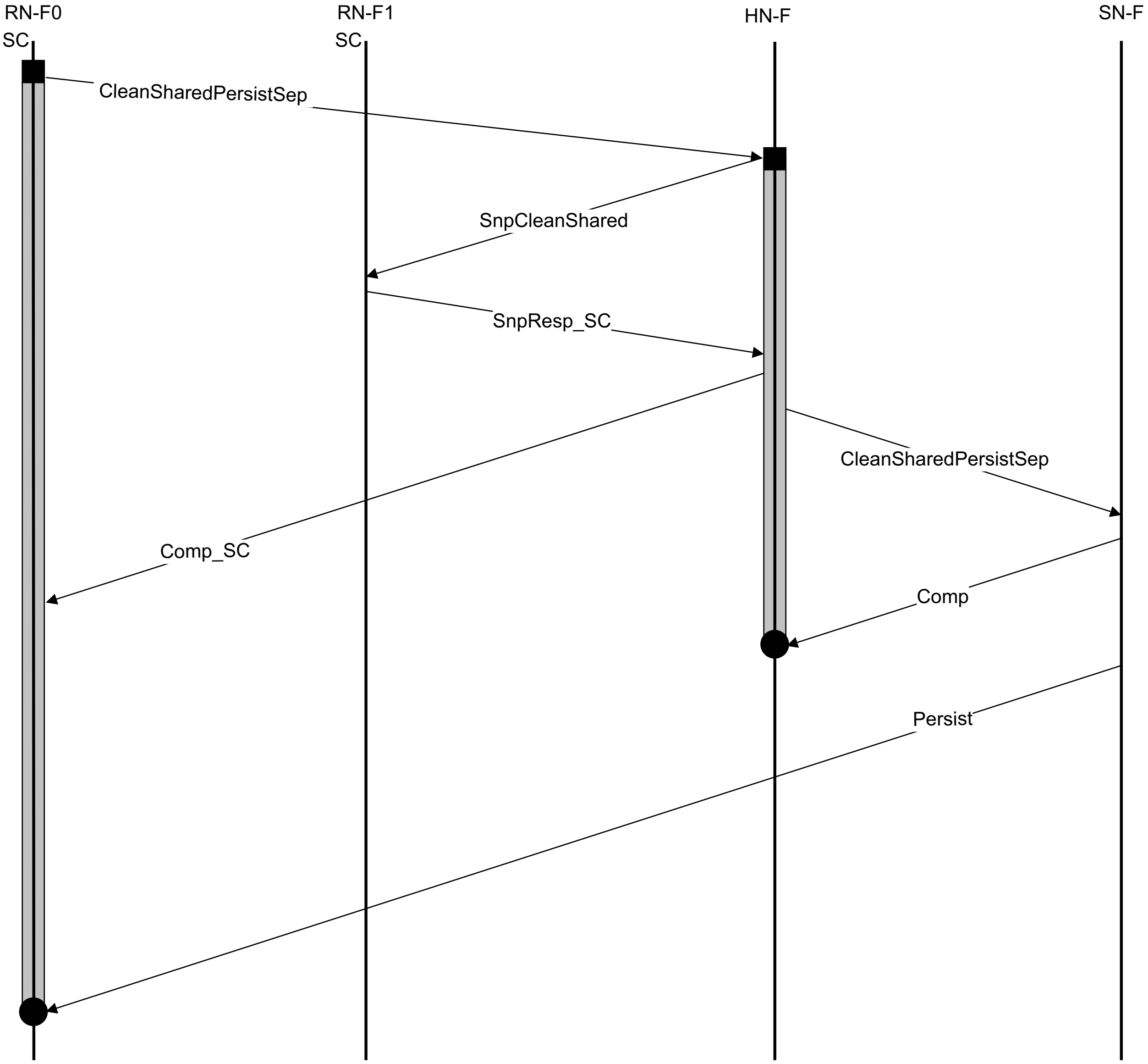


Figure 5-13 CleanSharedPersistSep transaction flow

5.2.4 Evict transaction

Figure 5-14 shows the Evict transaction flow.

RN-F0 moves the cache line to I state and issues an Evict transaction.

HN-F receives and allocates the request.

Note

The Evict request is a hint. A Comp response can be given by HN-F without updating the Snoop Filter or Snoop Directory.

HN-F returns the Comp response and deallocates the request.

RN-F0 deallocates the request.

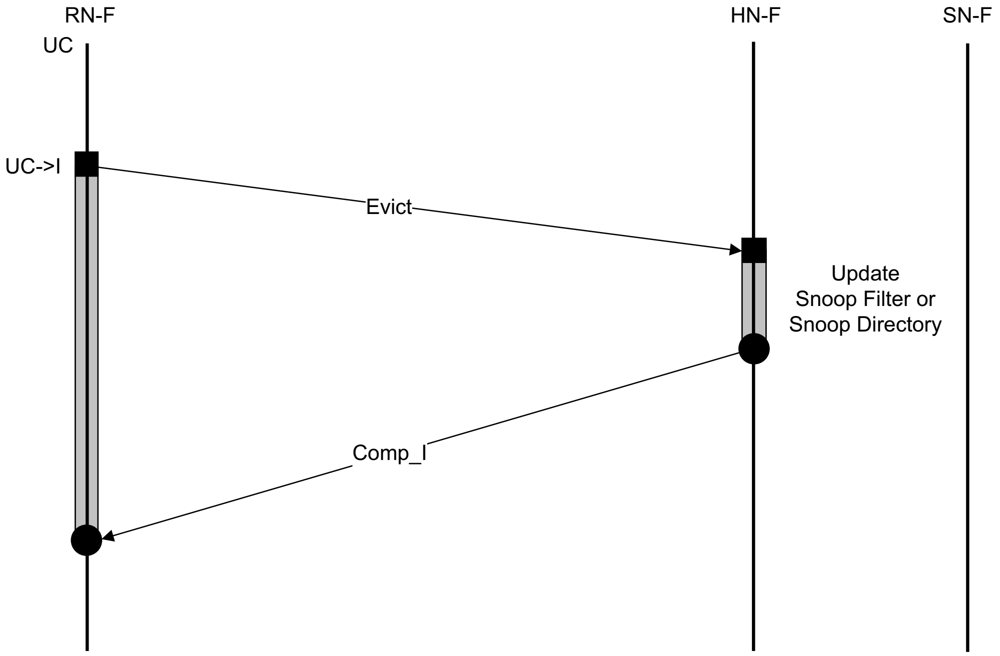


Figure 5-14 Evict transaction flow

Note

The cache state at the Requester must change to Invalid before the Evict message is sent.

5.3 Write transaction flows

This section gives examples of the interconnect protocol flow for Write transactions.

5.3.1 Write transaction with no snoop and separate responses

An example of this type of flow is a WriteNoSnp transaction. The steps in the WriteNoSnp transaction flow are:

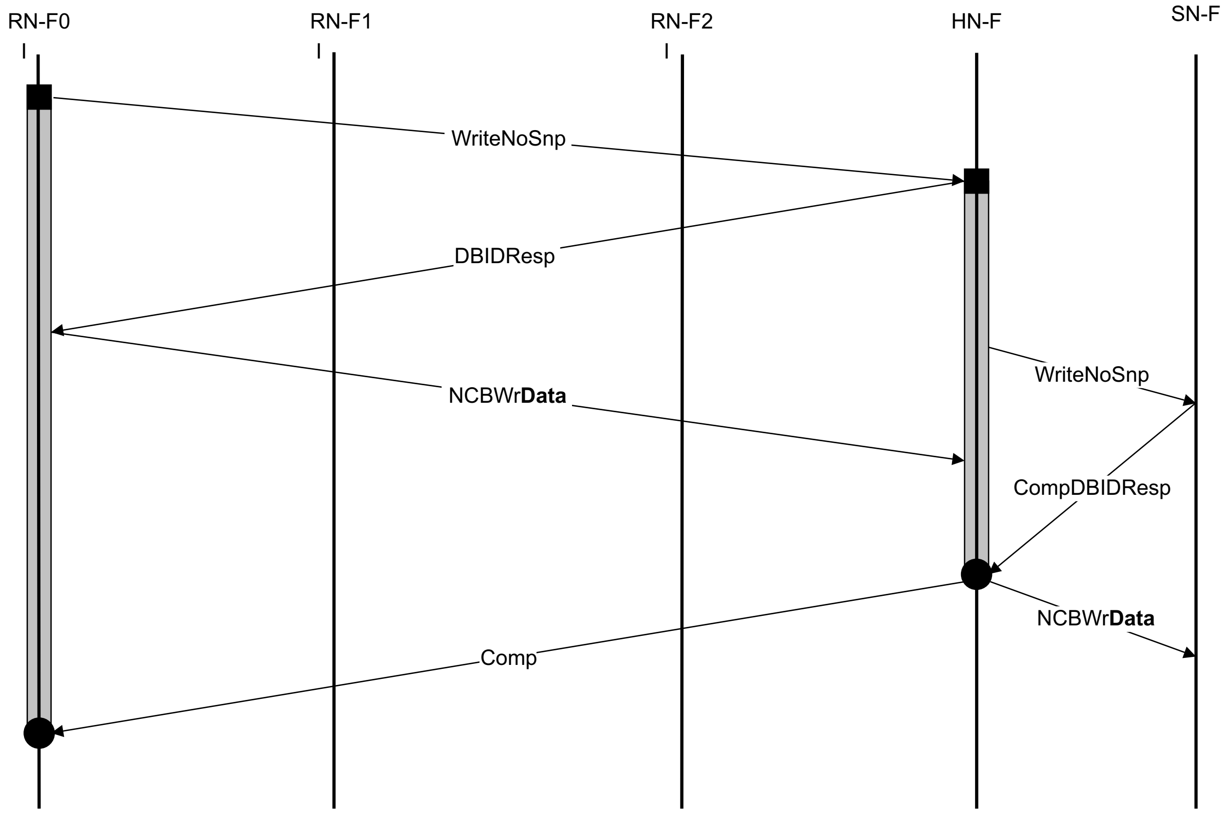
1. RN-F0 issues a WriteNoSnp transaction.
2. HN-F receives and allocates the request.
3. HN-F sends DBIDResp without Comp.
4. RN-F0 responds with data.
5. HN-F sends a Comp after it receives CompDBIDResp from SN-F.

   Note

   This flow example shows Comp is sent after CompDBIDResp is received from SN-F. However, HN-F is permitted to send Comp anytime after it receives the WriteNoSnp request from RN-F0.

6. RN-F0 waits for Comp from HN-F and deallocates its request.

Figure 5-15 shows the flow, the copy of data being transferred is marked in bold.



NCBWrData = NonCopyBackWrData

Figure 5-15 WriteNoSnp with separate responses from HN to RN

5.3.2 Write transaction with snoop and separate responses

An example of this type of flow is a WriteUniquePtl transaction.

The Comp_I response from HN-F must be sent when the coherency activity is complete at HN-F.

Figure 5-16 shows the transaction flow, the copy of data being transferred is marked in bold.

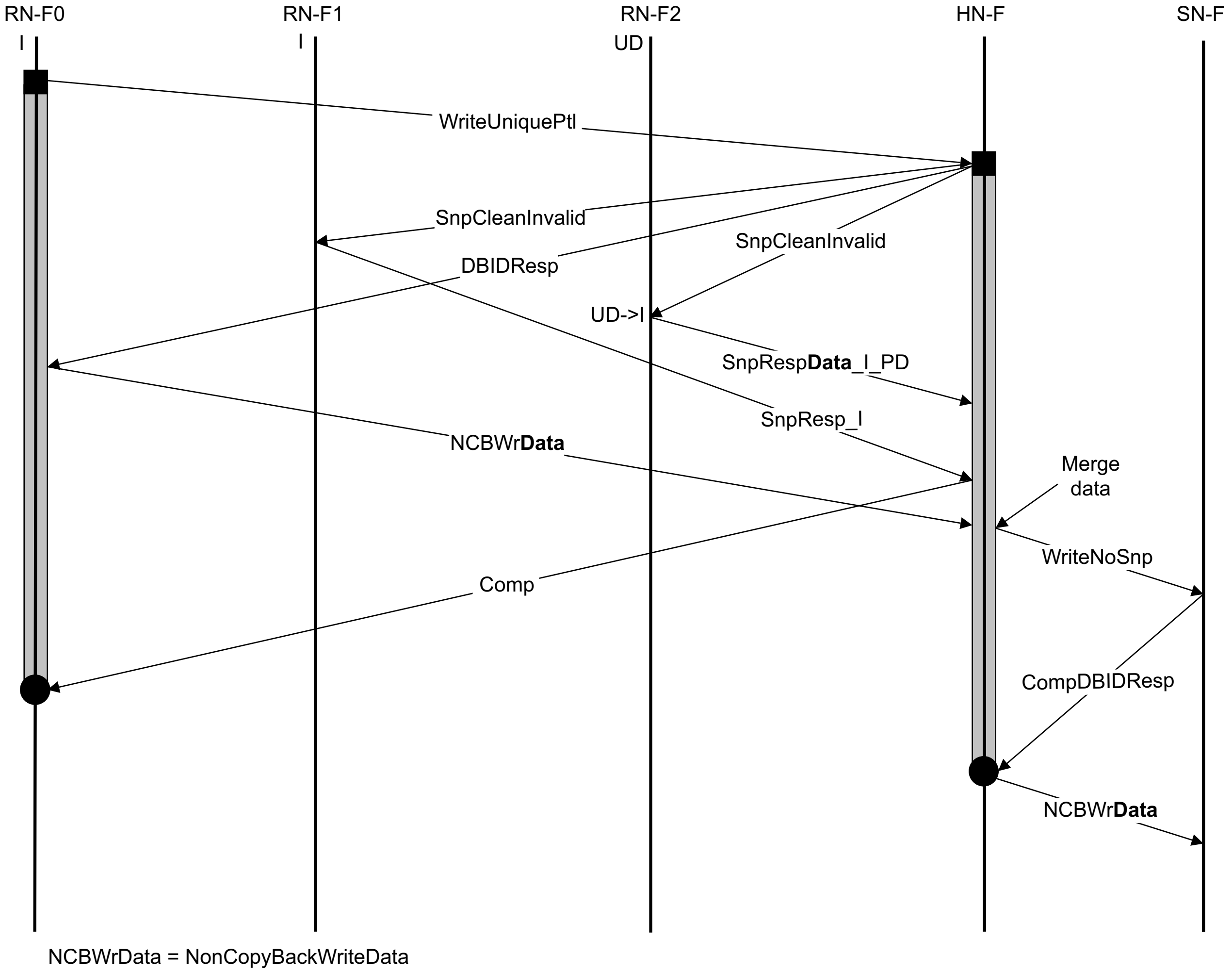


Figure 5-16 WriteUniquePtl with snoop

5.3.3 CopyBack Write transaction to memory

An example of this type of flow is a WriteBackFull transaction.

The data received from RN-F0 is written to SN-F by HN-F using a WriteNoSnp transaction.

Figure 5-17 shows the transaction flow, the copy of data being transferred is marked in bold.

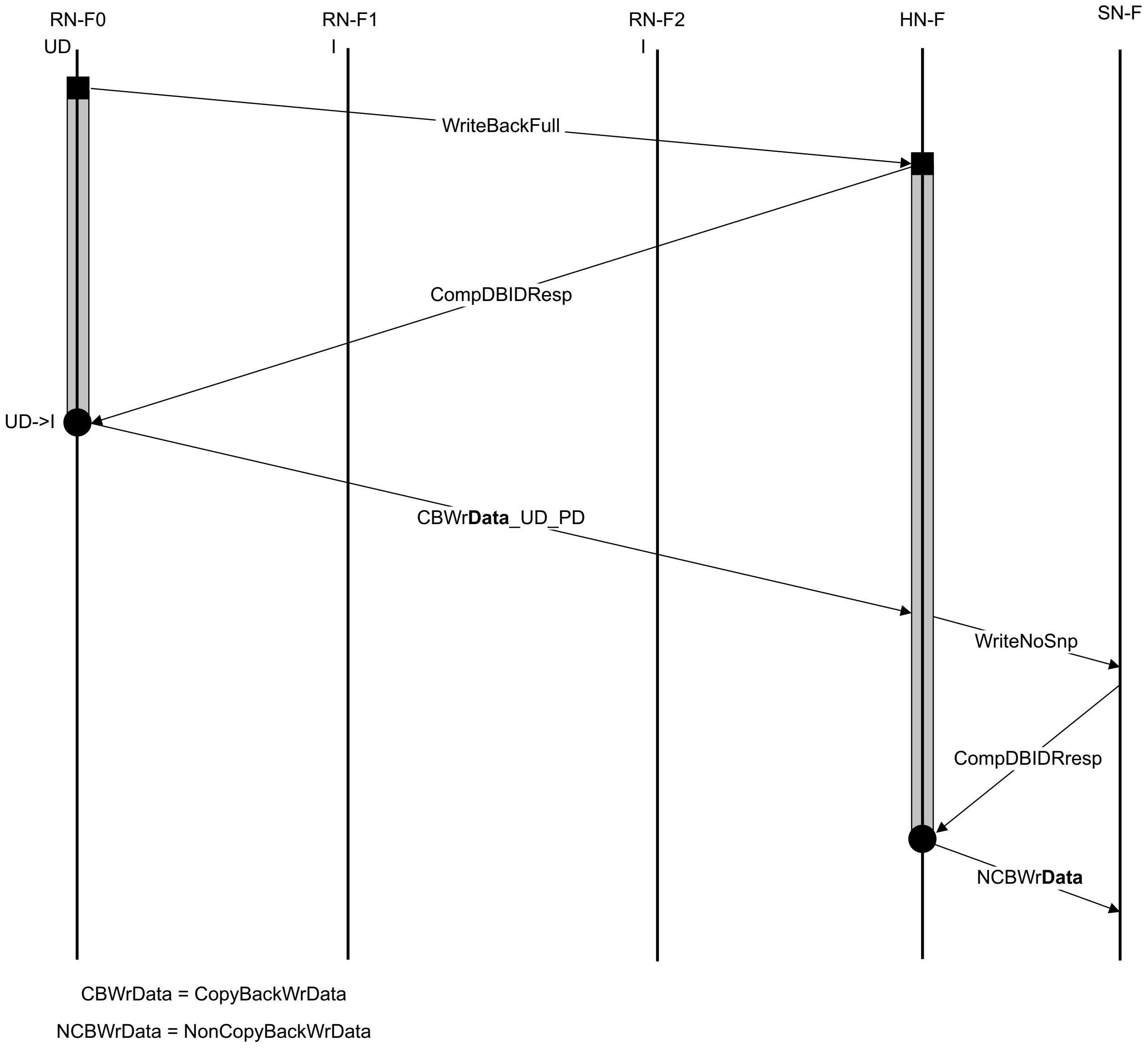


Figure 5-17 WriteBackFull returning Data Buffer Identifier

5.4 Atomic transaction flows

This section shows flows for different Atomic transaction types. It contains the following sub-sections:

- *Atomic transactions with data return*.
- *Atomic transaction without data return* on page 5-259.
- *Atomic operation executed at the SN* on page 5-261.

5.4.1 Atomic transactions with data return

This flow is applicable to:

- AtomicLoad.
- AtomicCompare.
- AtomicSwap.

Atomic transaction with snoops and data return

Figure 5-18 on page 5-257 shows the atomic operation executed at HN-F.

The steps in this transaction flow are:

1. RN-F0 sends an Atomic transaction to HN-F.
2. After receiving the Atomic request, HN-F:
   - Sends DBIDResp to RN-F0 to obtain the Atomic transaction data.
   - Sends SnpUnique Snoop request to other RN-Fs after determining that snoops are required.
   - HN-F is permitted to send a speculative ReadNoSnp to SN-F.
3. RN-F2 has the cache line in UD state and responds by sending data and invaliding its own cached copy.
   - The response is SnpRespData_I_PD.
   - This data is marked as (Old_Data) in Figure 5-18 on page 5-257 to distinguish it from both the data sent by the Requester and the data written to SN-F after the atomic operation is executed.
   - HN-F also receives a second Snoop response, SnpResp_I, from RN-F1.
4. After receiving all Snoop responses, HN-F sends CompData_I to the Requester.
   - The data sent with Comp is the old copy of the data.
   - This data must not be cached in a coherent state at RN-F0.
5. In response to the DBIDResp sent previously, HN-F receives the NonCopyBackWrData_I response from the Requester.
   - This data is marked as (Txn_Data) in Figure 5-18 on page 5-257 to distinguish it from the data sent by RN-F2 in response to the Snoop request from HN-F,
6. Once HN-F receives the NonCopyBackWrData_I response from the Requester, and the Snoop response with data from RN-F2, it executes the atomic operation.
7. The resulting value after atomic operation execution, marked as New_Data in the figure, is written to SN-F.
8. In this example, the read data received due to the speculative read is discarded by HN-F.

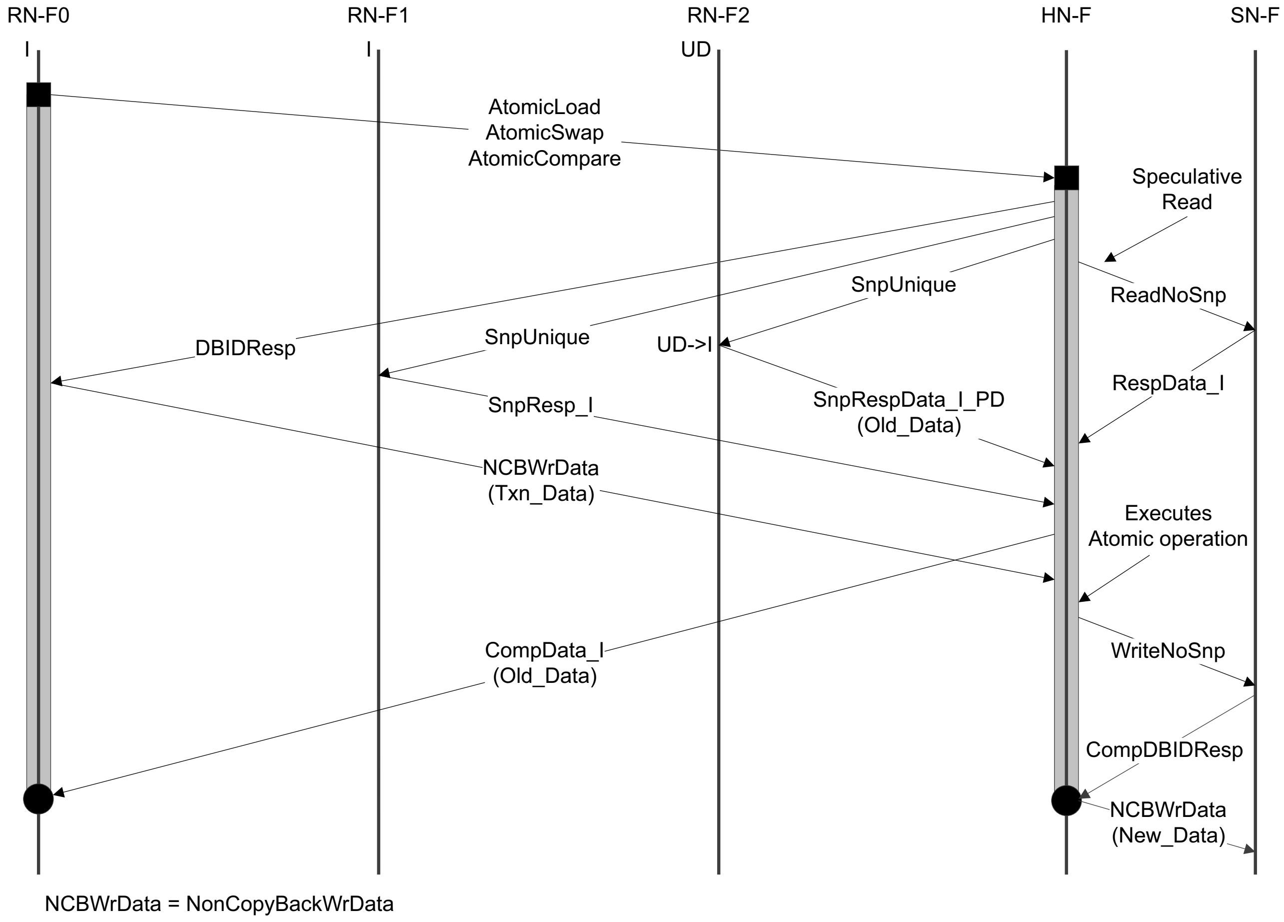


Figure 5-18 AtomicLoad, AtomicSwap, or AtomicCompare executed at HN-F

Note

In Figure 5-18, the CompData_I response from HN-F can be sent as soon as all Snoop responses are received.

Alternatively, to aid error reporting, CompData_I can be delayed until NCBWrData is received from the Requester and the atomic operation is executed.

Atomic transaction without snoops and with data return

Figure 5-19 shows the atomic operation executed at HN.

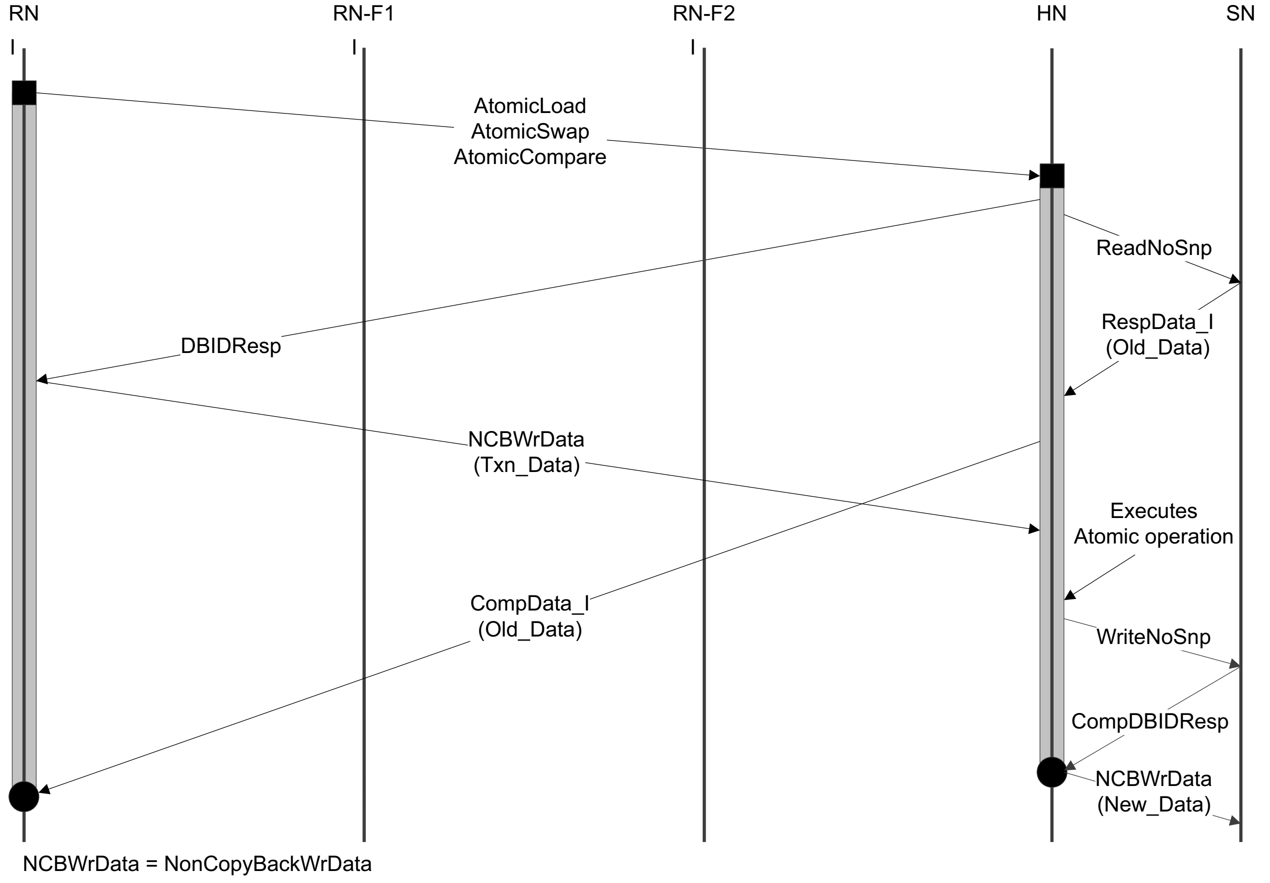


Figure 5-19 AtomicLoad, AtomicSwap, or AtomicCompare executed at HN

5.4.2 Atomic transaction without data return

This flow is applicable to AtomicStore transactions.

Atomic transaction with snoops and without data return

Figure 5-20 shows the atomic operation executed at HN-F. The flow is similar to the Atomic transaction with snoop and with data return, except that the Comp response to RN-F0 does not include data.

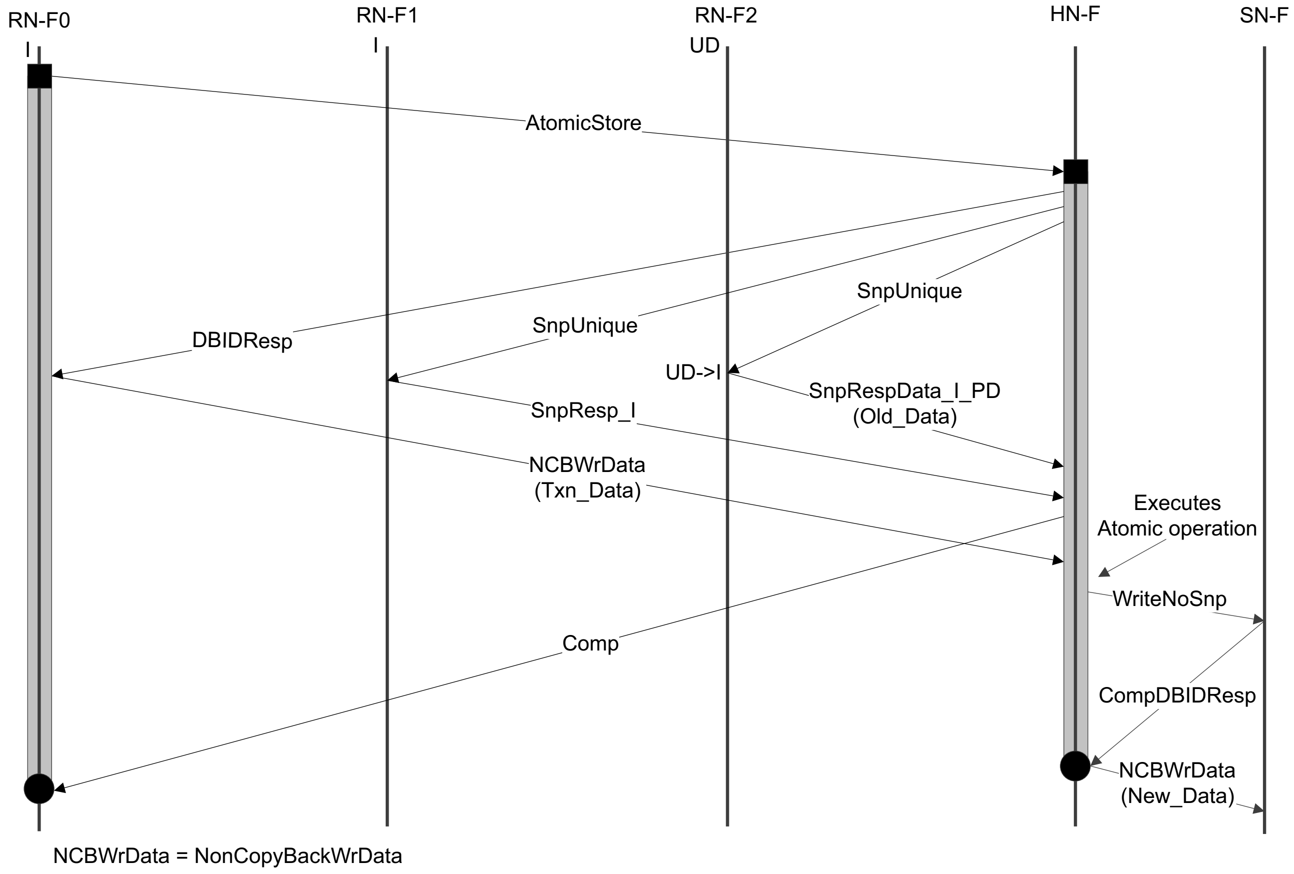


Figure 5-20 AtomicStore executed at HN-F

Atomic transaction without snoops and without data return

Figure 5-21 shows the atomic operation executed at HN. The flow is similar to the Atomic transaction without snoop and with data return except that the Comp response to RN does not include data.

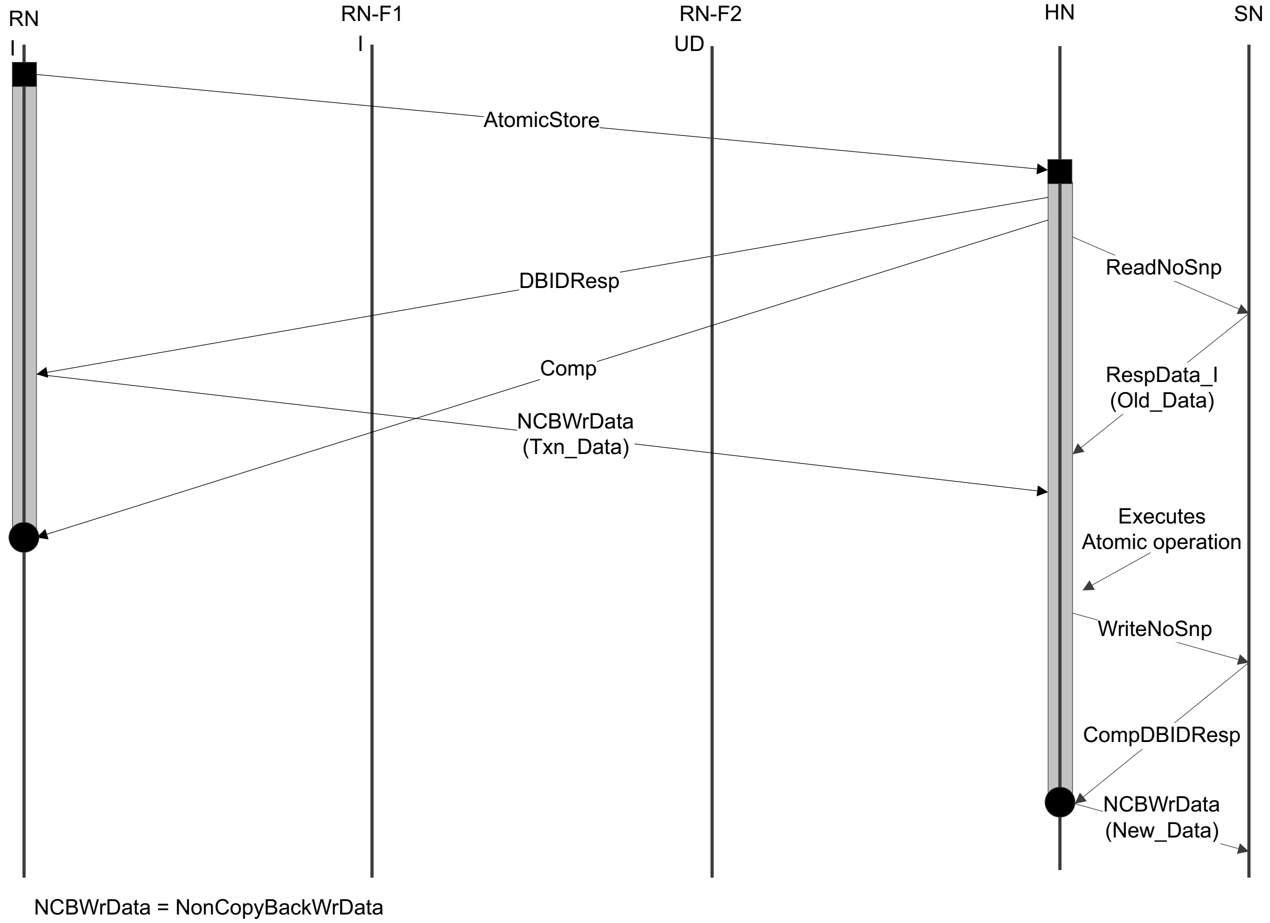


Figure 5-21 AtomicStore executed at HN

Note

- In Figure 5-21, the read from SN is required to obtain the Old_Data and is not speculative.
- The Comp response from HN can be combined with the DBIDResp response.

5.4.3 Atomic operation executed at the SN

Figure 5-22 on page 5-262 shows an example Atomic transaction flow where the SN-F is executing the atomic operation.

The steps in this transaction flow are:

1. RN-F0 sends an AtomicStore transaction to HN-F.
    - The Atomic request is to a Snoopable address location.
2. After receiving the Atomic request, HN-F:
    - Sends DBIDResp to RN-F0 to obtain the Atomic transaction data.
    - Sends a SnpUnique to other RN-Fs after determining that snoops are required.
3. RN-F2 has the cache line in UD state and responds by sending data and invaliding its own cached copy.
    - The response is SnpRespData_I_PD.
    - This data is marked as (Old_Data) in Figure 5-22 on page 5-262 to distinguish it from both the data sent by the Requester and the data written to SN-F after the atomic operation is executed.
    - HN-F also receives a second Snoop response, SnpResp_I, from the other snooped RN-F.
4. HN-F writes the received data to SN-F using a WriteNoSnp transaction.
5. In response to the DBIDResp sent previously, HN-F receives the NonCopyBackWrData response from the Requester.
6. HN-F after sending the Snoop response data to SN-F, sends an AtomicStore transaction request to SN-F, and executes the sequence of messages required to complete the Atomic transaction.
7. The HN-F deallocates the request once the Comp response is sent to the Requester and the Comp response for the Atomic transaction is received from SN-F.
    - The Comp response from HN-F can be sent as soon as all the Snoop responses are received.

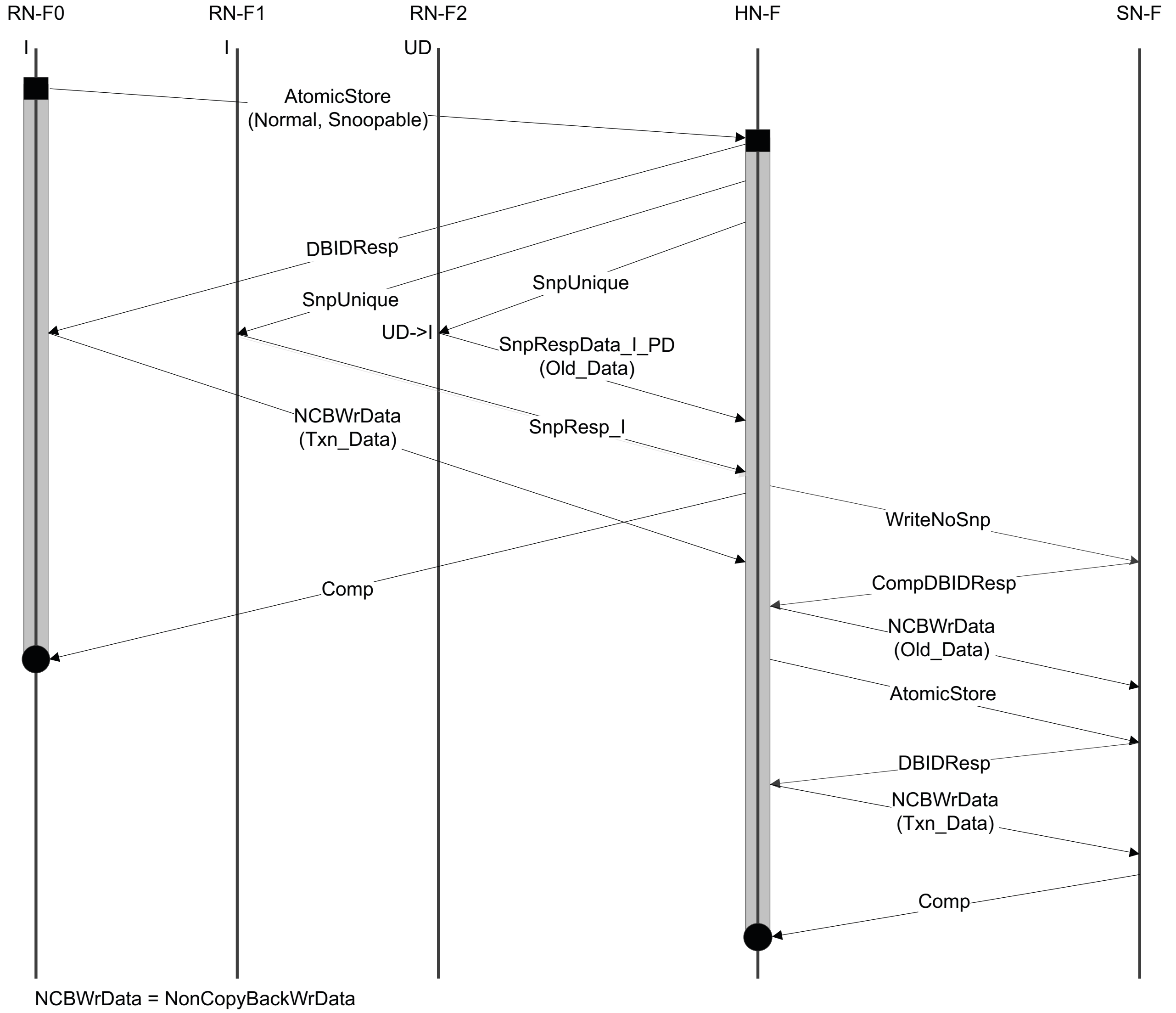


Figure 5-22 AtomicStore executed at SN-F

5.5 Stash transaction flows

This section shows example interconnect protocol flows for the two Stash transaction types:

- *Write with Stash hint*.
- *Independent Stash request* on page 5-265.

5.5.1 Write with Stash hint

Figure 5-23 on page 5-264 shows an example WriteUniqueStash with Data Pull transaction flow.

1. RN sends a WriteUniqueFullStash request to HN-F with the Stash target identified as RN-F1. Typically, the reqesting RN is an RN-I.
2. HN-F sends SnpMakeInvalidStash to RN-F1 and SnpUnique to RN-F2.
3. RN-F1 and RN-F2 send SnpResp response to HN-F. The Snoop response from RN-F1 also includes a Read request, that is, the Data Pull.
4. HN-F treats the Read request from RN-F1 as a ReadUnique, and sends a combined CompData to RN-F1. CompData response includes the data written by RN.
5. RN-F1 sends CompAck to HN-F to complete the Read transaction.

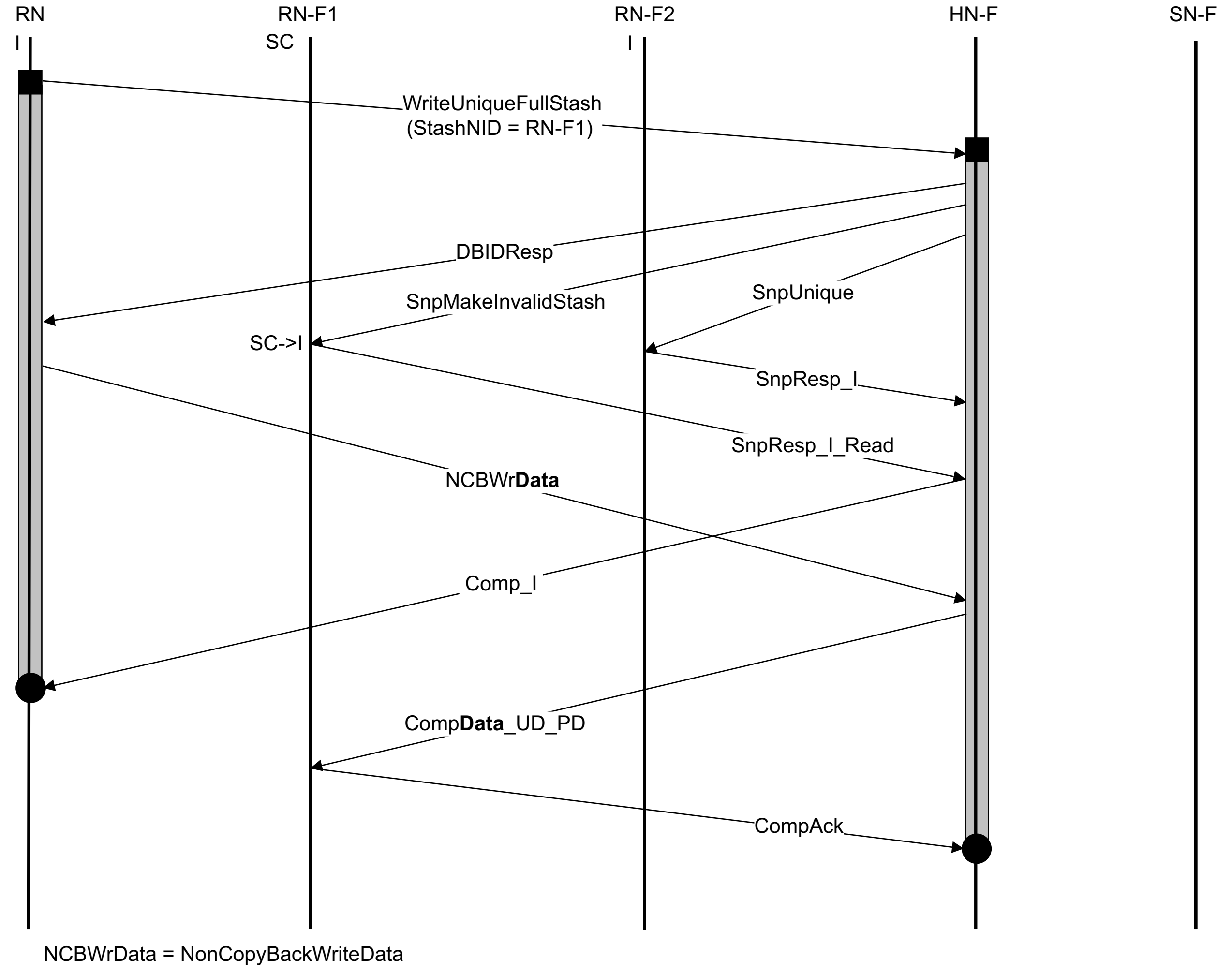


Figure 5-23 WriteUniqueStash with Data Pull

5.5.2 Independent Stash request

Figure 5-24 shows an example StashOnce with Data Pull transaction flow.

1. RN sends a StashOnceShared request to HN-F with the Stash target identified as RN-F1.
2. HN-F sends a Comp response after establishing processing order for the received request that is guaranteeing the request is processed before a request to the same address received later from any Requester.
3. HN-F sends a SnpStashShared snoop to RN-F1, and a ReadNoSnp request to SN-F to fetch Data.
4. RN-F1 sends SnpResp_I_Read response to HN-F.
5. HN-F treats the Read request from RN-F1 as a ReadNotSharedDirty, and sends a combined CompData to RN-F1.
6. RN-F1 sends CompAck to HN-F to complete the Read transaction.

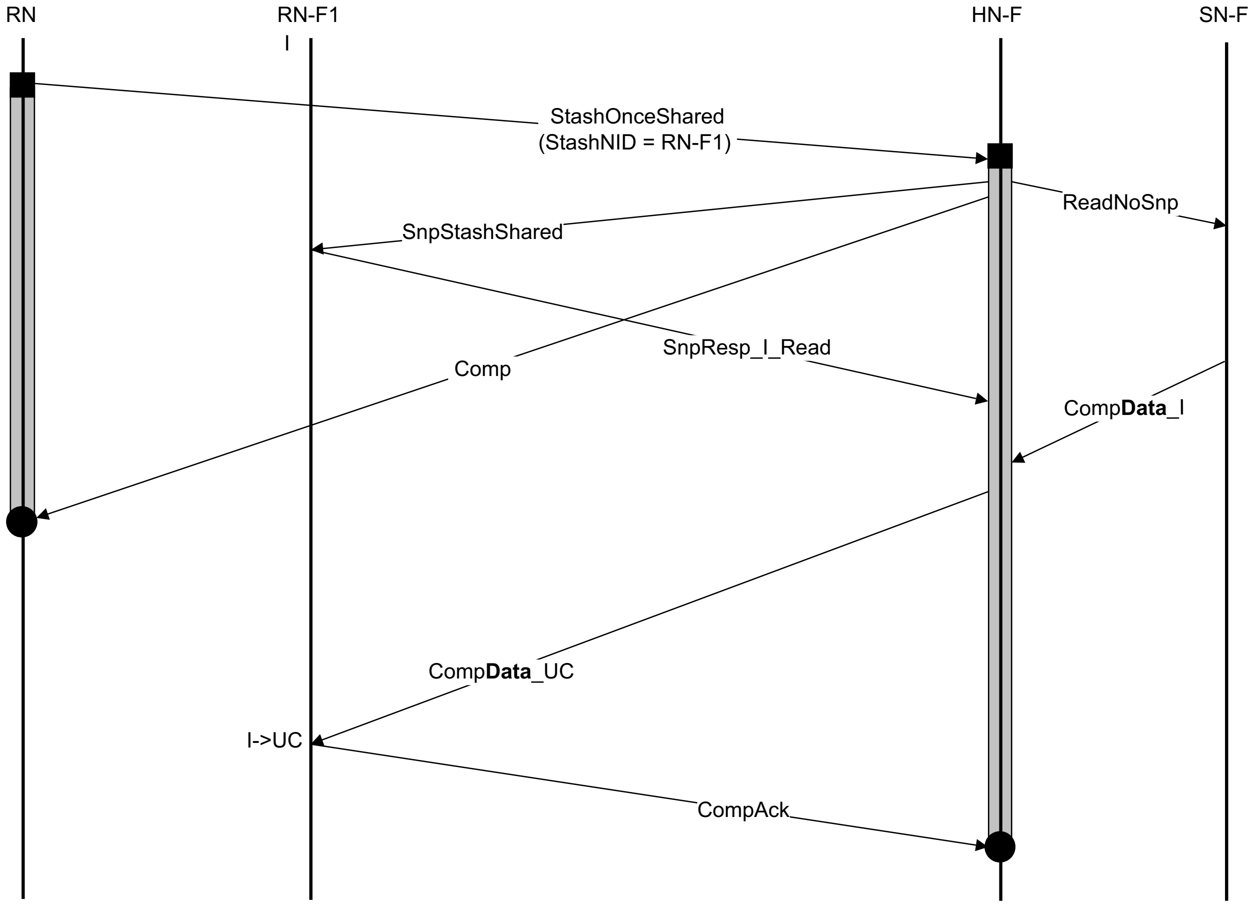


Figure 5-24 StashOnceShared with Data Pull

5.6 Hazard handling examples

This section shows how CopyBack-Snoop request hazard conditions are handled at the Requester and how various request to request, and request to snoop request hazard conditions are handled at the HN-F. It contains the following subsections:

- *CopyBack-Snoop hazard at RN-F*.
- *Request hazard at HN-F* on page 5-269.
- *Read - CopyBack or Dataless - CopyBack hazard at HN-F* on page 5-271.
- *Request-CompAck to HN-F race hazard* on page 5-272.

5.6.1 CopyBack-Snoop hazard at RN-F

Figure 5-25 on page 5-267 shows a Snoop request to an RN-F hazarding a pending CopyBack request at time C. The steps required to resolve this hazard are:

1. At time C:
    - The SnpShared transaction ignores the hazard and reads the cache line data.
    - The cache line state is changed from UD to SC.
2. At time D:
    - The CompDBIDResp for the CopyBack is sent to RN-F0.
    - RN-F0 sends back a CopyBackWrData_SC response.
    - The cache line state is changed from SC to I.

    The data is clean for coherence and is not required to be sent to the interconnect for correct functionality. However, the protocol requires the CopyBack flow to be consistent irrespective of a snoop hazard.

    The cache line state in the WriteData response is SC because that is the state of the cache line when the WriteData response is sent.

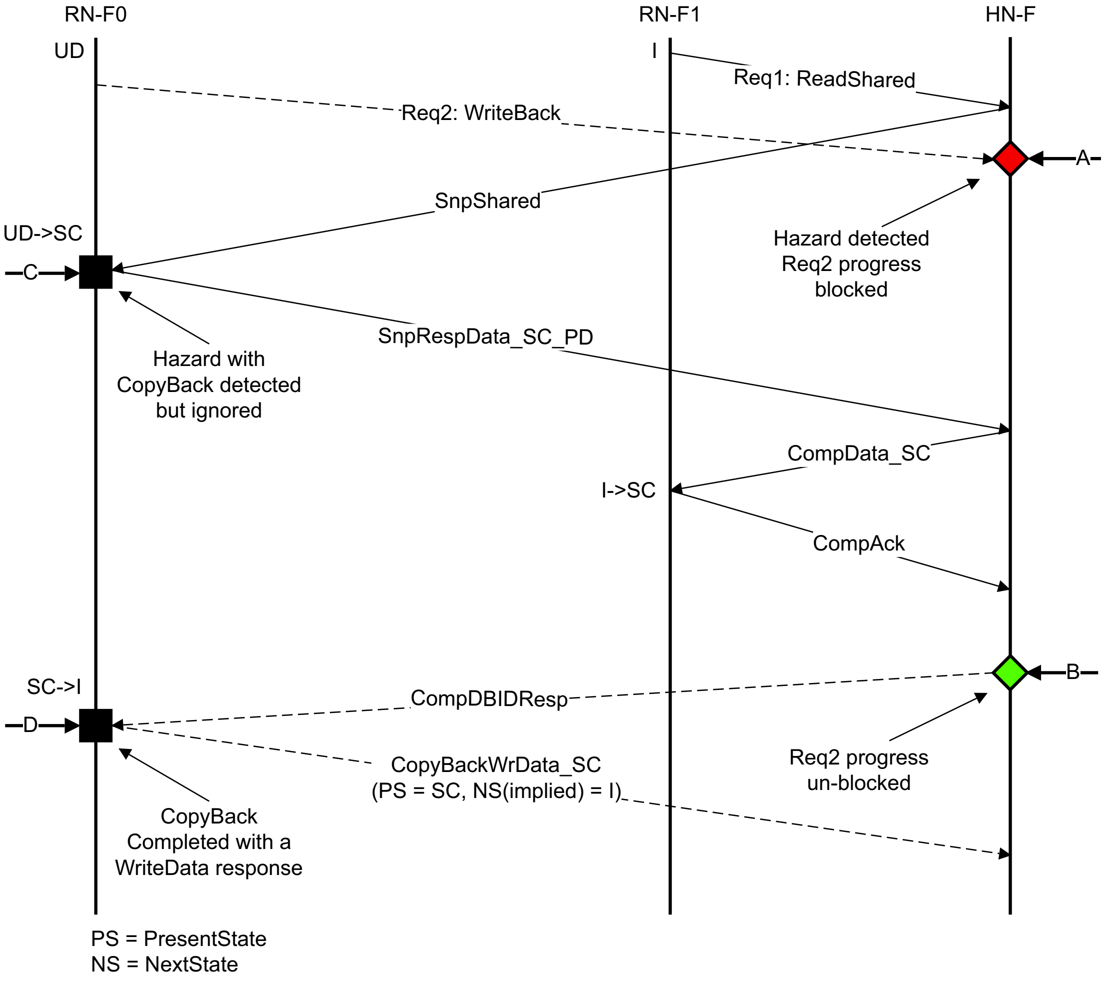


Figure 5-25 CopyBack-Snoop hazard at RN-F example

Note

- The response to a snoop request that hazards with an outstanding Evict must be SnpResp_I.
- During the period between receiving a snoop request and sending a snoop response, including data if applicable, while a CopyBack request to the same address is pending, the only response that can be received for the CopyBack request is a RetryAck.

Figure 5-26 on page 5-268 shows a further example of a snoop request hazarding with an outstanding CopyBack request. In this example, the snoop request is a SnpOnce request generated as a result of a ReadOnce request from RN-F1. The SnpOnce request receives a copy of the data with the snoop response but does not change the cache line state. In this case, the final data response from RN-F0 indicates that the data is Dirty and that HN-F must write the data back to memory.

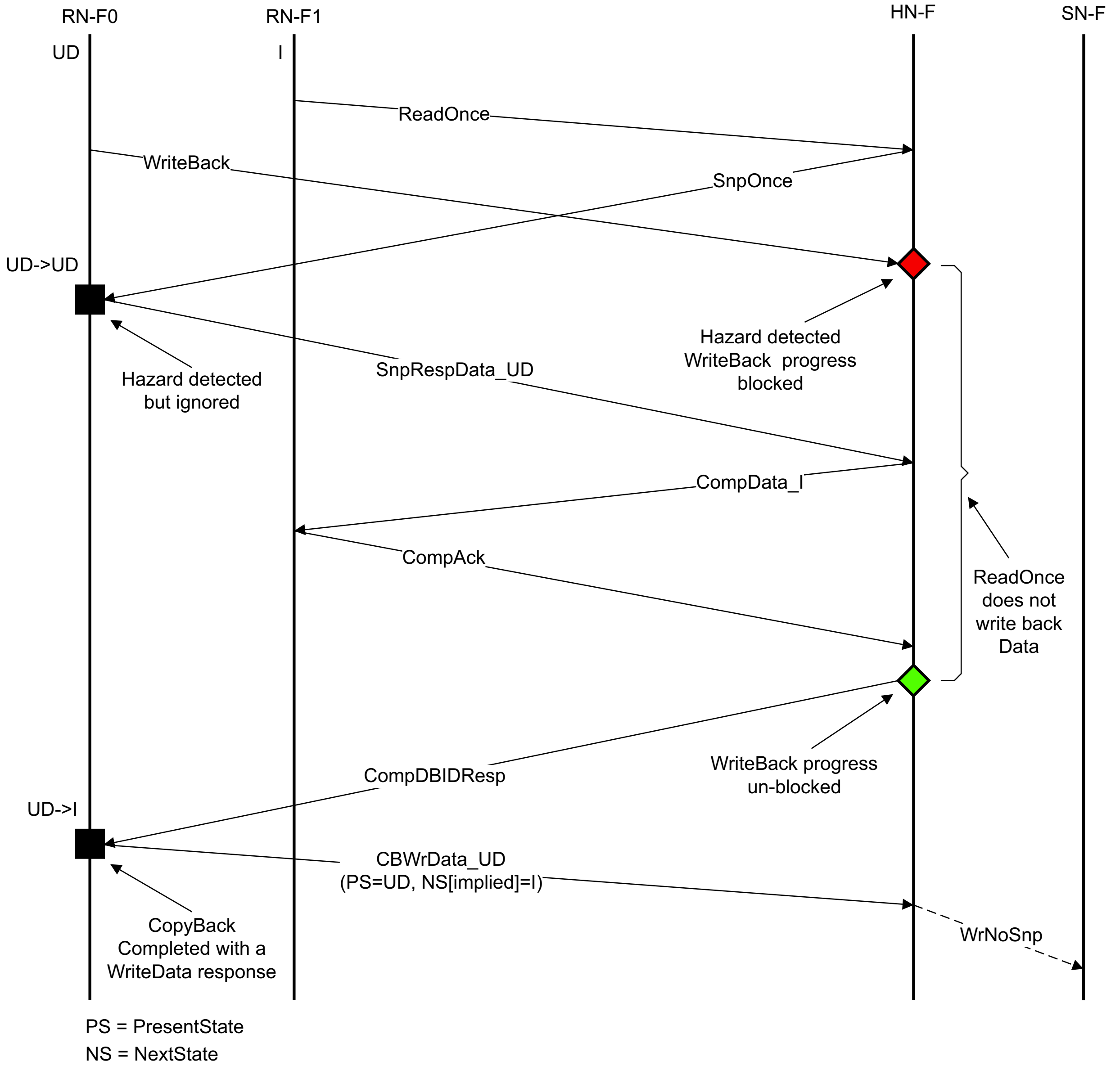


Figure 5-26 CopyBack-Snoop hazard with no cache state change example

5.6.2 Request hazard at HN-F

If more than one request to the same cache line is ready to be processed at the HN-F, then the HN-F can select the next request in any order, except for when the two requests have an ordering requirement and are from the same source, then the order of processing must match the order of arrival.

Figure 5-27 on page 5-270 shows an example where a ReadShared and a ReadUnique, for the same cache line, arrive at the HN-F at approximately the same time. The steps required to resolve this hazard are:

1. At time A:
   - ReadUnique from RN-F0 arrives and hazards a ReadShared request from RN-F2 for which the HN-F has already sent snoop requests.
   - ReadUnique progress is blocked at the HN-F.
2. At time B:
   - The HN-F has completed the ReadShared transaction request from RN-F2.
   - The ReadShared transaction is considered to be complete and the HN-F unblocks the ReadUnique transaction request from RN-F0.

With the exception of ReadNoSnp, the flows will be similar if the two transactions, that Figure 5-27 on page 5-270 shows, are replaced by any Read request type, or Dataless request type:

- A Read transaction request without DMT or DCT or separate Comp and Data response is completed at the HN-F when both of the following are true:
  - — All CompData is sent and, if applicable, CompAck is received. A CompAck is only required for transactions that assert ExpCompAck in the original Request message.
  - — A memory update is completed if required.

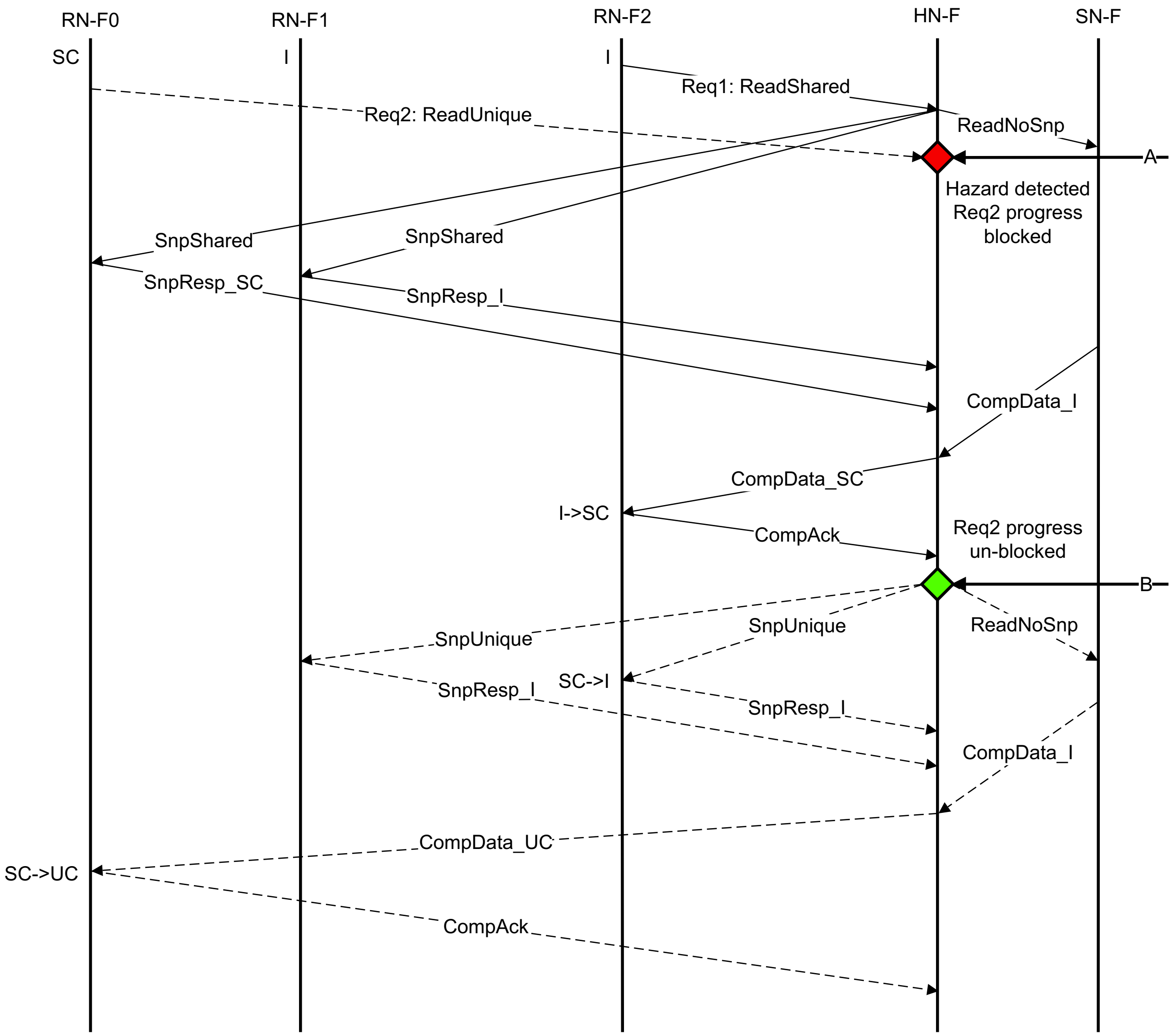


Figure 5-27 Read-Read request hazard example

5.6.3 Read - CopyBack or Dataless - CopyBack hazard at HN-F

A hazard between a Read or Dataless request and a CopyBack request at the HN-F is treated similarly to the Read-Read hazard described in *Request hazard at HN-F* on page 5-269. See also *CopyBack-Snoop hazard at RN-F* on page 5-266.

Figure 5-28 on page 5-272 shows the case where a ReadShared and a WriteBack, for the same cache line, arrive at the HN-F at approximately the same time. The steps required to resolve this hazard are:

1. At time A:
    - A WriteBack encounters a hazarding condition at the HN-F. The reason for the hazard is a ReadShared transaction that is already in progress.
    - The hazard detection results in the WriteBack being blocked.
    - The ReadShared transaction receives data with the snoop response and needs to update memory in addition to sending the data to the Requester.
2. At time B:
    - The WriteBack is unblocked because the HN-F has sent the Data response to the Requester and a WriteData response to memory for the ReadShared transaction.

If the ReadShared request reaches the HN-F, after the HN-F has started processing the WriteBack request, then the ReadShared request will be blocked until completion of the WriteBack request.

A CopyBack request is completed at HN-F when both of the following are true:

- A Data message corresponding to the CopyBack request is received.
- A memory update is completed if required.

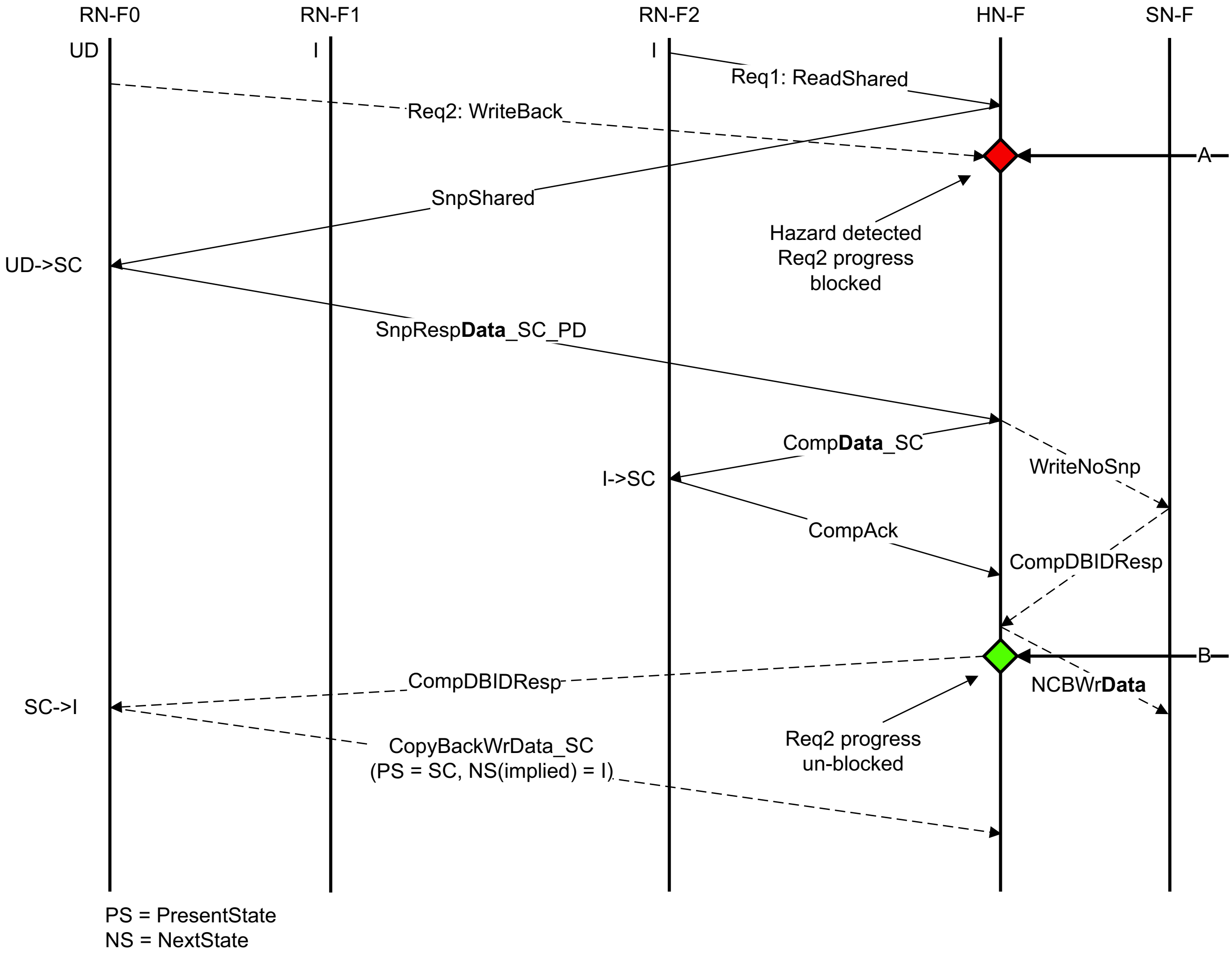


Figure 5-28 Read - CopyBack or Dataless - Copyback hazard example

5.6.4 Request-CompAck to HN-F race hazard

After completion, a request might silently evict the cache line from the cache and generate another request to the same address. For example:

1. The regenerated request reaches the HN-F before the CompAck response associated with the earlier request.
2. The HN-F detects an address hazard and blocks the processing of the new request until the CompAck response is received.

In such a scenario, upon arrival at HN-F, the CompAck response deallocates the previous request from the HN-F and unblocks the processing of the new request.

Chapter 6
Exclusive Accesses

This chapter describes the mechanisms that the architecture includes to support Exclusive accesses. It contains the following sections:

6.1 Overview

The principles of Exclusive accesses are that a *Logical Processor* (LP) 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.

Two different forms of Exclusive access are supported:

- Exclusive accesses to a Snoopable memory location.
- Exclusive accesses to a Non-snoopable memory location.

If the location is updated since the Exclusive Load, by a different LP, then the Exclusive Store must fail. In this case, the store does not occur and the LP does not update the value held at the location.

Note

- The term Exclusive Load is used to describe the action of an LP executing an appropriate program instruction such as LDREX. This action requires:
  - Obtaining the data from the location to which it wants to perform an exclusive sequence.
  - Indicating that it is starting an exclusive sequence.
- The term Exclusive Load transaction is used to describe a transaction issued on the interface to obtain data for an Exclusive Load, if the data is not available in the cache at the LP. Not every Exclusive Load requires an Exclusive Load transaction.
- The term Exclusive Store is used to describe the action of an LP executing an appropriate program instruction such as STREX. This action requires:
  - Determining if the exclusive sequence has passed or failed.
  - If appropriate, updating the data at the location.

  An Exclusive Store can pass or fail and this result is known to the executing processor. When an Exclusive Store passes, the data value at the address location is updated. When an Exclusive Store fails, this indicates that the data value at the address location has not been updated, and the exclusive sequence must be restarted.
- The term Exclusive Store transaction is used to describe a transaction issued on the interface that might be required to complete an Exclusive Store. 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 LP using the transaction response.

6.2 Exclusive monitors

The progress of an exclusive sequence is tracked by an exclusive monitor. The location of the monitor and the request type generated to support the Exclusive accesses is dependent on the memory attributes of the address.

The attributes of Exclusive accesses must be such that they are guaranteed to be observed by an exclusive monitor. For example, if there is a cache between the Master and monitor, then the Exclusive accesses should be Non-cacheable.

6.2.1 Snoopable memory location

For a Snoopable memory location two monitors are defined:

LP monitor Each LP within an RN-F must implement an exclusive monitor that observes the location used by an exclusive sequence. The LP monitor is set when the LP executes an Exclusive Load. The LP monitor is reset when either:

- The location is updated by another LP, which is indicated by an invalidating snoop request to the same address.
- There is a store to the location by the same LP. Resetting of the monitor if the store from the same LP is non-exclusive is IMPLEMENTATION DEFINED.

PoC monitor An HN-F must implement a PoC monitor that can pass or fail an Exclusive Store transaction. A pass indicates that the transaction has been propagated to other coherent RN-Fs. A fail indicates that the transaction has not been propagated to other coherent RN-Fs and therefore the Exclusive Store cannot pass.

The monitor is used to ensure that an Exclusive Store transaction from an LP is only successful if that LP could not have received a snoop transaction, relating to an Exclusive Store to the same address from another LP, after it issued its own Exclusive Store transaction.

The minimum requirement of the PoC monitor is to record when any LP performs a Snoopable transaction related to an exclusive sequence.

If an LP has performed a transaction related to an exclusive sequence, and it then performs an Exclusive Store transaction before a successful Exclusive Store transaction from another LP is scheduled, then the Exclusive Store transaction must be successful.

The monitor must support the parallel monitoring of all exclusive-capable LPs in the system.

When the HN-F receives a transaction associated with an Exclusive Load or an Exclusive Store, the monitor registers that the LP is attempting an exclusive sequence.

When the HN-F receives an Exclusive Store transaction:

- If the PoC monitor has registered that the LP is performing an exclusive sequence, that is, it has not been reset by an Exclusive Store transaction from another LP, then the Exclusive Store transaction is successful and is permitted to proceed. In such a case, registered attempts of all other LPs must be reset. This specification recommends, but does not require, that the PoC monitor for the successful LP is left as registered.
- If the PoC monitor has not registered that the LP is performing an exclusive sequence, that is, it has been reset by an Exclusive Store from another LP, then the Exclusive Store transaction is failed and is not permitted to proceed. The monitor must register that the LP is attempting an exclusive sequence.

Note

A successful Exclusive Store transaction from an LP does not have to reset that the LP is performing an exclusive sequence. The LP can continue to perform a sequence of Exclusive Store transactions, which will all be successful, until another LP performs a successful Exclusive Store transaction. For store transactions in which the LP is not identifiable, the store must be handled as from a different LP than the one which set the monitor.

From initial system reset, the first LP to perform an Exclusive Store transaction can be successful, but this specification does not require it. At that point, all other LPs must then register the start of their exclusive sequence for their Exclusive Store transaction to be successful.

When an Exclusive Store transaction from one LP passes and the registered attempts of all other LPs is reset, the other LPs can only register a new exclusive sequence after the CompAck response is observed for the Exclusive Store transaction that passed.

Note

An LP and PoC monitor pair are required to support an Exclusive access to a Snoopable memory location.

6.2.2 Additional address comparison

The PoC 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 LP's Exclusive Store transaction to a different address location. The number of bits of address comparison used is IMPLEMENTATION DEFINED.

Where an 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 Exclusive Store transaction from another LP to a matching address.

A monitor that includes additional address comparison must still include a minimum monitor of a single bit for every Exclusive-capable LP to ensure forward progress.

An Exclusive Store transaction is permitted to progress if one of the following occurs:

- The address monitor has registered an exclusive sequence for a matching address from the same LP and has not been reset by an Exclusive Store transaction from a different LP with a *matching address*.
- The minimum single-bit monitor has been set by an exclusive sequence from the same LP, and it has not been reset by an Exclusive Store transaction from a different LP to any address.

Note

- The term *matching address* is used to describe where a monitor only records a subset of address bits. The address bits that are recorded are identical, but the address bits that are not recorded can be different.
- An implementation does not require an address monitor for each Exclusive-capable LP. 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. For example, additional address monitors can be used on a first-come first-served basis, or by allocation to particular LPs. Alternatively, a more complex algorithm might be implemented.
- Additional PoC 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 forward progress independently of the progress of Non-secure accesses.

6.2.3 Alternatives to a PoC monitor

An HN-F is permitted to use the following mechanisms instead of a PoC monitor to determine the results of an Exclusive access:

- A precise Snoop Filter, to track if the Requester at the time of Exclusive Store processing retains a copy of the cache line.
- Snooping by the Home Node, to determine if the Requester still holds a copy of the cache line.

Note

Prior to CHI Issue E, a Home Node that supports Exclusive accesses was required to implement a PoC monitor.

6.2.4 Non-snoopable memory location

For a Non-snoopable memory location a single monitor is used:

System monitor The System monitor tracks Exclusive accesses to a Non-snoopable region. This monitor type is set by a ReadNoSnp(Excl) transaction and reset by an update to the location by another LP.

System monitors can be placed at a PoS or at endpoint devices. Potentially, the number of devices in the system is much larger than the number of PoS and placing System monitors at a PoS can:

- Reduce System monitor duplication.
- Reduce the time taken for the system to detect failure of an Exclusive access.

A System monitor must be located so it can observe all transactions to the monitored location.

6.3 Exclusive transactions

The following transaction types support Exclusive accesses through an Excl bit:

- Exclusive Load transaction to a Snoopable location:
  - ReadClean.
  - ReadNotSharedDirty.
  - ReadShared.
  - ReadPreferUnique.
- Exclusive Store transaction to a Snoopable location:
  - CleanUnique.
  - MakeReadUnique.
- Exclusive Load transaction to a Non-snoopable location:
  - ReadNoSnp.
- Exclusive Store transaction to a Non-snoopable location:
  - WriteNoSnp.

The communicating node pairs are:

- For Exclusives to a Snoopable location:
  - RN-F to ICN(HN-F).
- For Exclusives to a Non-snoopable location:
  - RN-F, RN-D, RN-I to ICN(HN-F, HN-I).
  - ICN(HN-F) to SN-F.
  - ICN(HN-I) to SN-I.

An exclusive transaction must use the correct LPID value, See *Logical Processor Identifier* on page 2-115.

Exclusive reads must not use Direct data transfer flow.

6.3.1 Responses to exclusive requests

Transaction responses to exclusive requests are similar to the normal responses to reads and writes with the following exceptions for ReadClean, ReadNotSharedDirty and ReadShared Exclusive transactions:

- Exclusive reads must not use separate Comp and Data response.
- Exclusive reads that do not fail must not use either DMT nor DCT.

However, the response for ReadClean, ReadNotSharedDirty and ReadShared Exclusive transactions must also indicate if the exclusive request has passed or failed. The RespErr field in the response is used for this purpose. See *RespErr* on page 13-425. The RespErr field value of `0b01`, Exclusive Okay, indicates a pass and a RespErr field value of `0b00`, Normal Okay, indicates an Exclusive access failure.

The Exclusive Okay response must only be given for a transaction that has the Excl attribute set.

Not all memory locations are required to support Exclusive accesses. An Exclusive Load transaction to a location that does not support Exclusive accesses must not be given an Exclusive Okay response.

Whether or not an Exclusive Store transaction to a location that does not support Exclusive accesses will update that location is IMPLEMENTATION DEFINED.

This specification recommends that an Exclusive Store transaction is not performed to a location that does not support Exclusive accesses.

ReadPreferUnique and MakeReadUnique do not use RespErr to determine pass or fail of an Exclusive operation. An Exclusive MakeReadUnique response with Shared cache state indicates the failure of an Exclusive access. When the response cache state is Unique, the Requester must use its Local Monitor state to determine if the Exclusive access is a pass.

Table 6-1 shows the Snoopable attributes of the request, the relevant monitor type and possible reasons for fail conditions and response requirements.

Table 6-1 Responses to an Exclusive access request

Request type	Snoopable	Monitor type	Fail condition	Response
ReadNoSnp(Excl)	No	System	Target does not support Exclusive accesses	Target must return a data response
WriteNoSnp(Excl)	No	System	Address content modified	The Requester must still complete the write flow by sending the Data message
			Address not present due to monitor overflow	
			Target does not support Exclusive accesses	
ReadClean(Excl) ReadNotSharedDirty(Excl) ReadShared(Excl)	Yes	LP, PoC	Target does not support Exclusive accesses	Target must return a data response
ReadPreferUnique	Yes	LP, Optional PoC	None	If a PoC monitor is available, the appropriate monitor bit must be set
CleanUnique(Excl)	Yes	LP, PoC	Address content modified	Target must return a Comp response
			Address not present due to monitor overflow	
			Target does not support Exclusive accesses	
MakeReadUnique(Excl)	Yes	LP, Optional PoC	Address content modified	Target uses PoC monitor, Precise Snoop Filter, or SnpQuery to determine response

MakeReadUnique(Excl)

When performing an Exclusive Store operation, MakeReadUnique(Excl) is the preferred alternative over the CleanUnique(Excl) request.

The permitted responses to the MakeReadUnique(Excl) request are included in Table 4-17 on page 4-209 and Table 4-18 on page 4-210.

Result of an Exclusive access

The result of an Exclusive access is determined in the following manner:

- RespErr value of ExOK is not permitted in response to a MakeReadUnique(Excl).
- The Requester must use its local exclusive monitor along with the cache state in the response to determine the success of an Exclusive Store.
  - When the cache state in the response is shared, the Requester must assume that the Exclusive access has failed.
  - When the cache state in the response is Unique, the Requester must use its own local monitor to determine the result of the Exclusive access.

Home behavior

If the Home, after checking the PoC monitor, determines the Exclusive Store passes, it must invalidate all other cached copies of the cache line and then send a Comp response with the cache state of Unique to the Requester. The cache state can include [_PD] if a cached copy in SD state was invalidated and the responsibility of writing back of the Dirty data is being passed to the Requester.

To avoid an Exclusive access livelock, the monitor must be capable of simultaneously tracking each individual LP Exclusive thread in the system.

This specification permits a Home to use a precise snoop filter to determine the success or failure of an Exclusive access. If the snoop filter indicates that the Requester has lost the cache line, then it must be assumed that the Exclusive access has failed.

In the absence of precise caching information from the snoop filter, the Home can use the SnpQuery snoop to determine the presence and state of the cache line at the Requester.

When a Home determines that an Exclusive Store transaction has failed, the following rules must be followed:

- If the Requester has lost the cache line, then the Home is expected to send SnpPreferUniqueFwd or SnpPreferUnique to get a copy of the cache line. The Home is permitted to send SnpNotSharedDirty(Fwd), or SnpClean(Fwd), or SnpShared instead. The snoop must not be SnpSharedFwd nor any invalidating snoop.
- The sender of the data is permitted to return, as appropriate, a UC or UD state in the response to the Requester if no other cached copies exist.

  The responder must return data with SC state, if other copies exist.

  SD state is not permitted in the response to the Requester. This is because the Home cannot determine from the MakeReadUnique(Excl) request whether the Requester will accept data in SD state.

The only case where data is provided in response to a MakeReadUnique(Excl) transaction is when a cache line is lost to a snoop between the issuing of the MakeReadUnique(Excl) transaction and the point at which the Home actions the transaction.

6.3.2 System responsibilities

A system that implements the CHI protocol has the following responsibilities:

- Should include a monitor per LP for the efficient handling of Exclusive accesses.
- Must have a starvation prevention mechanism for all exclusive requests, whether using the monitor mechanism or some other means.
- This specification recommends that progress on Secure Exclusive requests is independent of progress on Non-secure Exclusive requests.

6.3.3 Exclusive accesses to Snoopable locations

This section describes the behavior of an LP when performing Exclusive accesses to a Snoopable address location.

Snoopable Exclusive Load

The LP starts an exclusive sequence with an Exclusive Load. The start of the exclusive sequence must set the LP exclusive monitor.

An LP wanting to perform an Exclusive access to a Snoopable location might already hold the cache line in its local cache:

- If the LP holds the cache line in a Unique state, then it is permitted, but not recommended by this specification, that it performs an Exclusive Load transaction.
- If the LP holds the cache line in a Shared state, then it is permitted, but not required by this specification, that it performs an Exclusive Load transaction.
- If the LP does not hold a copy of the cache line, this specification recommends that the LP uses an Exclusive Load transaction to obtain the cache line, but is permitted to use ReadClean, ReadShared, ReadNotSharedDirty, or ReadPreferUnique without the Excl attribute asserted.

Snoopable Exclusive Load to Snoopable Exclusive Store

After the execution of an Exclusive Load an LP will typically calculate a new value to store to the location before it attempts the Exclusive Store.

It is not required that an LP always completes an exclusive sequence. For example, the value obtained by the Exclusive Load can indicate that a semaphore is held by another LP and that the value cannot be changed until the semaphore is released by the other LP. Therefore, a new exclusive sequence can be started with no attempt to complete the current exclusive sequence.

During the time between the Exclusive Load and the Exclusive Store the LP exclusive monitor must monitor the location to determine whether another LP might have updated the location.

Snoopable Exclusive Store

An LP 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 LP must wait for all messages for any such transaction to be exchanged, or to receive a RetryAck response, before issuing an Exclusive Store transaction. The transactions that register that an LP is performing an exclusive sequence are:

- Exclusive Load transactions to any location.
- Exclusive Store transactions to any location.

When an LP executes an Exclusive Store the following behavior is required:

- If the LP exclusive monitor has been reset the Exclusive Store must fail and the LP must not issue an Exclusive Store transaction. The LP must restart the exclusive sequence.

  Note

  When the LP monitor has been reset, not issuing a transaction for an Exclusive Store that must eventually fail avoids unnecessary invalidation of other copies of the cache line.

- If the cache line is held in a Unique state and the LP exclusive monitor is set then the Exclusive Store has passed and it can update the location without issuing a transaction.

- If the cache line is held in a Shared state and the LP exclusive monitor is set then the LP must issue an Exclusive Store transaction. A CleanUnique or MakeReadUnique transaction with the Excl attribute asserted must be used. The LP exclusive monitor must continue to operate and check that the cache line is not updated while the CleanUnique or MakeReadUnique transaction is in progress.

  An Exclusive CleanUnique transaction response is handled in the following way.

  The transaction will receive a Normal Okay or an Exclusive Okay response.

  If the transaction receives an Exclusive Okay response, then this indicates that the transaction has passed and has completed invalidating all other copies of the cache line. After an exclusive transaction completes with an Exclusive Okay response, the LP must again check the LP exclusive monitor:

  — If the LP exclusive monitor is set then the Exclusive Store has passed and the update is performed.

  — If the LP exclusive monitor is not set, it indicates that an update to the cache line has occurred 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 LP has not been able to track the exclusive nature of the cache line, because the cache line has been evicted, then the Exclusive Store must fail and the exclusive sequence must be restarted.

  If the Exclusive Store transaction receives a Normal Okay response then this indicates another LP has been permitted to progress a transaction associated with an Exclusive Store. The transaction associated with the Exclusive Store, from this LP, has failed and has not propagated to other LPs in the system. When an Exclusive Store transaction completes with a Normal Okay response, the options are:

  — The LP can fail the Exclusive Store and restart the exclusive sequence with or without checking the state of the cache line when the access completed.

  — The LP can check the LP exclusive monitor, and if the LP exclusive monitor has been reset, then the LP must fail the Exclusive Store and restart the exclusive sequence.

  — The LP can check the LP exclusive monitor, and if the LP exclusive monitor is set, then the LP can repeat the Exclusive Store transaction.

For handling of an Exclusive MakeReadUnique at the Home Node see *Home behavior* on page 6-280.

Exclusive accesses to Non-snoopable locations

The following restrictions apply to Exclusive accesses to Non-snoopable locations:

- The address of an Exclusive access must be aligned to the total number of bytes in the transaction.
- The number of bytes to be transferred in an Exclusive access must be a legal data transfer size, that is, 1, 2, 4, 8, 16, 32, or 64 bytes.

Failure to observe these restrictions results in behavior that is UNPREDICTABLE.

For Exclusive read and Exclusive write transactions to be considered a pair, the following criteria must apply:

- The addresses of the Exclusive read and the Exclusive write must be identical.
- The value of the control signals, that is MemAttr and SnpAttr of the Exclusive read and the Exclusive write transaction, must be identical.
- The data size in the Exclusive read and the Exclusive write must be identical.
- The LPID value of the Exclusive read must match the LPID value of the Exclusive write transaction.

The minimum number of bytes to be monitored during an exclusive operation is defined by the transaction size. The System monitor can monitor a larger number of bytes, up to 64, 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 while the Exclusive access was in progress.

Multiple Exclusive transactions to Non-snoopable memory locations, either read or write, to the same or different addresses, from the same LP must not be outstanding at the same time.

If the SN does not support Exclusive accesses, as indicated by an Exclusive Fail on the Exclusive ReadNoSnp, then the write will update the location if the write is given an Exclusive Fail response.

If the SN does support Exclusive accesses, as indicated by an Exclusive Pass on the Exclusive ReadNoSnp, then the write will not update the location if the write is given an Exclusive Fail response.

Chapter 7
Cache Stashing

This chapter describes the cache stashing mechanism whereby data that is written from an RN can be installed in a peer cache. It contains the following sections:

7.1 Overview

Cache stashing is a mechanism to install data within particular caches in a system. Cache stashing ensures that data is located close to its point of use, therefore improving the system performance.

Cache stashing is permitted to Snoopable memory only.

This specification supports two main forms of cache stashing transaction:

Write with stash hint

WriteUniqueStash. This is used when the cache in which the data should be allocated is known at the point in time that the data is written. A write with stash hint can be a [Full] or [Ptl] cache line write, and this will affect the Snoop transactions that are used. See *Write with Stash hint* on page 7-288.

Independent stash request

StashOnce. This is used when the request to stash data into a particular cache is separated from the writing of the data. An independent Stash transaction can indicate if the cache line should be held in a Unique or Shared state by using a StashOnceUnique or StashOnceSepUnique, or a StashOnceShared or StashOnceSepShared transaction respectively, which corresponds to whether the next expected use of the cache line is for storing or for reading. See *Independent Stash request* on page 7-289.

Both forms of cache stashing can target installation of data at different cache levels. The Stash target cache can be a peer cache, specified by using the peer cache target NodeID, or a logical processor cache within the peer node, if the peer node has multiple logical processors. The logical processor is identified by the LPID in the target cache field. See *Stash target identifiers* on page 7-291.

The cache stashing requests can also target the cache below the peer cache in the cache hierarchy, which can be an interconnect cache or a system cache. This is done by not specifying the peer cache NodeID. See *Stash target not specified* on page 7-291.

In all cases of cache stashing, the stashing is only a performance hint and it is permitted for the Stash request receiver to not perform the stashing behavior.

7.1.1 Snoop requests and Data Pull

The following Snoop requests are used to notify a peer cache that it is the target of a Stash request:

- SnpUniqueStash.
- SnpMakeInvalidStash.
- SnpStashUnique.
- SnpStashShared.

Table 7-1 shows the Snoop requests associated with each of the Stash requests.

Table 7-1 Stash request and the corresponding Snoop request

Stash request	Snoop request
WriteUniquePtlStash	SnpUniqueStash
WriteUniqueFullStash	SnpMakeInvalidStash
StashOnceUnique StashOnceSepUnique	SnpStashUnique
StashOnceShared StashOnceSepShared	SnpStashShared

A Snoopee that receives a Stash type Snoop request does one of the following:

- Provides a Snoop response that also acts as a Read request for the associated cache line. Including a Read request with Snoop response is referred to as a Data Pull. Table 7-2 shows the type of Read request that is implied by a Data Pull in the response to each Stash type Snoop request.

Table 7-2 Snoop response with Data Pull and implied Read request

Snoop request	Implied Read request
SnpUniqueStash	ReadUnique
SnpMakeInvalidStash	ReadUnique
SnpStashUnique	ReadUnique
SnpStashShared	ReadNotSharedDirty

- Provides a Snoop response without a Data Pull response so ignoring the cache stash hint.

The value of the DataPull field in the SnpResp and SnpRespData responses indicates if Data Pull is requested. See *DataPull* on page 13-419 for legal values for DataPull.

The use of Data Pull to complete a Snoop request with Stash is optional.

If the Snoopee is not able to support the Data Pull transaction flow then it is permitted to ignore the stash operation.

7.2 Write with Stash hint

The rules for sending and processing a WriteUniqueFullStash and WriteUniquePtlStash request at the Stash requester, the Home, and the Stash target node are as follows:

Requester responsibilities:

- Sends a WriteUniqueFullStash or WriteUniquePtlStash request depending on whether a full cache line or a partial cache line is to be written.
- The request is expected to include a Stash target.

Home responsibilities:

- Permitted to send a RetryAck response to a WriteUniqueStash request and follow the Retry transaction flow.
- Sends SnpUniqueStash to the identified Stash target.
- Sends SnpUnique to all other Requesters that share the cache line.
- Permitted to send SnpMakeInvalidStash and SnpMakeInvalid instead of SnpUniqueStash and SnpUnique respectively for WriteUniqueFullStash.
- Send Comp to the Requester after the coherency action is completed.
- Permitted to ignore the stash hint in the Write request and process the request as a regular WriteUnique.
- Handles a request without a Stash target in the manner described in *Stash target not specified* on page 7-291.
- Permitted to use DMT to get data from SN-F to the Stash target in response to a Data Pull request, when the data is neither available at Home nor obtained from any caches.
- Permitted to use separate Non-data and Data-only response to the Stash target in response to a Data Pull request.

The Stash target responsibilities:

- The responses that include Data Pull are:
  - SnpResp_I_Read.
  - SnpRespData_I_Read.
  - SnpRespData_I_PD_Read.
  - SnpRespDataPtl_I_PD_Read.
- Must not request Data Pull if:
  - Snoop has an address hazard with an outstanding request.
  - The Stash target has an outstanding request to the same address that has received a DBIDRespOrd but has not completed.
- When requesting Data Pull:
  - The Stash target must guarantee the Read data is accepted without any structural or protocol dependencies that might result in deadlock.
  - The Read request is treated by Home as ReadUnique.
  - The Stash target must populate the DBID field in the response with the TxnID that is to be used by Home for the Read transaction. If the Snoop response with Data Pull includes data, then the DBID field value in all data packets must be the same.
- Permitted to ignore the Stash hint and handle the snoop as SnpUnique.

7.3 Independent Stash request

The second mechanism for implementing cache stashing is to permit the Stash request to be sent separated in time from the writing of Stash data. Examples of when such a mechanism is useful are:

- When the data that is being written is not required by the target immediately. This delayed stash avoids polluting the cache with data that is not used immediately.
- When the data is already in the system and the data has to be prefetched into caches.
- When the process using the data being written is not scheduled when the data is written, and therefore the precise target of the Stash data is not known until later.

In these cases, a Requester can use StashOnce or StashOnceSep requests to request Home or a peer node to fetch a cache line.

The rules for sending and processing an independent Stash request at the Stash requester, Home, and the Stash target are as follows:

Requester Node responsibilities:

- Sends StashOnce*Unique to Home if the stashed cache line is to be modified.
- Sends StashOnce*Shared to Home if the stashed cache line is not to be modified.
- Sends StashOnceSep only if the Requester is capable of handling the StashDone response.
- The StashOnce* requests provide a Stash target when the data is to be stashed in a peer cache.
- The StashOnce* requests do not provide a Stash target when the data is to be allocated to the next level cache.

Home Node responsibilities:

- Permitted to send a RetryAck response to a Stash request and follow the Retry transaction flow.
- Send a SnpStashUnique to the target RN-F for StashOnce*Unique.
- Send a SnpStashShared to the target RN-F for StashOnce*Shared.
- Permitted to not send a Snoop request in response to a Stash request.
- Must send a Comp response, even if it abandons the Stash request.
- For the StashOnce, the Home must send a Comp only after establishing processing order for the received request that is guaranteeing that any request to the same address received later from any Requester is ordered behind this request. The Comp response must come from the PoC.
- For the StashOnceSep request, the Home must send a Comp only after establishing the guarantee that it will not send a RetryAck response. The StashDone response must only be sent after establishing the guarantee that the request is ordered at the Home.
- Fetches the addressed cache line from memory into the shared system cache when a Stash request without a Stash target is received.
- Permitted to send Comp after receiving the Stash request, and before sending any SnpStash or receiving the Snoop response.
- Send Comp with a non-I state, if the cache line is cached in the cache at the next level.

  Send Comp_I if the cache line is not cached, or it is not known if the cache line is cached, in the cache at the next level.
- Send a Comp_I response if either the cache look up at Home is a miss or Home did not look up the cache before responding.
- Permitted to use DMT to get data from SN-F to the Stash target in response to a Data Pull request.

- Permitted to use separate Non-data and Data-only response to the Stash target in response to a Data Pull request.

Stash target responsibilities:

- The snoop must not change the state of the cache line at the Stash target.
- The snoop is treated as a hint at the Stash target to obtain a copy of the cache line.
- Must not request Data Pull if:
  - Snoop has an address hazard with an outstanding request.
  - Response is sent before performing a local cache lookup.
  - The snoop is SnpStashShared and the cache has a copy of the cache line.
  - The Stash target has an outstanding request to the same address that has received a DBIDRespOrd but not yet completed.
- When requesting Data Pull:
  - The Stash target must guarantee the Read data is accepted without any structural or protocol dependencies that might result in deadlock.
  - The DataPull request is treated by Home as ReadNotSharedDirty for SnpOnceShared.
  - The DataPull request is treated by Home as ReadUnique for SnpOnceUnique.
  - The Home must treat the Stash request and the DataPull request atomically as a single request. That is, the Home must not order any other request to the same address between the Stash request and the corresponding DataPull request.
  - The Stash target must populate the DBID field in the response with the TxnID that is to be used by Home for the Read transaction.
- A DataPull request can be sent, but is not required to be sent, when the snoop is SnpStashUnique and a shared copy is present.
- The Stash target is permitted, but not required, to wait until it completes the local cache lookup before sending the Snoop response.
- The cache state in the Snoop response is not required to be precise:
  - An imprecise response must be SnpResp_I.
  - Any state other than I in the response must be precise.

Note

- For StashOnce*, care is needed 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 StashOnce*Unique transaction can cause the invalidation of a copy of the cache line and care must be taken to ensure such transactions do not interfere with Exclusive access sequences.

The requirements regarding the sending of Stash type snoops from a Home, and the permitted responses by the target, are the same as the request/response rules for StashOnceSep transactions. See *StashOnce and StashOnceSep transactions* on page 2-59.

7.4 Stash target identifiers

For all Stash requests, both options of specified and non-specified Stash target are supported.

7.4.1 Stash target specified

If the Stash target is available in the Stash request then Home sends the snoop with a stash hint to the specified target. The specified target can be an RN or a logical processor within an RN.

7.4.2 Stash target not specified

The Home Node that receives a WriteUniquePtlStash or WriteUniqueFullStash request without a Stash target does the following:

- If the cache line is cached in a Unique state at an RN, then Home can treat that RN as the Stash target.
- If the cache line is not cached in a Unique state then Home must only send SnpUnique as required, and must not send SnpUniqueStash to any RN.
- For WriteUniquePtlStash, if the cache line is not in any cache then this specification recommends Home to prefetch and allocate the cache line in the system cache. It is permitted, but not recommended, to perform a partial write to main memory.
- For WriteUniqueFullStash, if the cache line is not in any cache then Home is permitted to allocate the cache line in the shared system cache.

The Home Node that receives a StashOnce or StashOnceSep request without a Stash target does the following:

- If the cache line is not cached in any peer cache then this specification recommends that the cache line is allocated in the shared system cache.
- If the cache line is cached in a peer cache then it is IMPLEMENTATION DEFINED if a snoop is sent to transfer a copy of the cache line and allocate it in the shared system cache. For StashOnceUnique and StashOnceSepUnique it is also IMPLEMENTATION DEFINED if all cached copies are invalidated before allocating the cache line in the shared system cache.

7.5 Stash messages

Stash messages are classified as:

- Write requests:
  - WriteUniqueFullStash.
  - WriteUniquePtlStash.

  See *Write transactions* on page 4-173.
- Dataless requests:
  - StashOnceUnique.
  - StashOnceSepUnique.
  - StashOnceShared.
  - StashOnceSepShared.

  See *Dataless transactions* on page 4-170.
- Snoop requests:
  - SnpUniqueStash.
  - SnpMakeInvalidStash.
  - SnpStashUnique.
  - SnpStashShared.

  See *Snoop request types* on page 4-184.
- Stash responses:
  - Comp.
  - StashDone.
  - CompStashDone.

  See *StashOnce and StashOnceSep transactions* on page 2-59.

7.5.1 Supporting REQ packet fields

The fields defined in the REQ packet to support Stash requests are:

- StashNID, StashLPID.
- StashNIDValid, StashLPIDValid.

See *Protocol flit fields* on page 13-404.

7.5.2 Supporting SNP packet fields

The fields defined in the SNP packet to support Stash requests are:

- StashLPID.
- StashLPIDValid.

See *Protocol flit fields* on page 13-404.

7.5.3 Supporting RSP packet field

The field defined in the RSP packet to support Stash requests is:

- DataPull.

See *Protocol flit fields* on page 13-404.

7.5.4 Supporting DAT packet fields

The field defined in the DAT packet to support Stash requests is:

- DataPull.

See *Protocol flit fields* on page 13-404.

Chapter 8
DVM Operations

This chapter describes *Distributed Virtual Memory* (DVM) operations that the protocol uses to manage virtual memory. It contains the following sections:

- *DVM transaction flow* on page 8-294.
- *DVM Operation types* on page 8-305.
- *DVM Operations* on page 8-309.

8.1 DVM transaction flow

All DVM transactions have similar requirements and are mapped to a single flow. The following sections show the Non-sync and Sync type DVM transaction requirements:

- *Non-sync type DVM transaction flow*.
- *Sync type DVM transaction flow* on page 8-296.
- *Flow control* on page 8-297.
- *DVMOp field value restrictions* on page 8-299.
- *Field value requirements* on page 8-304.

8.1.1 Non-sync type DVM transaction flow

Figure 8-1 shows the steps in a Non-sync type DVM transaction.

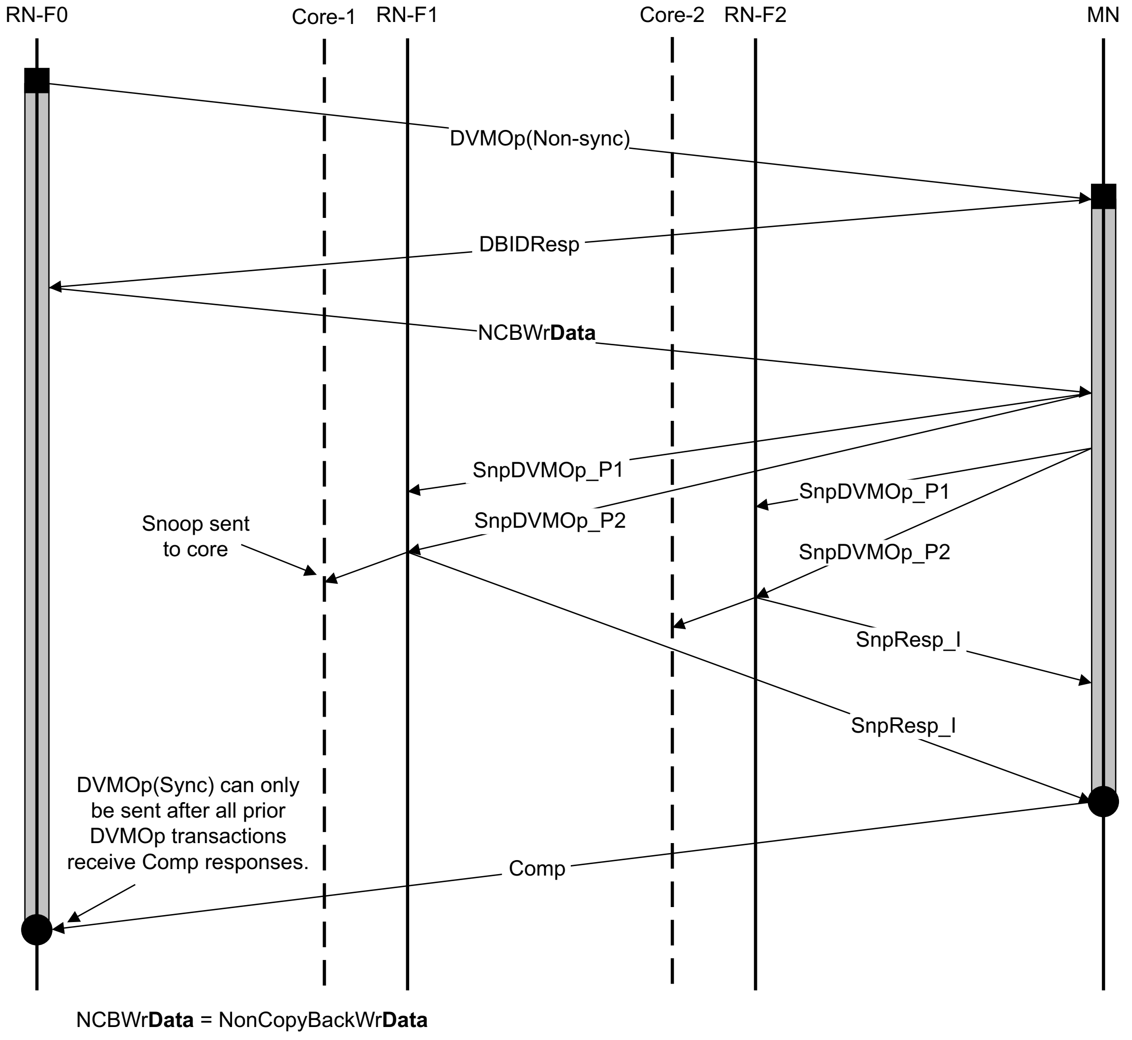


Figure 8-1 Non-sync type DVM transaction flow

The required steps that Figure 8-1 on page 8-294 shows are:

1. RN-F0 sends a DVMOp(Non-sync) to the MN using the appropriate write semantics for the DVMOp type.

2. The MN accepts the DVMOp(Non-sync) request and provides a DBIDResp response.

3. The RN-F0 sends an 8-byte data packet on the data channel.

4. The MN broadcasts the SnpDVMOp snoop request to all the RN-F and RN-D nodes in the system. The SnpDVMOp is sent on the snoop channel, and requires two snoop requests. The two parts of the SnpDVMOp are labeled by the suffix _P1 and _P2.

   Note

   - Both parts of the message must carry the same *Transaction ID* (TxnID).
   - RN must have resources available to accept the SnpDVMOp. See *Flow control* on page 8-297.

5. After completing the required actions, each recipient of the SnpDVMOp sends a single SnpResp response to the MN.

   Note

   Sending of a SnpResp implies that the target RN has forwarded the SnpDVMOp to the required RN structures and has freed up the resources needed to accept another DVM operation. It does not imply that the requested DVM operation has completed. See *Sync type DVM transaction flow* on page 8-296.

6. After receiving all the SnpResp responses, the MN sends a Comp response to the requesting node.

DVM early Comp for Non-sync DVMOps

This specification permits the Misc Node (MN) in the interconnect to send Comp for a Non-sync DVMOp without waiting to complete the required snooping of RNs. Such an MN must take the responsibility of ordering this Non-sync DVMOp against any Sync DVMOp received later from the same source. If an MN cannot provide such an ordering guarantee, then it must wait for the snooping to be completed before sending a Comp response for a Non-sync DVMOp.

An MN that is enabled to send early Comp for a Non-sync DVMOp is permitted to opportunistically combine Comp and DBIDResp responses into a single CompDBIDResp response.

Note

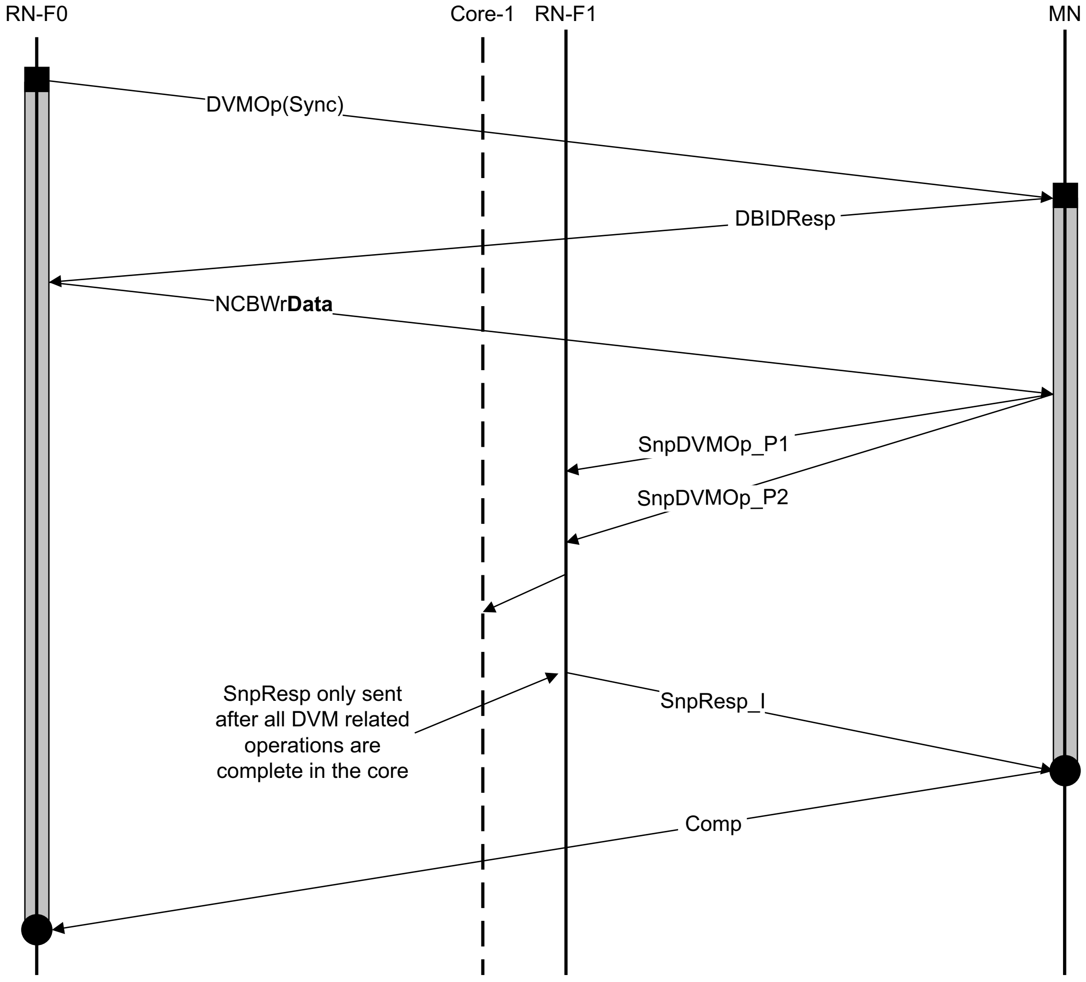
- Sending of a Comp response early for Non-sync DVMOp reduces round-trip latency for DVMOp completion. This enables a greater number of DVMOp transactions to be pipelined from a single source.

- Such an early completion also enables a Sync DVMOp, which is waiting for completions of all related DVMOp transactions sent earlier from the same RN.

The MN must still wait for the Snoop response for a Sync DVMOp to be received before sending a Comp to the Requester for that Sync DVMOp.

8.1.2 Sync type DVM transaction flow

Figure 8-2 shows the flow in a Sync type DVM transaction.



NCBWr**Data** = NonCopyBackWr**Data**

Figure 8-2 Sync type DVM transaction flow

The required steps that Figure 8-2 on page 8-296 shows are:

1. RN-F0 sends a DVMOp(Sync) to the MN.

   Note

   All previous DVMOp requests whose completion needs to be guaranteed by the DVMOp(Sync) must have received a Comp response before the RN can send a DVMOp(Sync).

2. The MN accepts the DVMOp(Sync) request and sends a DBIDResp response to the Requester.
3. The RN-F0 sends a data packet on the data channel with a data size of 8 bytes.
4. The MN sends the SnpDVMOp to RN-F1. The SnpDVMOp is sent on the snoop channel, and requires two snoop requests. The two parts of a SnpDVMOp are labeled by the suffix _P1 and _P2.
5. After completing the DVM Sync operation, RN-F1 sends a SnpResp response to the MN.

   Note

   Sending of a SnpResp implies that all DVM related operations have completed at the RN structures and the target RN has freed up the resources needed to accept another SnpDVMOp.

6. After receiving the SnpResp, the MN sends a Comp response to RN-F0.

8.1.3 Flow control

DVMOp request flow control:

- A DVMOp can receive a RetryAck response from an MN.
- A DVMOp that receives a RetryAck response must wait for a PCrdGrant response from the MN that has the appropriate PCrdType.
- All previous DVMOp requests whose completion needs to be guaranteed by the DVMOp(Sync) must have received a Comp response before the RN can send the DVMOp(Sync).
- The interconnect must guarantee forward progress on DVMOp(Non-Sync), which requires that there should be at least one tracker entry in MN reserved for DVMOp(Non-Sync).
- It is permitted to overlap a DVMOp(Non-sync) and a DVMOp(Sync), from the same RN, if the DVMOp(Sync) is not required to guarantee completion of the DMVOp(Non-sync).

SnpDVMOp flow control:

- Each SnpDVMOp transaction requires two SnpDVMOp request packets.
- Both SnpDVMOp request packets corresponding to a single transaction must use the same TxnID.
- The two SnpDVMOp request packets corresponding to a single transaction can be sent or received in any order.
- Multiple SnpDVMOp(Non-sync) transactions can be outstanding from an MN.
- Only one SnpDVMOp(Sync) transaction can be outstanding from an MN to an RN.
- To prevent deadlocks, due to the two part SnpDVMOp requests that uses the snoop channel, a SnpDVMOp transaction must only be sent when the receiving RN has pre-allocated resources to accept both parts of the SnpDVMOp transaction.
- A Sync DVM operation at an RN that has issued a StashOnceSep request must wait until all outstanding StashDone responses are received for StashOnceSep requests that use invalidated page entries.
- An RN must provide a response to a SnpDVMOp transaction only after it has received both SnpDVMOp request packets corresponding to that transaction.

- An RN must provide a response to a SnpDVMOp only when it can accept a further SnpDVMOp from an MN.
- Each RN-F and RN-D in the system specifies the number of SnpDVMOp transactions it can accept concurrently.
- Each RN-F and RN-D in the system must be able to accept at least one SnpDVMOp(Non-Sync) transaction in addition to a SnpDVMOp(Sync) transaction.
- The minimum number of SnpDVMOp transactions that must be accepted concurrently is two. This is the default number for RNs that do not specify a number.

8.1.4 DVMOp field value restrictions

Field value restrictions during a DVMOp transaction are shown:

- Request messages in Table 8-1
- Data messages in Table 8-2 on page 8-300
- Snoop messages in Table 8-3 on page 8-301
- Response messages:
  - DBIDResp in Table 8-4 on page 8-302
  - SnpResp in Table 8-5 on page 8-303
  - Comp in Table 8-6 on page 8-304

Table 8-1 shows the Request message field value restrictions for a DVMOp transaction.

Table 8-1 Request message field value restrictions for DVMOp

Field Name	Restriction
QoS	None. Can be any value.
TgtID	Expected to be node ID of MN. Can be remapped to correct TgtID by the interconnect.
SrcID	Source ID of the Requester that initiated the DVM message.
TxnID	An ID generated by the Requester. Must follow the same rules as any other transaction.
ReturnNID StashNID SLCRepHint	Must be all zeros.
StashNIDValid Endian Deep	Must be zero.
ReturnTxnID StashLPIDValid StashLPID	Must be all zeros.
Opcode	Must be DVMOp.
Size	Must be 8-byte.
Addr	Req_Addr_Width. See *DVM Operations* on page 8-309
NS	Must be zero.
LikelyShared	Must be zero.
AllowRetry	Can be any value, because a DVMOp can be given a Retry.
Order	Must be all zeros
PCrdType	Must be all zeros if AllowRetry is asserted, otherwise the Credit type value.
MemAttr	Must be all zeros.
SnpAttr	None. Can be any value. See *DVM domain* on page 8-321.

Table 8-1 Request message field value restrictions for DVMOp (continued)

Field Name	Restriction
LPID	None. Don't Care.
GroupIDExt	Must be all zeros.
Excl SnoopMe	Must be zero.
ExpCompAck	Must be zero.
TagOp	Must be zero.
TraceTag	None.
MPAM	Must be all zeros.
RSVDC	None. Don't Care.

Table 8-2 shows the Data message field value restrictions for a DVMOp transaction.

Table 8-2 Data message field value restrictions for DVMOp

Field Name	Restriction
QoS	None. Can be any value.
TgtID	Must be the same as SrcID returned in the DBIDResp response.
SrcID	Must be ID of the original Requester.
TxnID	Must be the same as DBID of the DBIDResp response.
HomeNID	Must be all zeros.
Opcode	Must be NonCopyBackWriteData.
RespErr	Must be `0b00` or `0b10`.
Resp	Must be all zeros.
FwdState DataPull DataSource	Must be all zeros.
CBusy	Must be all zeros.
DBID	None. Don't Care.
CCID	Must be all zeros.
DataID	Must be all zeros.
TagOp	Must be all zeros.
Tag	Must be zero.
TU	Must be zero.
TraceTag	None.
RSVDC	None.

Table 8-2 Data message field value restrictions for DVMOp (continued)

Field Name	Restriction
BE	Only BE[7:0] must be asserted.
Data	Unused bits must be zero for Data[63:0] and Data[n:64] = Don't Care.
DataCheck	Must be the appropriate value for the Data field.
Poison	None. Can take any value.

Table 8-3 shows the Snoop message field (SnpDVMOp) value restrictions during a DVMOp transaction.

Table 8-3 Snoop message field value restrictions for DVMOp

Field Name	Restriction
QoS	None. Can be any value.
SrcID	Must be node ID of MN.
TxnID	An ID generated by MN.
FwdNID	None. Used as Range and Num[4:0] fields. See Table 8-29 on page 8-321
VMIDExt	Must be used for VMID extension.
Opcode	Must be SnpDVMOp.
Addr	(Req_Addr_Width - 3). See *DVM Operations* on page 8-309.
NS	Must be zero.
DoNotGoToSD	Must be zero.
RetToSrc	Must be zero.
TraceTag	None.
MPAM	Must be all zeros.

Table 8-4 shows the Response DBIDResp message field value restrictions during a DVMOp transaction.

Table 8-4 Response DBIDResp message field value restrictions for DVMOp

Field Name	Restriction
QoS	None. Can be any value.
TgtID	Must be ID of the original Requester.
SrcID	Must be ID of the MN that is handling DVMs.
TxnID	Must match TxnID of the original request.
Opcode	Must be DBIDResp.
RespErr	Must be all zeros.
Resp	Must be all zeros.
FwdState DataPull	Must be all zeros.
CBusy	Expected to be populated by the Completer.
DBID TagGroupID StashGroupID PGroupID	Generated by the MN that is handling DVMOps.
PCrdType	Must be all zeros.
TagOp	Must be all zeros.
TraceTag	None.

Table 8-5 shows the Response SnpResp message field value restrictions during a DVMOp transaction.

Table 8-5 Response SnpResp message field value restrictions for DVMOp

Field Name	**Restriction**
QoS	None. Can be any value.
TgtID	Must be ID of the MN that is handling DVMOps.
SrcID	Must be ID of the node that is responding to the snoop.
TxnID	Must match the TxnID of the SnpDVMOp snoop request.
Opcode	Must be SnpResp.
RespErr	Must be all zeros.
Resp	Must be all zeros.
FwdState DataPull	Must be all zeros.
CBusy	Expected to be populated by the Completer.
DBID TagGroupID StashGroupID PGroupID	None. Don't Care.
PCrdType	Must be all zeros.
TagOp	Must be all zeros.
TraceTag	None.

Table 8-6 shows the Response Comp and CompDBIDResp message field value restrictions during a DVMOp transaction.

Table 8-6 Response Comp and CompDBIDResp message field value restrictions for DVMOp

Field Name	Restriction
QoS	None. Can be any value.
TgtID	Must be ID of the original Requester.
SrcID	Must be ID of the MN that is handling DVMs.
TxnID	Must match TxnID of the original request.
Opcode	Must be Comp.
RespErr	Must be `0b00`, `0b10`, or `0b11`.
Resp	Must be all zeros.
FwdState DataPull	Must be all zeros.
CBusy	Expected to be populated by the Completer.
DBID TagGroupID StashGroupID PGroupID	Generated by the MN that is handling DVMs.
PCrdType	Must be all zeros.
TagOp	Must be all zeros.
TraceTag	None.

8.1.5 Field value requirements

Both SnpDVMOp request packets, corresponding to a single DVMOp, must have the same value in the following fields:

- TxnID.
- Opcode.
- SrcID.
- TgtID.

8.2 DVM Operation types

The following DVM Operations are supported:

- TLB Invalidate.
- Branch Predictor Invalidate.
- Instruction Cache Invalidate:
  - Physical address invalidate.
  - Virtual address invalidate.
- Synchronization.

8.2.1 DVMOp payload

The payload of a DVM operation from the RN to the MN is distributed within:

- The address field in the DVM request from the RN.
- The lower 8 bytes of write data in the NonCopyBackWrData packet.

The payload of a DVM operation from the MN to the RN is distributed over two SnpDVMOp request packets using the address fields.

The various fields in the payload and their encodings are shown in Table 8-7.

Table 8-7 DVMOp fields and encodings

Field	Bits	Function
VA Valid	1	`0b1` indicates that the specified address is valid.
VMID Valid	1	`0b1` indicates that the *Virtual Machine IDentifier* (VMID) or *Virtual Index* (VI) is valid.
ASID Valid	1	`0b1` indicates that the *Address Space IDentifier* (ASID) or VI is valid.
Security	2	Indicates that the transaction applies to: `0b00` Secure and Non-secure `0b01` Non-secure address from a Secure context `0b10` Secure `0b11` Non-secure
Exception Level	2	Indicates that the transaction applies to: `0b00` Hypervisor and all Guest OS `0b01` EL3[a] `0b10` Guest OS `0b11` Hypervisor
DVMOp type	3	Indicates the DVM operation type as: `0b000` TLB Invalidate `0b001` Branch Predictor Invalidate `0b010` Physical Instruction Cache Invalidate `0b011` Virtual Instruction Cache Invalidate `0b100` Synchronization `0b101-0b111` Reserved
VMID	8	Virtual Machine ID VMID[7:0] or Virtual Index VA[27:20]
ASID	16	Address Space ID or Virtual Index VA[19:12][b]

Table 8-7 DVMOp fields and encodings (continued)

Field	Bits	Function
S2-S1 Staged Invalidation	2	Indicates Stage 2 or Stage 1 invalidation: `0b00` • DVMv7: Any transaction. • DVMv8: Both Stage 1 and Stage 2 invalidation. `0b01` Stage 1 invalidation only. `0b10` Stage 2 invalidation only. `0b11` Reserved.
Leaf Entry Invalidation	1	`0b1` indicates that only leaf level translation invalidation is required.
Range[c]	1	TLBI operations that include address range are applicable to all old and new TLBI operations that are an invalidation by VA or IPA. Range can take any value. When: **Range = 1** The transaction is a range-based TLBI operation. **Range = 0** The transaction is a non-range based TBLI operation. Range is inapplicable and must be zero for non-TLBI DVM transactions.
Num[c]	5	Used as a constant multiplication factor in the range calculation. All binary values are valid.
Scale[c]	2	Used as a constant in address range exponent calculation. All binary values are valid.
Translation Table Level[c]	2	Hint of Translation Table Level, which includes the addresses to be invalidated: `b00` The entries in the range can be using any level for the translation table entries. `b01` All entries to invalidate are Level1 translation table entries. `b10` All entries to invalidate are Level2 translation table entries. `b11` All entries to invalidate are Level3 translation table entries.
Translation Granule Size[c]	2	Translation Granule Size takes the following values: `0b00` Reserved `0b01` 4K `0b10` 16K `0b11` 64K
BaseAddr or VA or PA	37 49-53 44-52	Range base address Virtual address Physical address
VMIDExt	8	Virtual Machine ID VMID[15:8]

a. DVMv8 only.

b. When used as Virtual Index, the upper 8 bits of ASID are Don't Care.

c. Inapplicable and must be set to zero in non-TLBI DVM Operations. See *Range based TLBI* on page 8-319 and *Level Hint in TBLI Operations* on page 8-321 for details of how these fields are used in TLBI operations.

8.2.2 DVMOp and SnpDVMOp packet

Table 8-8 shows the distribution of the payload in the DVMOp request from the RN, using 8-byte write semantics, and the distribution of the payload in the SnpDVMOp requests from the MN.

In the DVMOp, the combination of the address field in the request and the 8-byte write data transports the complete payload. Addr[3] is not used in the request and must be set to zero.

In the two SnpDVMOp requests the combination of the two address fields transports the complete payload. Addr[3] is used in a SnpDVMOp request to indicate which part of the payload is being transported.

The valid combinations of *Maximum PA* (MPA) and *Maximum VA* (MVA) address bits are:

- MPA = 44 : MVA = 49.
- MPA = 45 : MVA = 51.
- MPA = 46 to 52 : MVA = 53.

Note

In Table 8-8, the number given shows which Address or Data bit is replaced by the DVMOp field.

For example, the VA Valid field is placed in the same position that Addr[4] normally occupies. In a Request packet, this would be the fifth bit position in the Addr field, but in a Snoop packet it would be the second bit position because the Snoop packet does not include the three least significant address bits.

Also, PA[6] is placed in the same position that Data[4] normally occupies in a write data packet, and in the same position that Addr[4] normally occupies in a Snoop packet. PA[6] is provided in the Part 2 Snoop packet, while VA Valid is provided in the Part 1 Snoop packet.

Table 8-8 DVMOp and SnpDVMOp request payloads

Field	Bits	Request	Data	Snoop		Notes
		Addr	Data	Addr		
				Part 1	Part 2	
Part Num	1	[3]	-	[3]	[3]	Must be `0b0` for the Request and Snoop Part 1. Must be `0b1` for Snoop Part 2.
VA Valid	1	[4]	-	[4]	-	-
VMID Valid	1	[5]	-	[5]	-	-
ASID Valid	1	[6]	-	[6]	-	-
Security	2	[8:7]	-	[8:7]	-	-
Exception Level	2	[10:9]	-	[10:9]	-	-
DVMOp type	3	[13:11]	-	[13:11]	-	-
VMID[7:0]	8	[21:14]	-	[21:14]	-	For VMID[15:8], see below.
ASID	16	[37:22]	-	[37:22]	-	-
S2-S1 Staged Invalidation	2	[39:38]	-	[39:38]	-	-
Leaf Entry Invalidation	1	[40]	-	[40]	-	-
PA[(MPA-1):6]	(MPA-6)	-	[(MPA-3):4]	-	[(MPA-3):4]	-

Table 8-8 DVMOp and SnpDVMOp request payloads (continued)

Field	Bits	Request	Data	Snoop		Notes
		Addr	Data	Addr		
				Part 1	Part 2	
VA bits when REQ Addr width = 44 bits						
VA[45:6]	40	-	[43:4]	-	[43:4]	Either VA or PA are used not both.
VA[48:46]	3	-	[46:44]	[43:41]	-	
Additional VA bits where REQ Addr = 45 bits						
VA[49]	1	-	[47]	-	[44]	Either VA or PA are used not both.
VA[50]	1	-	[48]	[44]	-	
Additional VA bits where REQ Addr = 46-52 bits						
VA[51]	1	-	[49]	-	[45]	Either VA or PA are used not both.
VA[52]	1	-	[50]	[45]	-	
VMID[15:8]	8	-	[63:56]	VMIDExt	-	VMIDExt is a separate field outside of the Addr field. See Table 8-7 on page 8-305.

8.3 DVM Operations

This section describes the supported DVM Operations:

- *TLB Invalidate* on page 8-310.
- *Branch Predictor Invalidate* on page 8-316.
- *Physical Instruction Cache Invalidate* on page 8-317.
- *Virtual Instruction Cache Invalidate* on page 8-318.
- *Synchronization* on page 8-322.

Table 8-9 shows the values for the Part Num field in all supported DVM Operations.

Table 8-9 Part Num field values

Addr		**Value**			**Status**
Bit	**Field**	**Request**	**Snoop**		
			Part 1	**Part 2**	
[3]	Part Num	`0b0`	`0b0`	`0b1`	Not utilized in the Request

8.3.1 TLB Invalidate

This section shows the supported TLB Invalidate operations.

- *TLB Invalidate, All Guest OS, Secure operations*
- *TLB Invalidate, All Guest OS, Non-secure IPA, Secure operations* on page 8-311.
- *TLB Invalidate, All Guest OS, Non-secure operations* on page 8-312
- *TLB Invalidate, Hypervisor, Secure operations* on page 8-313
- *TLB Invalidate, Hypervisor, Non-secure operations* on page 8-314
- *TLB Invalidate EL3, Secure operations* on page 8-315

TLB Invalidate, All Guest OS, Secure operations

Table 8-10 shows the fixed value fields for TLB Invalidate, All Guest OS, Secure operations.

Table 8-10 TLB Invalidate, All Guest OS, Secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp Type	`0b000`	TLBI
[10:9]	Exception Level	`0b10`	All Guest OS
[8:7]	Secure	`0b10`	Secure

Table 8-11 shows the TLB Invalidate values for All Guest OS, Secure operations.

In Table 8-11, the VA field holds the IPA or PA as required.

Table 8-11 TLB Invalidate, All Guest OS, Secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b0`	Secure TLB Invalidate all
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b1`	Secure TLB Invalidate by VA
`0b0`	`0b0`	`0b1`	`0b00`[a]	`0b1`	Secure TLB Invalidate by VA Leaf Entry only
`0b1`	`0b0`	`0b0`	`0b00`[a]	`0b0`	Secure TLB Invalidate by ASID
`0b1`	`0b0`	`0b0`	`0b00`[a]	`0b1`	Secure TLB Invalidate by ASID and VA
`0b1`	`0b0`	`0b1`	`0b00`[a]	`0b1`	Secure TLB Invalidate by ASID and VA Leaf Entry only
`0b0`	`0b1`	`0b0`	`0b01`	`0b0`	Secure Guest OS, TLBI all, S1 invalidation only
`0b1`	`0b1`	`0b0`	`0b00`	`0b0`	Secure Guest OS, TLBI by ASID
`0b0`	`0b1`	`0b0`	`0b00`	`0b1`	Secure Guest OS, TLBI by VA
`0b0`	`0b1`	`0b1`	`0b00`	`0b1`	Secure Guest OS, TLBI by VA, Leaf only
`0b1`	`0b1`	`0b0`	`0b00`	`0b1`	Secure Guest OS, TLBI by ASID and VA
`0b1`	`0b1`	`0b1`	`0b00`	`0b1`	Secure Guest OS, TLBI by ASID and VA, Leaf only
`0b0`	`0b1`	`0b0`	`0b10`	`0b1`	Secure Guest OS, TLBI by Secure IPA

Table 8-11 TLB Invalidate, All Guest OS, Secure operations (continued)

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b1`	`0b1`	`0b10`	`0b1`	Secure Guest OS, TLBI by Secure IPA, Leaf only
`0b0`	`0b1`	`0b0`	`0b00`	`0b0`	Secure Guest OS TLBI all

a. All DVMv7 transactions must use `0b00`.

TLB Invalidate, All Guest OS, Non-secure IPA, Secure operations

Table 8-12 shows the fixed value fields for TLB Invalidate, All Guest OS, Non-secure IPA. Secure operations.

Table 8-12 TLB Invalidate, All Guest OS, Non-secure IPA, Secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp	`0b000`	TLBI
[10:9]	Exception Level	`0b10`	All Guest OS
[8:7]	Secure	`0b01`	Non-secure address from a Secure context

Table 8-13 shows the values for TLB Invalidate, All Guest OS, Non-Secure IPA, Secure operations.

In Table 8-13, the VA field holds the IPA.

Table 8-13 TLB Invalidate, All Guest OS, Non-secure IPA, Secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b1`	`0b0`	`0b10`	`0b1`	Secure Guest OS, TLBI by Non-secure IPA
`0b0`	`0b1`	`0b1`	`0b10`	`0b1`	Secure Guest OS, TLBI by Non-secure IPA Leaf only

TLB Invalidate, All Guest OS, Non-secure operations

Table 8-14 shows the fixed value fields for TLB Invalidate, All Guest OS, Non-secure operations.

Table 8-14 TLB Invalidate, All Guest OS, Non-secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp	`0b000`	TLBI
[10:9]	Exception Level	`0b10`	All Guest OS
[8:7]	Secure	`0b11`	Non-Secure

Table 8-15 shows the values for TLB Invalidate, Guest OS, Non-Secure operations.

Table 8-15 TLB Invalidate Guest OS, Non-secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b0`	All Guest OS TLB Invalidate all
`0b0`	`0b1`	`0b0`	`0b01`	`0b0`	Guest OS TLB Invalidate all Stage 1 invalidation only
`0b0`	`0b1`	`0b0`	`0b00`[a]	`0b0`	Guest OS TLB Invalidate all ARMv7 must carry out Stage 1 and Stage 2 invalidation
`0b0`	`0b1`	`0b0`	`0b00`[a]	`0b1`	Guest OS TLB Invalidate by VA
`0b0`	`0b1`	`0b1`	`0b00`[a]	`0b1`	Guest OS TLB Invalidate by VA Leaf Entry only
`0b1`	`0b1`	`0b0`	`0b00`[a]	`0b0`	Guest OS TLB Invalidate by ASID
`0b1`	`0b1`	`0b0`	`0b00`[a]	`0b1`	Guest OS TLB Invalidate by ASID and VA
`0b1`	`0b1`	`0b1`	`0b00`[a]	`0b1`	Guest OS TLB Invalidate by ASID and VA Leaf Entry only
`0b0`	`0b1`	`0b0`	`0b10`	`0b1`[b]	Guest OS TLB Invalidate by IPA
`0b0`	`0b1`	`0b1`	`0b10`	`0b1`[b]	Guest OS TLB Invalidate by IPA Leaf Entry only

a. All DVMv7 transactions must use `0b00`.

b. IPA is the Intermediate Physical Address. The IPA is sent using the same format as the Virtual Address (VA).

TLB Invalidate, Hypervisor, Secure operations

Table 8-16 shows the fixed value fields for TLB Invalidate Hypervisor, Secure operations.

Table 8-16 TLB Invalidate, Hypervisor, Secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp	`0b000`	TLBI
[10:9]	Exception Level	`0b11`	Hypervisor
[8:7]	Secure	`0b10`	Secure

Table 8-17 shows the values for TLB Invalidate, Hypervisor with Secure operations.

Table 8-17 TLB Invalidate, Hypervisor, Secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b0`	`0b0`	`0b00`	`0b0`	Secure Hypervisor, TLBI all
`0b1`	`0b0`	`0b0`	`0b00`	`0b1`	Secure Hypervisor, TLBI by ASID and VA
`0b1`	`0b0`	`0b1`	`0b00`	`0b1`	Secure Hypervisor, TLBI by ASID and VA Leaf only
`0b0`	`0b0`	`0b0`	`0b00`	`0b1`	Secure Hypervisor, TLBI by VA
`0b0`	`0b0`	`0b1`	`0b00`	`0b1`	Secure Hypervisor, TLBI by VA, Leaf only
`0b1`	`0b0`	`0b0`	`0b00`	`0b0`	Secure Hypervisor, TLBI by ASID

TLB Invalidate, Hypervisor, Non-secure operations

Table 8-18 shows the fixed value fields for TLB Invalidate, Hypervisor, Non-secure operations.

Table 8-18 TLB Invalidate, Hypervisor, Non-secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp	`0b000`	TLBI
[10:9]	Exception Level	`0b11`	Hypervisor
[8:7]	Secure	`0b11`	Non-secure

Table 8-19 shows the values for TLB Invalidate, Hypervisor, Non-secure operations.

Table 8-19 TLB Invalidate Hypervisor, Non-secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b0`	Hypervisor TLB Invalidate all
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b1`	Hypervisor TLB Invalidate by VA
`0b0`	`0b0`	`0b1`	`0b00`[a]	`0b1`	Hypervisor TLB Invalidate by VA Leaf Entry only
`0b1`	`0b0`	`0b0`	`0b00`[a]	`0b0`	Hypervisor TLB Invalidate by ASID
`0b1`	`0b0`	`0b0`	`0b00`[a]	`0b1`	Hypervisor TLB Invalidate by ASID and VA
`0b1`	`0b0`	`0b1`	`0b00`[a]	`0b1`	Hypervisor TLB Invalidate by ASID and VA Leaf Entry only

a. All DVMv7 transactions must use 0b00.

TLB Invalidate EL3, Secure operations

Table 8-20 shows the fixed value fields for TLB Invalidate, EL3, Secure operations.

Table 8-20 TLB Invalidate, EL3, Secure operations fixed values

Addr	Field	Value	Status
[13:11]	DVMOp	`0b000`	TLBI
[10:9]	Exception Level	`0b01`	EL3
[8:7]	Secure	`0b10`	Secure

Table 8-21 shows values for TLB Invalidate EL3 with Secure operations.

Table 8-21 TLB Invalidate EL3, Secure operations

Addr					Operation
[6] ASID valid	[5] VMID valid	[40] LEAF	[39:38] S2-S1	[4] VA valid	
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b1`	EL3 TLB Invalidate by VA
`0b0`	`0b0`	`0b1`	`0b00`[a]	`0b1`	EL3 TLB Invalidate by VA Leaf Entry only
`0b0`	`0b0`	`0b0`	`0b00`[a]	`0b0`	EL3 TLB Invalidate All

[a] All DVMv7 transactions must use 0b00.

8.3.2 Branch Predictor Invalidate

This section shows the Branch Predictor Invalidate operations.

Table 8-22 shows the fixed value fields in the Branch Predictor Invalidate operation.

Table 8-22 Branch Predictor Invalidate operation fixed values

Addr		Value	Status
Bits	**Field**		
[3]	Part Num	-	See Table 8-9 on page 8-309
[5]	VMID Valid	`0b0`	VMID field not valid
[6]	ASID Valid	`0b0`	ASID field not valid
[8:7]	Secure	`0b00`	Applies to both secure and Non-secure
[10:9]	Exception Level	`0b00`	Applies to all Guest OS and Hypervisor
[21:14]	VMID VMIDExt	`0xXX`	VMID not specified
[37:22]	ASID	`0xXXXX`	ASID not specified
[39:38]	S2, S1 Staged Invalidation	`0b00`	Reserved, set to Zero
[40]	Leaf Entry Invalidation	`0b0`	Reserved, set to Zero

Note

The use of Branch Predictor Invalidate with a 16-bit ASID is not supported.

Table 8-23 shows the operations supported by Branch Predictor Invalidate.

Table 8-23 Branch Predictor Invalidate operations

Addr		Operation
[13:11] DVMOp type	**[4] VA valid**	
`0b001`	`0b0`	Branch Predictor Invalidate all
	`0b1`	Branch Predictor Invalidate by VA

8.3.3 Physical Instruction Cache Invalidate

This section shows the Physical Instruction Cache Invalidate operations.

Table 8-24 shows the fixed value fields in the Physical Instruction Cache Invalidate operation.

Table 8-24 Physical Instruction Cache Invalidate operation fixed values

Addr		Value	Status
Bits	**Field**		
[3]	Part Num	-	See Table 8-9 on page 8-309
[10:9]	Exception Level	`0b00`	Applies to all Guest OS and Hypervisor
[39:38]	S2, S1 Staged Invalidation	`0b00`	Reserved, set to zero
[40]	Leaf Entry Invalidation	`0b0`	Reserved, set to zero

Table 8-25 shows the operations supported by Physical Instruction Cache Invalidate.

Note

When Virtual Index is `0b11`, then VA[19:12] and VA[27:20], at Addr[29:22] and Addr[21:14] respectively, are used as part of the Physical Address. Addr[37:30] are not used, and are Don't Care values.

Table 8-25 Physical Instruction Cache Invalidate operations

[13:11] DVMOp type	[8:7] Secure	[6:5] Virtual Index	[4] VA	Operation
`0b010`	`0b10`	`0b00`	`0b0`	Secure Physical Address Cache Invalidate all
		`0b00`	`0b1`	Secure Physical Address Cache Invalidate by PA without Virtual Index
		`0b11`	`0b1`	Secure Physical Address Cache Invalidate by PA with Virtual Index
	`0b11`	`0b00`	`0b0`	Non-secure Physical Address Cache Invalidate all
		`0b00`	`0b1`	Non-secure Physical Address Cache Invalidate by PA without Virtual Index
		`0b11`	`0b1`	Non-secure Physical Address Cache Invalidate by PA with Virtual Index

8.3.4 Virtual Instruction Cache Invalidate

This section shows the Virtual Instruction Cache Invalidate operations.

Table 8-26 shows the fixed value fields in the Virtual Instruction Cache Invalidate operation.

Table 8-26 Virtual Instruction Cache Invalidate operation fixed values

Addr		Value	Status
Bits	**Field**		
[3]	Part Num	-	See Table 8-9 on page 8-309
[39:38]	S2, S1 Staged Invalidation	`0b00`	Reserved. set to zero
[40]	Leaf Entry Invalidation	`0b0`	Reserved, set to zero

Table 8-27 shows the operations supported by Virtual Instruction Cache Invalidate.

Table 8-27 Virtual Instruction Cache Invalidate operations

Addr						Operation
[13:11] DVMOp type	**[10:9] Exception Level**	**[8:7] Secure**	**[6] ASID valid**	**[5] VMID valid**	**[4] VA valid**	
`0b011`	`0b00`	`0b00`	`0b0`	`0b0`	`0b0`	Invalidate all. Applies to Secure and Non-secure. Applies to Hypervisor and all Guest OS.
		`0b11`	`0b0`	`0b0`	`0b0`	Invalidate all. Applies to Non-secure. Applies to Hypervisor and all Guest OS.
	`0b10`	`0b10`	`0b1`	`0b0`	`0b1`	Secure Invalidate by ASID and VA.
			`0b0`	`0b1`	`0b0`	Secure Invalidate all per VMID.
			`0b1`	`0b1`	`0b1`	Secure Invalidate by ASID and VA per VMID.
		`0b11`	`0b0`	`0b1`	`0b0`	Non-secure Guest OS, Invalidate all per VMID.
			`0b1`	`0b1`	`0b1`	Non-secure Guest OS, Invalidate by ASID and VA per VMID.
	`0b11`	`0b11`	`0b0`	`0b0`	`0b1`	Hypervisor, Invalidate by VA.
			`0b1`	`0b0`	`0b1`	Hypervisor, Invalidate by ASID and VA

8.3.5 Range based TLBI

TLBI operations that include an address range are included starting with the CHI issue E specification.

These operations are applicable to all old and new TLBI operations that are an invalidation by VA or IPA.

A 1-bit field called 'Range' is added to both DVMOp and SnpDVMOp. Range can take any value. When

Range = 1	The transaction is a range-based TLBI operation.
Range = 0	The transaction is a non-range based TBLI operation.

Range field is inapplicable and must be set to zero in non-TLBI based DVM operations.

Range payload packing

Address range to invalidate is calculated using the following formula:

$$\texttt{Base Addr to Base Addr + ((Num+1) * } 2^{\texttt{5*Scale+1}} \texttt{ * TG)}$$

Where:

BaseAddr	Shifted Base address of the range - 37 bits. The base address in the message is shifted based on the Transaction granule. The lowest address bit carried in the message is Addr[12], Addr[14], and Addr[16] for Transaction Granule size of 4K, 16K, and 64K, respectively.
TG	Translation Granule Size - 2 bits, takes the following values: **0b00** Reserved **0b01** 4K **0b10** 16K **0b11** 64K
TTL	Translation Table Level - 2-bits.
Scale	2 bits, constant used in the address range exponent calculation. All binary values are valid.
Num	5 bits, constant used as multiplication factor in the range calculation. All binary values are valid.

The total number of bits in the Address range calculation is 48.

The range parameters are placed in the request and snoop packets as shown in Table 8-28 on page 8-320. Unused bits of FwdNID in SnpDVMOp must be set to zero.

Base address of the range operation is placed in Addr[MaxVA:12], with significant address bits adjusted TG value in the following manner.

TG = 4K VA[MaxVA:12].

TG = 16K VA[MaxVA:14]; VA[13:12] must be set to zero.

TG = 64K VA[MaxVA:16]; VA[15:12] must be set to zero.

Table 8-28 Range Based TLBI payload

Addr[x]	DVM request		SnpDVMOp request	
	Request	Data	Part 1	Part 2
0	-	Num[0]	-	-
1	-	Num[1]	-	-
2	-	Num[2]	-	-
3	0	Num[3]	0	1
4	VA valid	Scale[0]	VA valid	Scale[0]
5	VMID valid	Scale[1]	VMID valid	Scale[1]
6	ASID valid	TTL[0]	ASID valid	TTL[0]
7	Security	TTL[1]	Security	TTL[1]
8	Security	TG[0]	Security	TG[0]
9	Exception level	TG[1]	Exception level	TG[1]
10	Exception level	VA[12]	Exception level	VA[12]
11	DVM Op type	VA[13]	DVM Op type	VA[13]
12	DVM Op type	VA[14]	DVM Op type	VA[14]
13	DVM Op type	VA[15]	DVM Op type	VA[15]
21:14	VMID[7:0]	VA[23:16]	VMID[7:0]	VA[23:16]
37:22	ASID[15:0]	VA[39:24]	ASID[15:0]	VA[39:24]
38	Stage	VA[40]	Stage	VA[40]
39	Stage	VA[41]	Stage	VA[41]
40	Leaf	VA[42]	Leaf	VA[42]
41	Range	VA[43]	VA[46]	VA[43]
42	Num[4]	VA[44]	VA[47]	VA[44]
43	-	VA[45]	VA[48]	VA[45]
44	-	VA[46]	VA[50]	VA[49]
45	-	VA[47]	VA[52]	VA[51]
50:46	-	VA[52:48]	-	-
63:56	-	VMID[15:8]	-	-

Table 8-29 shows the placement of the Num value in the SnpDVMOp payload.

Table 8-29 Num value placement in SnpDVMOp payload

FwdNID[x]	SnpDVMOp request	
	Part 1	Part 2
0	Range	Num[0]
1	0	Num[1]
2	0	Num[2]
3	0	Num[3]
4	0	Num[4]
5	0	0
6	0	0

8.3.6 Level Hint in TBLI Operations

Non-range based TLBI operations by VA or IPA are permitted to use TG and TTL. These operations use TG and TTL as an indication of the level of the page table walk that holds the leaf entry for the address that is being invalidated. For such operations:

- Range field must be set to zero.
- Num and Scale fields are inapplicable and must be set to zero.
- When TG[1:0] = 00, TTL information is not provided.

TG and TTL fields are inapplicable and must be set to zero in non-range based TLBI operations which are not by VA or IPA.

Range, Num, Scale, TG and TTL fields are inapplicable and must be set to zero in all non-TLBI DVM operations.

8.3.7 DVM domain

The SnpAttr bit in a DVM request is used to differentiate between Inner and Outer domain. Table 8-30 shows the SnpAttr value encoding in DVM transactions.

Table 8-30 SnpAttr value encoding in DVM transactions

SnpAttr	Domain value
0	Inner domain
1	Outer domain

Two additional optional interface broadcast pins **BROADCASTTLBIINNER** (BTI) and **BROARDCASTTLBIOUTER** (BTO) are defined that determine broadcasting of TLBI operations in the interconnect. See *Optional interface broadcast signals* on page 16-460.

8.3.8 Synchronization

This section shows the DVMSync Synchronization operation.

Table 8-31 shows the fixed value fields in the Sync operation.

Table 8-31 Sync operation fixed values

Addr		Value	Status
Bits	**Field**		
[3]	Part Num	-	See Table 8-9 on page 8-309
[4]	VA Valid	`0b0`	Not applicable
[5]	VMID Valid	`0b0`	Ignore VMID
[6]	ASID Valid	`0b0`	Ignore ASID
[8:7]	Secure	`0b00`	Applies to both Secure and Non-secure
[10:9]	Exception Level	`0b00`	Applies to all Guest OS and Hypervisor
[13:11]	DVMOp type	`0b100`	Synchronization message
[21:14]	VMID VMIDExt	`0xXX`	VMID not specified
[37:22]	ASID	`0xXXXX`	ASID not specified
[39:38]	S2, S1 Staged Invalidation	`0b00`	Set to zero
[40]	Leaf Entry Invalidation	`0b0`	Set to zero

Chapter 9
Error Handling

This chapter describes the error handling requirements. It contains the following sections:

- *Error types* on page 9-324.
- *Error response fields* on page 9-325.
- *Errors and transaction structure* on page 9-326.
- *Error response use by transaction type* on page 9-327.
- *Poison* on page 9-337.
- *Data Check* on page 9-338.
- *Use of interface parity* on page 9-339.
- *Interoperability of Poison and DataCheck* on page 9-342.
- *Hardware and software error categories* on page 9-343.

9.1 Error types

This specification supports two types of error reporting at sub packet level, and two types of error reporting at packet level.

The packet level error reporting types are:

Data Error, DERR

Used when the correct address location has been accessed, but an error is detected within the data. Typically, this is used when data corruption has been detected by ECC or a parity check.

Data Error reporting is supported by the RespErr, Poison, and DataCheck fields in the DAT packet.

When processing of a request received by Home that is required to be propagated to the Slave results in a DERR, the Home must not stop propagating the request to the Slave.

Note

An error in data being evicted from Home, or received in a Snoop response as a result of the request, are examples of the request resulting in a DERR.

Non-data Error, NDERR

Used when an error is detected that is not related to data corruption. This specification does not define all cases when this error type is reported. Typically, this error type is reported for:

- An attempt to access a location that does not exist.
- An illegal access, such as a write to a read only location.
- An attempt to use a transaction type that is not supported.

Non-data Error reporting is supported by the RespErr field in the RSP and DAT packets.

When processing of a request received by Home results in an NDERR then it is permitted, but not required, to propagate the request to the Slave. The Home is required to pass-back the NDERR in the response to the Requester.

9.2 Error response fields

The RespErr field is used to indicate error conditions. The RespErr field is included in both response and data packets.

Table 9-1 shows the encoding of the RespErr field. See *Responses to exclusive requests* on page 6-278 for more details on the Exclusive Okay response.

Table 9-1 Error response field encodings

RespErr[1:0]	Name	Description
`0b00`	OK	Okay. Indicates that a Non-exclusive access has been successful. Also used to indicate an Exclusive access failure.
`0b01`	EXOK	Exclusive Okay. Indicates that either the read or write portion of an Exclusive access has been successful.
`0b10`	DERR	Data Error.
`0b11`	NDERR	Non-data Error.

A single transaction is not permitted to mix OK and EXOK responses.

A transaction with a Data response is required to include NDERR either in none or in all Data packets.

The mixing of OK and DERR responses within a single transaction is permitted.

The mixing of EXOK and DERR responses within a single transaction is permitted.

The mixing of OK and NDERR responses within a single transaction is permitted, which can occur only in transactions with both Data and Non-data responses.

The mixing of EXOK and NDERR is not permitted.

9.3 Errors and transaction structure

All transactions must complete in a protocol compliant manner, even if they include an error response.

Error handling for a transaction that utilizes DMT is the same as the error handling for the same request without DMT.

Because there is no mechanism to propagate errors on requests or snoops, a request must not use DMT or DCT if an error is detected at the interconnect.

If the transaction contains data packets then the source of the data packets is required to send the correct number of packets, but the data values are not required to be valid.

The Resp field gives the cache states associated with a transaction and can be influenced by an error condition. See *Response types* on page 4-190 for more details on the legal Resp field values. If a response to a transaction does not have a legal cache state, then the RespErr field must indicate a Non-data Error for all data packets.

The Resp field in a response must have the same value for every packet of a Data message regardless of whether or not there is an error condition.

A Requester that receives a response with a Non-data Error for a Snoopable request must:

For an allocating transaction:

- When the start state is Invalid (I), the Requester must not allocate the received data.
- If the request was sent from a non-Invalid state, the Requester must leave the cached copy unchanged.

In both cases, the cache state must not be changed.

The allocating transactions are:

- ReadClean.
- ReadNotSharedDirty.
- ReadShared.
- ReadUnique.
- ReadPreferUnique.
- MakeReadUnique.
- CleanUnique.
- MakeUnique.

For a deallocating transaction:

- The Requester must continue as normal, in a protocol compliant manner.

The deallocating transactions are:

- WriteBack.
- WriteEvictFull.
- Evict.
- WriteEvictOrEvict.

For Other transactions, that do not change allocation:

- The Requester must not upgrade the cache state but is permitted to downgrade it.

The cache state in a SnpResp message with a Non-data Error must be Invalid (I). The responder must invalidate local cached copies of the cache line. In addition, when the response to a Forwarding snoop results in a Non-data Error, the Snoopee must not forward data to the Requester. As a consequence, if the CompData message has already been sent to the Requester then the Snoop response to the Home must not include the Non-data Error.

9.4 Error response use by transaction type

This section defines the permitted use of the error fields for each transaction type.

The tables that follow show the Data and Response packets associated with the following transaction types:

- *Read Transactions*.
- *Dataless transactions* on page 9-329.
- *Write transactions* on page 9-330.
- *Atomic transactions* on page 9-332.
- *Other transactions* on page 9-334.
- *Cache Stashing transactions* on page 9-334.
- *Snoop transactions* on page 9-335.

The following keys are used by the tables:

OK	The RespErr field must contain the OK RespErr value of `0b00`.
Y	This value of RespErr is permitted.
N	This value of RespErr is not permitted.
-	Data or Response packet is not used for this transaction type.

9.4.1 Read Transactions

Read transactions can contain multiple CompData data packets.

When RespSepData includes a Non-data Error, all corresponding DataSepResp packets must be marked with Non-data Error.

In a Data response to a Read request, a Non-data Error response is only permitted either in none or in all data response packets.

Note

Prior to CHI Issue E, a Read transaction was permitted to include a Non-data Error response in some of the data responses.

Table 9-2 Read transaction's Data and Response packets legal RespErr field values

Read transaction	Associated Data and Response packets					
	Read Receipt	CompData				CompAck
		OK	EXOK	DERR	NDERR	
ReadNoSnp	OK	Y	Y	Y	Y	OK
ReadNoSnpSep	OK	-	-	-	-	-

Table 9-2 Read transaction's Data and Response packets legal RespErr field values (continued)

Read transaction	Associated Data and Response packets					
	Read Receipt	CompData				CompAck
		OK	EXOK	DERR	NDERR	
ReadOnce ReadOnceCleanInvalid ReadOnceMakeInvalid	OK	Y	N	Y	Y	OK
ReadClean ReadNotSharedDirty ReadShared	-	Y	Y	Y	Y	OK
ReadUnique ReadPreferUnique[a] MakeReadUnique[b]	-	Y	N	Y	Y	OK

[a] EXOK response is not permitted even when Excl bit in the request is set to one.

A response of OK in the returned data response is not be taken as a failure of ReadPreferUnique. Failure of the exclusive sequence is only determined from the corresponding MakeReadUnique (Excl) or CleanUnique (Excl) transaction completion.

[b] Applicable only when data is returned to the Requester. See Table 9-4 on page 9-329 for permitted errors when data is not returned to the Requester.

Table 9-3 Read transaction's Data-only and Non-data packets legal RespErr field values

Read transaction	Associated Data-only and Non-data packets							
	DataSepResp				RespSepData			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
ReadNoSnp	Y	N	Y	Y	Y	N	N	Y
ReadNoSnpSep	Y	N	Y	Y	-	-	-	-
ReadOnce ReadOnceCleanInvalid ReadOnceMakeInvalid	Y	N	Y	Y	Y	N	N	Y
ReadClean ReadNotSharedDirty ReadShared	Y	N	Y	Y	Y	N	N	Y
ReadUnique ReadPreferUnique[a] MakeReadUnique[b]	Y	N	Y	Y	Y	N	N	Y

[a] A response of OK in the returned data response is not be taken as a failure of ReadPreferUnique. Failure of the exclusive sequence is only determined from the corresponding MakeReadUnique (Excl) or CleanUnique (Excl) transaction completion.

[b] Applicable only when data is returned to the Requester. See Table 9-4 on page 9-329 for permitted errors when data is not returned to the Requester.

9.4.2 Dataless transactions

A Data Error can be reported for a Dataless transaction when the processing of the transaction by another component encounters a data corruption error. This data error can be indicated back to the originating component, even though a transfer of data does not occur.

Table 9-4 shows the Dataless transaction packets legal RespErr field values.

Table 9-4 Dataless transaction packets legal RespErr field values

Dataless transaction	Associated Response packets												
	Comp				Persist				CompPersist				CompAck
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR	
CleanUnique	Y	Y	Y	Y	-	-	-	-	-	-	-	-	OK
MakeReadUnique[a]	Y	N	Y	Y	-	-	-	-	-	-	-	-	OK
MakeUnique	Y	N	Y	Y	-	-	-	-	-	-	-	-	OK
CleanShared	Y	N	Y	Y	-	-	-	-	-	-	-	-	-
CleanSharedPersist	Y	N	Y	Y	-	-	-	-	-	-	-	-	-
CleanSharedPersistSep	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	-
CleanInvalid MakeInvalid	Y	N	Y	Y	-	-	-	-	-	-	-	-	-
Evict	Y	N	N	Y	-	-	-	-	-	-	-	-	-

a. Applicable only when data is not returned to the Requester. See Table 9-2 on page 9-327 and Table 9-3 on page 9-328 for permitted errors when data is returned to the Requester.

Table 9-5 shows the permitted RespErr values in responses to StashOnce transactions.

Table 9-5 Dataless Stash transaction packets legal RespErr field values

Dataless Stash transaction	Associated Response packets											
	Comp				StashDone				CompStashDone			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
StashOnceUnique StashOnceShared	Y	N	Y	Y	-	-	-	-	-	-	-	-
StashOnceSepUnique StashOnceSepShared	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y

9.4.3 Write transactions

A Write transaction can include either a Non-data Error or a Data Error. Errors can be signalled in both directions, from the Requester to the Completer, and from the Completer back to the Requester.

For a Write transaction an error can be signalled from the Completer back to the Requester using either the combined CompDBIDResp or using the Comp response. It is permitted for the Completer to signal an error even before it has observed the WriteData for the transaction and this can occur when the processing of the transaction, such as the cache lookup, encounters a data corruption error.

Table 9-6 shows the Write transaction Response packets legal RespErr field values.

Table 9-6 Write transaction Response packets legal RespErr field values

Write transaction	Associated Response packets									
	DBIDResp	Comp				CompDBIDResp				CompAck
		OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR	
WriteNoSnp	OK	Y	Y	Y	Y	Y	Y	Y	Y	OK
WriteUnique	OK	Y	N	Y	Y	Y	N	Y	Y	OK
WriteNoSnpZero WriteUniqueZero	OK	Y	N	Y	Y	-	-	-	-	-
WriteBack WriteClean WriteEvictFull	-	-	-	-	-	Y	N	Y	Y	-
WriteEvictOrEvict	-	Y	N	N	Y	Y	N	Y	Y	OK

A Requester that detects an error in the write data to be sent can include an error indication with the write data packet. This indicates that the data value is known to be corrupt.

Table 9-7 shows the Write transaction Data packets legal RespErr field values.

Table 9-7 Write transaction Data packets legal RespErr field values

Write transaction	Associated Data packets											
	WriteData				WriteDataCancel				NCBWrDataCompAck			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
WriteNoSnp	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
WriteUnique	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
WriteBack WriteClean WriteEvictFull WriteEvictOrEvict	Y	N	Y	N	-	-	-	-	-	-	-	-

Table 9-8 shows the Write transaction TagMatch packet legal RespErr field values when MTE is supported on the Requester's interface.

Table 9-8 Write transaction TagMatch packet legal RespErr field values

Write transaction	Associated response packets			
	TagMatch			
	OK	EXOK	DERR	NDERR
WriteNoSnp WriteUnique	Y	N	Y	Y
WriteBack WriteClean WriteEvictFull WriteEvictOrEvict	-	-	-	-

For permitted RespErr field values in a Combined Write transaction response:

- See Table 9-6 on page 9-330 for the corresponding responses for the Write request.
- See Table 9-4 on page 9-329 for the corresponding responses for the CMO request.

  The permitted error responses in the CompCMO are the same as those in the Comp in Table 9-4 on page 9-329.

9.4.4 Atomic transactions

It is permitted for a Completer to give a Comp response before it has received all the write data associated with a transaction and has performed the required operation. This behavior is not compatible with a component that wants to signal a data error associated with the write data, and such components must use a delayed form of Comp or CompData response.

A Data Error or Non-data Error can be signaled at the following points within a transaction:

- With the DBIDResp response.
- With the CompDBIDResp response.
- For an AtomicStore transaction, with the Comp response.
- For an AtomicLoad, AtomicSwap, and AtomicCompare transaction, with the CompData response.

Note

If a read data at Home due to a non-store Atomic request results in a Data Error or Non-data Error, then such an error can be propagated onto the DBIDResp or CompDBIDResp response for that request.

For Atomic transactions that are not able to complete, a Non-data Error must be used. The transaction structure, including all write data transfers, read data transfers, and other responses must still take place.

There is no need to specify an error associated with the execution of an atomic operation, such as overflow. All atomic operations are fully specified for all input combinations.

A transaction includes both outbound and inbound data, but only has a single Error field. For Atomic transactions it is permitted for the Error field to indicate an error on either write data or read data. There is no mechanism supported within the transaction to differentiate between the potential different causes of an error. A fault log, or a similar structure, might be able to provide such information, but this is not a requirement of this specification.

The permitted RespErr values in Atomic transactions are an amalgamation of those permitted in Read and Write transactions.

A Data Error can vary between data packets.

Table 9-9 shows the Atomic transaction Response packets legal RespErr field values

Table 9-9 Atomic transaction Response packets legal RespErr field values

Atomic transaction	Associated Response packets								
	DBIDResp	Comp				CompDBIDResp			
		OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
AtomicStore	OK	Y	N	Y	Y	Y	N	Y	Y
AtomicLoad AtomicSwap AtomicCompare	OK	Y	N	Y	Y	-	-	-	-

Table 9-10 shows the Atomic transaction Data packets legal RespErr field values.

Table 9-10 Atomic transaction Data packets legal RespErr field values

Atomic transaction	Associated response packets							
	WriteData				CompData			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
AtomicStore	Y	N	Y	N	-	-	-	-
AtomicLoad AtomicSwap AtomicCompare	Y	N	Y	N	Y	N	Y	Y

Table 9-11 shows the Atomic transaction TagMatch packet legal RespErr field values when MTE is supported.

Table 9-11 Atomic transaction TagMatch packet legal RespErr field values

Atomic transaction	Associated response packets			
	TagMatch			
	OK	EXOK	DERR	NDERR
AtomicStore AtomicLoad AtomicSwap AtomicCompare	Y	N	Y	Y

9.4.5 Other transactions

This section describes the error handling requirements for the DVMOp and PrefetchTgt transactions.

DVMOp

A DVMOp transaction can include a Non-data Error in the Comp response. The interconnect can consolidate error responses from all the snoop responses for a DVMOp and include a single error response in the final Comp message to the Requester. The DBIDResp packet must only use the OK response. Even though the Sender of a WriteData response might not use DERR, the packet can be marked as DERR if it encounters errors during transmission. See *Interoperability of Poison and DataCheck* on page 9-342.

Table 9-12 shows the DVM transaction Response packets legal RespErr field values.

Table 9-12 DVM transaction Response packets legal RespErr field values

DVM transaction	Associated Response packets								
	DBIDResp	Comp				CompDBIDResp			
		OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
DVMOp	OK	Y	N	Y	Y	Y	N	Y	Y

Table 9-13 shows the DVM transaction Data packets legal RespErr field values.

Table 9-13 DVM transaction Data packets legal RespErr field values

DVM transaction	Associated Data packets			
	NCBWrData			
	OK	EXOK	DERR	NDERR
DVMOp	Y	N	Y	N

PrefetchTgt

A PrefetchTgt transaction request to a non-supporting address must be discarded.

Note

A component is permitted to record and report such an error.

9.4.6 Cache Stashing transactions

If the specified stash target does not support receiving Stash type snoops then Home must disregard the Stash hint and complete the transaction without Stashing. Examples of such Stash targets are RN-I, RN-D, legacy RN-F or a Non-request node. In these circumstances, Home must not signal an error to the Requester. Such a wrongly specified Stash target can be attributed to a software based error.

If the Home does not support Stash requests, it must complete the transaction in a protocol-compliant manner without signaling an error.

9.4.7 Snoop transactions

A snoop transaction response that includes data can indicate a Data Error. A Snoop transaction response that includes data can mix Okay and Data Error responses for different packets within the transaction.

A snoop transaction response that does not include data can indicate a Non-data Error.

Table 9-14 shows the Snoop request Response packets legal RespErr field values.

Table 9-14 Snoop request Response packets legal RespErr field values

Snoop Transaction	**Associated Data and Response packets**							
	SnpResp				**SnpRespData**			
	OK	**EXOK**	**DERR**	**NDERR**	**OK**	**EXOK**	**DERR**	**NDERR**
SnpOnce SnpClean SnpNotSharedDirty SnpShared SnpUnique SnpPreferUnique SnpUniqueStash SnpCleanShared SnpCleanInvalid	Y	N	N	Y	Y	N	Y	N
SnpStashUnique SnpStashShared SnpMakeInvalid SnpMakeInvalidStash SnpQuery SnpDVMOp	Y	N	N	Y	-	-	-	-

It is recommended, but not required, that a DERR on a Clean cache line is dropped, and the error is not propagated to the memory, nor included in the response to the Requester.

A DERR on a Dirty cache line must be propagated to the memory, and in the response to the Requester.

A DERR in response to the Data Pull request is not expected to be transferred to the Comp response to the Stash request.

For a Forwarding snoop transaction, when simultaneously forwarding data to the Requester and returning Data to Home, it is permitted for only one response to include an indication of a Data Error if the other response does not encounter the error.

Table 9-15 shows the Forwarding Snoop response packets legal RespErr field values.

Table 9-15 Forwarding Snoop response packets legal RespErr field values

Snoop transaction	Associated Response packets							
	SnpResp				SnpRespFwded			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
SnpOnceFwd SnpCleanFwd SnpNotSharedDirtyFwd SnpSharedFwd SnpUniqueFwd SnpPreferUniqueFwd	Y	N	N	Y	Y	N	Y	N

Table 9-16 shows the Forwarding Snoop Data response packets legal RespErr field values.

Table 9-16 Forwarding Snoop Data response packets legal RespErr field values

Snoop transaction	Associated Data packets							
	SnpRespData SnpRespDataFwded				CompData			
	OK	EXOK	DERR	NDERR	OK	EXOK	DERR	NDERR
SnpOnceFwd SnpCleanFwd SnpNotSharedDirtyFwd SnpSharedFwd SnpUniqueFwd SnpPreferUniqueFwd	Y	N	Y	N	Y	N	Y	N

9.5 Poison

The Poison bit is used to indicate that a set of data bytes have previously been corrupted. Passing the Poison bit alongside the data in the DAT packet permits any future user of the data to be notified that the data is corrupt.

When Poison is supported:

- The DAT packet includes one Poison bit per 64 bits of data.
- Data marked as poisoned:
  - Must not be utilized by any Requester.
  - Is permitted to be stored in caches and memory if marked as poisoned.
- The Poison value, once set, must be propagated along with the data.
- When a Poison error is detected, it is permitted to over poison the data.
- Poison on MTE tags is not supported.

Poison must be accurate if there are any valid bytes in the 64-bit chunk, which is Poison granularity. Otherwise, the Poison bit is a Don’t Care, that is, when all 8 bytes in the 64-bit chunk are invalid, then it is a Don’t Care.

A Data_Poison property is used to indicate if a component supports Poison.

9.6 Data Check

The DataCheck field is used to detect data errors in the DAT packet.

When Data Check is supported:

- The DAT packet carries eight Data Check bits per 64 bits of data.
- The Data Check bit is a parity bit that generates Odd Byte parity.

The Data_Check property is used to indicate if Data Check is supported.

Note

Interface parity optionally extends the error detection provided on the DAT channel by the DataCheck field.

The protection scheme employed on an interface is defined by the property Check_Type. See *Interface properties and parameters* on page 16-456.

9.7 Use of interface parity

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 connections are relatively short. This section describes a parity scheme for detecting single-bit errors on the interface between components. Multi-bit errors may be detected if they occur in different parity signal groups. Figure 9-1 shows locations where parity can be used.

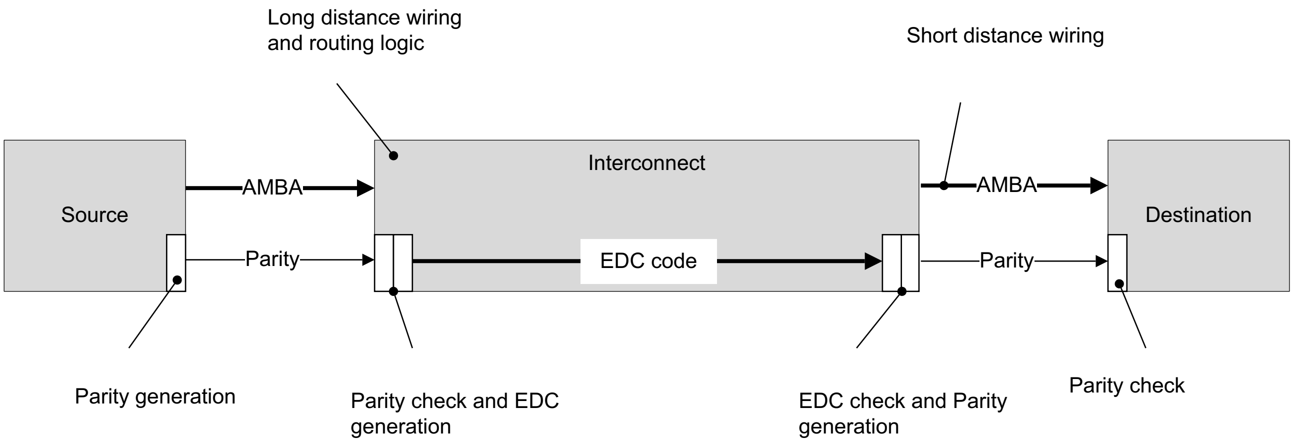


Figure 9-1 Parity use in AMBA

AMBA parity optionally extends the error detection provided on the DAT channel by the DataCheck field to cover the complete flit and control signals on all channels.

The protection scheme employed on an interface is defined by the property Check_Type. See *Interface properties and parameters* on page 16-456.

9.7.1 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 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.
- The least significant check bit of the check signal covers the least significant byte of payload.
- If the bits in a payload does not fill the most significant byte, the most significant bit of the check signal covers fewer than 8 bits.

- Check signals must be driven correctly in every cycle that the Check Enable term is True, see Table 9-17 on page 9-341.
- Parity signals must be driven appropriate to all the bits in the associated payload, irrespective of whether those bits are applicable.

9.7.2 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. Also, an error on one signal can also cause parity failure on other signals.

When an error is detected, the receiver has options to:

- Terminate or propagate the transaction. It is permitted, but not required, to be protocol compliant when the transaction is terminated.
- Correct the parity check signal or propagate the error.
- Update its memory or leave untouched. It is permitted, but not required, to mark the location as poisoned.
- Signal an error response through other means, for example, with an interrupt.

9.7.3 Interface parity check signals

Table 9-17 shows the parity check signals and their properties on each of the channels.

Table 9-17 Interface parity check signals

Channel	Check signal	Signals covered	Check signal width	Check granularity	Check enable
Common	**SYSCOREQCHK**	**SYSCOREQ**	1	1	**RESETn** == 1
	SYSCOACKCHK	**SYSCOACK**	1	1	**RESETn** == 1
Common	**TXSACTIVECHK**	**TXSACTIVE**	1	1	**RESETn** == 1
	RXSACTIVECHK	**RXSACTIVE**	1	1	**RESETn** == 1
Common	**TXLINKACTIVEREQCHK**	**TXLINKACTIVEREQ**	1	1	**RESETn** == 1
	TXLINKACTIVEACKCHK	**TXLINKACTIVEACK**	1	1	**RESETn** == 1
	RXLINKACTIVEREQCHK	**RXLINKACTIVEREQ**	1	1	**RESETn** == 1
	RXLINKACTIVEACKCHK	**RXLINKACTIVEACK**	1	1	**RESETn** == 1
REQ	**REQFLITPENDCHK**	**REQFLITPEND**	1	1	**RESETn** == 1
	REQFLITVCHK	**REQFLITV**	1	1	**RESETn** == 1
	REQFLITCHK	**REQFLIT**	ceil(R/8)[a]	1 to 8	**REQFLITV** == 1
	REQLCRDVCHK	**REQLCRDV**	1	1	**RESETn** == 1
RSP	**RSPFLITPENDCHK**	**RSPFLITPEND**	1	1	**RESETn** == 1
	RSPFLITVCHK	**RSPFLITV**	1	1	**RESETn** == 1
	RSPFLITCHK	**RSPFLIT**	ceil(T/8)[b]	1 to 8	**RSPFLITV** == 1
	RSPLCRDVCHK	**RSPLCRDV**	1	1	**RESETn** == 1
SNP	**SNPFLITPENDCHK**	**SNPFLITPEND**	1	1	**RESETn** == 1
	SNPFLITVCHK	**SNPFLITV**	1	1	**RESETn** == 1
	SNPFLITCHK	**SNPFLIT**	ceil(S/8)[c]	1 to 8	**SNPFLITV** == 1
	SNPLCRDVCHK	**SNPLCRDV**	1	1	**RESETn** == 1
DAT	**DATFLITPENDCHK**	**DATFLITPEND**	1	1	**RESETn** == 1
	DATFLITVCHK	**DATFLITV**	1	1	**RESETn** == 1
	DATFLITCHK	**DATFLIT**	ceil(D/8)[d]	1 to 8	**DATFLITV** == 1
	DATLCRDVCHK	**DATLCRDV**	1	1	**RESETn** == 1

a. R = Request flit width See *Request flit* on page 13-398.

b. T = Response flit width. See *Response flit* on page 13-400.

c. S = Snoop flit width. See *Snoop flit* on page 13-401.

d. D = Data flit width. See *Data flit* on page 13-402.

9.8 Interoperability of Poison and DataCheck

If the recipient of a Data packet does not support the Poison and DataCheck features then the interconnect must enumerate and convert, as necessary, the Poison and Data Check error responses to a Data Error in the DAT packet.

If support for the Poison and DataCheck features is not similar across an interface, then the following rules apply:

- Poison must be mapped to DataCheck or DERR if Poison is not supported across the interface. At such an interface, Poison is expected but not required to be mapped to DataCheck instead of DERR, if DataCheck is supported.

  When converting from Poison to DataCheck, when an 8-byte chunk is marked as Poisoned, all 8 bits of DataCheck corresponding to that chunk must be manipulated to generate a parity error.
- DataCheck must be mapped to Poison or DERR if DataCheck is not supported across the interface. At such an interface, DataCheck is expected but not required to be mapped to Poison instead of DERR, if Poison is supported.

  When converting from DataCheck to Poison, if one or more DataCheck bits in a given 8-byte chunk generates a parity error, then the Poison bit corresponding to that chunk must be set.

Note

The difference between the handling of Poison and DERR is that a Poison error in a received Data packet is typically deferred by the receiver, but a DERR error is typically not deferred by the receiver.

It is sufficient for the Sender of a Data packet that detects a Poison error to indicate this in the Poison bits. It is not a requirement that the Sender sets the RespErr field value to DERR.

It is sufficient for the Sender of a Data packet that detects a DataCheck error to indicate this in the DataCheck field and is not required to set RespErr field value to DERR.

As Poison and DataCheck fields are independently set, one type of error does not require setting of the other.

In a Data packet that has the RespErr field value set to DERR or NDERR the value of the Poison and DataCheck fields are Don’t Care.

9.9 Hardware and software error categories

This specification defines two error categories, a software based error and a hardware based error.

9.9.1 Software based error

A software based error occurs when multiple accesses to the same location are made with mismatched Snoopable or Memory attributes.

A software based error can cause a loss of coherency and the corruption of data values. This specification requires that the system does not deadlock for a software based error, and that transactions always progress through a system in a timely manner.

A software based error, for an access within one 4KB memory region, must not cause data corruption within a different 4KB memory region.

For locations held in Normal memory, the use of appropriate stores and software cache maintenance can be used to return memory locations to a defined state.

When accessing a peripheral device 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 required to return a peripheral device that has been accessed incorrectly to a known working state is IMPLEMENTATION DEFINED.

9.9.2 Hardware based error

A hardware based error is defined as any protocol error that is not a software based error.

Warning

If a hardware based error occurs then recovery from the error is not guaranteed. The system might crash, lock-up, or suffer some other non-recoverable failure.

Chapter 10

Quality of Service

This chapter describes the mechanisms in the CHI protocol to support *Quality of Service* (QoS). It contains the following sections:

10.1 Overview

A system might utilize a QoS scheme to achieve:

- A guaranteed maximum latency for transactions in a particular stream.
- Minimum bandwidth guarantees for a stream of requests.
- Best effort value of bandwidth and latency provided to requests of a particular stream.

The low latency, or guaranteed throughput requirements, required to meet system QoS demands are primarily the responsibility of the transaction end points with support from the intermediate interconnect. The protocol supports this by defining a QoS priority value for packets and controlling request flow using a defined credit mechanism.

10.2 QoS priority value

A 4-bit value is used to prioritize the processing of the packets at protocol nodes and within the interconnect. The QoS *Priority Value* (PV) for packets is assigned by the source of the transaction. In typical usage models this value is dependent on the source type and the class of traffic, with ascending values of QoS indicating a higher priority level. The source might also dynamically vary this value, depending on some accumulated latency and required throughput metric.

10.3 Repeating a transaction with higher QoS value

When a transaction has been sent with a particular QoS value, it is permitted to send the same transaction again with a different, typically higher, QoS value. The Completer is required to handle this situation as multiple different requests.

In this situation, if one of the transactions receives a RetryAck response, then it is permitted to cancel the transaction and return the credit. See *Credit Return* on page 2-148.

Chapter 11
System Debug, Trace, and Monitoring

This chapter describes mechanisms that provide additional support for the debugging and tracing of systems, and the monitoring of systems to enhance performance. It contains the following sections:

11.1 Data Source indication

This specification permits the Completer of a Read request to specify the source of the data. The source is specified in the DataSource field of the CompData, DataSepResp, SnpRespData and SnpRespDataPtl responses. DataSource field value is valid even in data responses with error.

The DataSource field can also be used to transport information to bias SLC replacement policy. See *SLC replacement hint* on page 11-353.

11.1.1 DataSource value assignment

The DataSource values must be assigned as follows:

- Fixed values are used for DataSource when Data comes from memory and are used to indicate the following:
  - **`0b0110`** PrefetchTgt memory prefetch was useful.

    Read data was obtained from Slave with lower latency as the PrefetchTgt request already read or initiated a read of data from memory.
  - **`0b0111`** PrefetchTgt memory prefetch was not useful.

    Read request went through a complete memory access and therefore did not have any latency reduction due to the PrefetchTgt request sent earlier.

    The precise reason for signaling that a prefetch was not useful is IMPLEMENTATION DEFINED.

  Note

  There are several reasons why the PrefetchTgt request might not be useful. Examples are that the prefetch was dropped by the Slave, the data obtained by the prefetch was replaced in the buffer, or the Read request arrived at the Slave before the prefetch.

- For a response not from memory, that is, from a cache, the DataSource value is IMPLEMENTATION DEFINED.

  This specification recommends, but does not require, settings for DataSource in these cases.

  A component is permitted to have software programmability to override the DataSource value to:
  - Change the groupings to more suitable specific configuration settings.
  - Change the values where the values are not correct.

- A responder is permitted to not support sending a useful DataSource value:
  - The responder, except for a memory SN-F, must return a `0b0000` value.
  - A memory SN-F component must return `0b0111` as a default value.

  Such exceptions must be understood by the system.

11.1.2 Crossing a chip-to-chip interface

It is the responsibility of the chip interface module, if one exists, to map DataSource values in the incoming Data packets to different values to identify that the response came from the remote chip caches.

Example approaches that the chip interface module might take are:

- Group the remote caches into a single encoding, as Figure 11-1 on page 11-351 shows.
- Have a maximum size of an eight entry table, to remap the implementation values of the DataSource field in the incoming Data packet to new values.

Suggested DataSource values

Figure 11-1 shows an example multichip configuration and the suggested mapping of DataSource values to different components in the system:

- Each chip in the system has two processors per cluster, with a three level cache hierarchy.
- The cache in the chip-to-chip interface module is identified as part of the interconnect caches.
- All the caches in the remote peer chip are grouped together.
- A non-memory component that is not programmed to identify itself as the source of data can return the default value of `0b0000`.

Table 11-1 lists the suggested DataSource encodings.

Table 11-1 Suggested DataSource value encodings

DataSource	Suggested mapping
`0b0000`	Non-memory default. Source does not support sending a useful DataSource value
`0b0001`	Peer processor cache within local cluster
`0b0010`	Local cluster cache
`0b0011`	Interconnect cache
`0b0100`	Peer cluster caches
`0b0101`	Remote chip caches
`0b1010`	Local cluster cache, unused prefetch[a]
`0b1011`	Interconnect cache, unused prefetch[a]

[a] New in CHI Issue E. See *SLC replacement hint* on page 11-353.

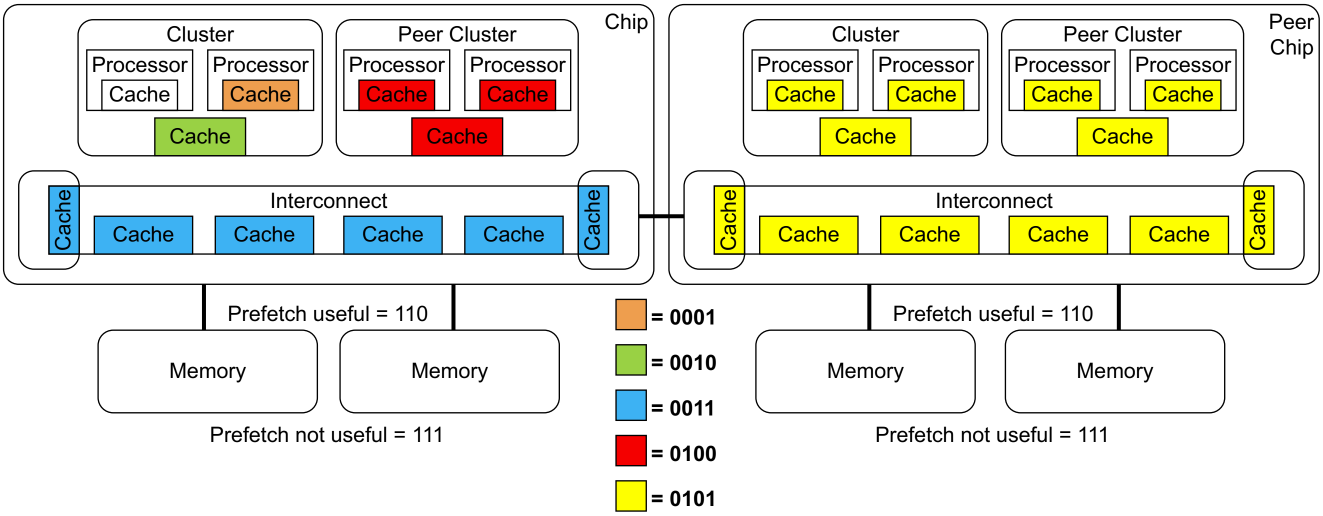


Figure 11-1 Suggested DataSource values

11.1.3 Example use cases

Two examples of how DataSource information can be used by a Requester are:

- To determine the usefulness of a PrefetchTgt transaction in initiating a memory controller prefetch.
    - By monitoring the DataSource value in the data returned from the memory SNF, the Requester can determine the usefulness of sending PrefetchTgt requests and can modulate the rate, as well as the sending, of PrefetchTgt requests.
- Can be used by performance profiling and debug software to evaluate and optimize the data sharing pattern.

11.2 SLC replacement hint

The purpose of this feature is to forward cache replacement hints from the Requesters to the caches in the interconnect, the *System Level Caches* (SLC). Typically, an RN has the best knowledge of the utility of a cache line. A system level cache that is informed of this knowledge can use it to bias its own replacement algorithms and manage cache line replacement in a more efficient manner.

11.2.1 Characteristics

Although the replacement information is most useful in CopyBack requests, this feature is not restricted to CopyBack transactions alone, it is extended to all requests from RN to HN-F except for the following:

- Atomics.
- Stash transactions when StashNIDValid is 1.
- PrefetchTgt.
- PCrdReturn.
- DVMOp.

This feature is supported by a 7-bit field called SLCRepHint that is included in the REQ channel. SLCRepHint includes two subfields, a 3-bit Replacement field and a 1-bit UnusedPrefetch field.

Figure 11-2 shows the placement of the SLCRepHint subfields.

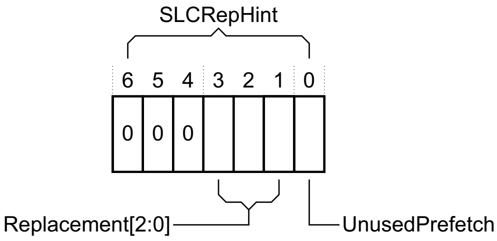


Figure 11-2 SLCRepHint subfield placement

The SLCRepHint field shares the same REQ packet location as ReturnNID and StashNID. When the Node ID widths are greater than 7-bit and the field is used as SLCRepHint then the unused bits of the shared field must be set to zero.

Table 11-2 shows the encoding of the UnusedPrefetch subfield.

Table 11-2 UnusedPrefetch subfield encoding

UnusedPrefetch	Description
0	The cache line might have been used since being fetched
1	The cache line was not used since being fetched

Note

An RN that does not track the usage of the cache line can set the UnusedPrefetch bit value to zero.

The SLCRepHint field is only applicable in requests from an RN to HN-F.

The field is inapplicable in requests from RN to HN-I and HN to SN and can take any value.

Table 11-3 shows the suggested Replacement subfield encodings and their meaning.

Table 11-3 Suggested Replacement subfield encodings

Replacement[2:0]	Description
`0b000`	No recommendation (default)
`0b100`	Most likely to be used again
`0b101`	More likely to be used again
`0b110`	Somewhat likely to be used again
`0b111`	Least likely to be used again
`0b011` - `0b001`	Unused

11.3 MPAM

Memory System Performance Resource Partitioning and Monitoring (MPAM) is a mechanism to efficiently utilize the memory resources among users and to monitor the utilization of those resources. The resources are partitioned among users by *Partition ID* (PartID) and *Performance Monitoring Group* (PerfMonGroup). A Requester that supports MPAM includes in each request it sends a label, identifying the partition to which it belongs, together with the performance monitoring group within that partition. The Home or the Slave use this information to allocate their resources to this request.

The MPAM field is applicable only in the REQ and SNP channels:

- On the REQ channel, when a sender does not want to use MPAM for a request, the MPAM values must be set to default settings. See *Default values for MPAM sub-fields* on page 11-356.
- On the SNP channel, MPAM values are only applicable in Stash type snoops.

  In Non-stash type snoops, MPAM values are inapplicable and must be set to default values.

Field width is either 0 or 11 bits:

- The width is zero on interfaces that do not support MPAM.
- The width is 11 bits on interfaces that support MPAM. The field is further divided into the sub-fields:
  - PartID = 9 bits.
  - PerfMonGroup = 1 bit.
  - MPAMNS = 1 bit.

Figure 11-3 shows the allocation of the MPAM field bits.

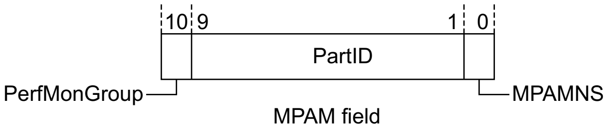


Figure 11-3 MPAM sub-fields bit allocation

How the MPAM field values are used by the receiver is IMPLEMENTATION DEFINED.

11.3.1 MPAMNS

A Non-secure bit in the MPAM field, this is in addition to and different from the NS bit of the request.

The polarity of the MPAMNS bit encoding is the same as that of the NS bit.

Table 11-4 shows the MPAMNS encodings.

Table 11-4 MPAMNS encodings

MPAMNS	Description
0	Secure partition
1	Non-secure partition

Table 11-5 shows the permitted combinations of MPAMNS and NS.

Table 11-5 Permitted value combinations of MPAMNS and NS

MPAMNS	NS	Permitted	Comments
0	0	Yes	Secure partition context Secure location
0	1	Yes	Secure partition context Non-secure location
1	0	No	Non-secure partition context Secure location
1	1	Yes	Non-secure partition context Non-secure location

11.3.2 MPAM value propagation

The receiver is permitted, but not required, to support the full range of partitions and performance monitor groups received in the request. It is expected that during the system discovery and configuration process the capabilities of the system are discovered and the range of partitions and performance monitor groups that are used will match the capabilities of the system.

MPAM field values must be propagated onto an interface that supports MPAM.

It is permitted, but not required, to propagate MPAM field values onto an interface that does not support MPAM.

Table 11-6 shows the default values for MPAM fields when not supported or not propagated.

Table 11-6 Default values for MPAM sub-fields

MPAM field	Value
PerfMonGroup	0
PartID	0
MPAMNS	• Same as NS value in the Request message • Same as NS value in the Snoop message

11.3.3 Stash transaction rules

MPAM values in Stash type snoops must be the same as in the request that generated the snoops.

For responses to Stash type snoops, when the response includes a Data Pull request, the Home must assume the MPAM values in the Data Pull request are the same as in the original Stash request.

11.3.4 Request to Slave rules

MPAM values in a request to Slave, where the request is generated by a request to the Home, must be the same as the MPAM values in that request to Home.

11.4 Completer Busy

The Completer Busy indication is a mechanism for the Completer of a transaction to indicate its current level of activity. This signaling provides additional information to a Requester on how aggressive it can be in generating speculative activity to improve performance.

CBusy Completer Busy. A 3-bit field that is applicable in the appropriate DAT and RSP packets.

When separate Data and Comp responses are used for a single Read request, the Busy indication in each response can be independently set.

The CBusy field is not applicable in:
- NonCopyBackWrData.
- NonCopyBackWrDataAck.
- CopyBackWriteData.
- WriteDataCancel.
- CompAck.

Note

DataSource can be used as a qualifier to the Completer Busy indication such that some data sources, for example a Forwarding snoop, do not influence the busy indication.

11.4.1 Use case

This specification does not define how a CBusy indication is set by the Completer nor how it should be interpreted by a Requester. This is IMPLEMENTATION DEFINED. However, this specification recommends that implementations consider mechanisms to avoid a few very busy Completers among many skewing the results.

Example use case

The following is an example encoding of the CBusy field.

CBusy[2] When asserted indicates multiple cores are actively making requests.

CBusy[1:0] Indicates the degree of fullness of the tracker at the Completer as:
- 00 = Less than 50% full.
- 01 = Greater than 50% full.
- 10 - Greater than 75% full.
- 11 = Greater than 90% full.

The prefetcher at the Requester can use the CBusy field values and fine tune the prefetcher in the following manner:
- If CBusy[2] = 1 and CBusy[1:0] = 11: Disable inaccurate prefetchers.
- If CBusy[1:0] = 10: Very conservative mode on inaccurate prefetchers.
- If CBusy[2] = 1 and CBusy[1:0] = 01: Moderately aggressive mode on inaccurate prefetchers.
- If CBusy[1:0] = 00: Fully aggressive mode on inaccurate prefetchers.

11.5 Trace Tag

This specification includes a TraceTag bit per channel that provides enhanced support for debugging, tracing, and performance measurement of systems.

11.5.1 TraceTag usage and rules

The rules for when to set and how to propagate TraceTag bit values are:

- The TraceTag bit can be set by the transaction initiator or an interconnect component.
- A component that receives a packet with a TraceTag bit set in every received packet must preserve and reflect the value back in any response packet or spawned packet generated in response to the received packet.
- If a received packet spawns multiple responses, such as a Write request resulting in separate Comp and DBIDResp responses, or a Read request generating separate DataSepResp and RespSepData responses, all such spawned responses are required to have the TraceTag bit set if the spawning packet has the TraceTag bit set. If the spawning packet does not have the TraceTag bit set, then the value of the TraceTag bit in a spawned packet is independent of the value of the bit in other related spawned packets.
- If a component can receive multiple packets that are associated with a single transaction, then for each packet that it, in turn, generates, the TraceTag value is only required to be set if it is set in the associated received packet. For example:
  - — A Write transaction flow at RN might have write data and CompAck as two responses for received packets DBIDResp and Comp respectively. As CompAck is in response to the received Comp only, its TraceTag bit value is only required to be dependent on the TraceTag bit value in the Comp packet and similarly for the write data and DBIDResp Response-Received Packet pair.

    A Request that receives separate DataSepResp and RespSepData responses and generates a CompAck, is only required to have the TraceTag bit set in CompAck if RespSepData has the TraceTag bit set.
  - — The TraceTag bit in the NCBWrDataCompAck response from RN must be set if either one of Comp or DBIDResp that caused the WriteData response have the TraceTag bit set.
- When an interconnect receives a packet with the TraceTag bit set, it must preserve the value and not reset the value.

Note

- Propagating the value of the TraceTag bit on a resulting cache eviction is IMPLEMENTATION DEFINED.
- The precise mechanism to trigger and utilize the TraceTag bit is IMPLEMENTATION DEFINED.
- It is expected that the TraceTag bit will be limited to single system wide use at any time.

  Some of the ways the trace tag mechanism can be used are:
  - — Debug, by tracing transaction flows through the system.
  - — Performance counting.
  - — Latency measurement.

Note

Examples, not an exhaustive list, of Request-Response pairs are:

- Snoop response, with or without data, in response to a Snoop request.
- A Snoop response in response to a SnpDVMOp request.
- Data response from SN in response to a Read request.
- Spawned requests from HN-F:
  - Snoops generated in response to a request from RN.
  - Request to SN-F generated in response to a request from RN.
- Spawned request from HN-I:
  - Read or Write request to SN-I generated in response to a request from RN.
- A CompAck from RN in response to CompData, Comp or RespSepData.
- A RetryAck response from HN or SN to any request.
- A ReadReceipt response from HN or SN to a Read request or from SN to a ReadNoSnpSep request.
- A DBIDResp response to a Write request.

Chapter 12
Memory Tagging

This chapter describes the Memory tagging mechanism and contains the following sections:

12.1 Introduction

The *Memory Tagging Extension* (MTE) is a mechanism that is used to check the correct usage of data held in memory. When a memory location is allocated for a particular use it can be also assigned a memory tag. This memory tag is held alongside that data in memory and is referred to as the Allocation Tag. When the memory location is later accessed, the Requester uses both the address of the location and the tag value that it believes is associated with the location. This tag is referred to as the Physical Address Tag or Physical Tag.

For any access where tag checking is enabled, the Physical Tag is checked against the Allocation Tag. The access always progresses as normal, and the result of the tag check determines whether or not an error condition is signaled.

This mechanism ensures that a memory access is for its expected purpose, rather than an erroneous or malicious access. It can be used at run-time to identify many common programming memory errors, such as buffer-overflow and use-after-free.

The memory tag consists of a 4-bit tag associated with each aligned 16 bytes of data in memory.

The following behavior is supported:

- Memory tagging is permitted only in requests to Normal WriteBack memory.
- Read transactions have an indication in the transaction request that determines whether the Allocation Tag value must be returned alongside the data.

  Checking of the returned Physical Tag against the Allocation Tag is performed by the Requester.

  In the case where a cache holds the data value, but does not hold the Allocation Tag value, then a Read transaction that returns both data and tag must be performed. The data returned can be dummy data.
- Read requests that require tags to be fetched must not use Forwarding snoops.
- StashOnce transactions that request Allocation Tags.

  The Allocation Tags are expected to be stashed along with the stashing of data.
- Write transactions that have a Physical Tag supplied alongside the write data that must be checked against the Allocation Tag.

  Checking of the Physical Tag against the Allocation Tag is performed by the Completer. In the case of a mismatch a notification of the failure is required.
- Write transactions that update the Allocation Tag to a new value.

  These Write transactions typically update the data at the same time. However, it is permitted to have no *Byte Enables* (BE) asserted so that only the tag is updated.
- Write transactions which pass a Dirty or Clean cache line to a downstream cache or memory controller without either updating or checking the tags.

  These Write transactions always include data and provide an indication whether the Allocation Tag value is also being passed with the data.
- Snoop transactions that return data can also return the associated Allocation Tag.

  If the tags are Dirty, they must be returned. If the tags are Clean, then returning them is optional.
- Cache Maintenance Operations must operate on both the data and the corresponding memory tags.

12.2 Message extensions

The following extensions to the CHI message definitions are used to support Memory Tagging:

Tag Provides sets of 4-bit tags, each associated with an aligned 16 bytes of data.

- Applicable on the DAT channel only.
- Size is Data_Width/32 bits.

See *Tag* on page 13-421.

TU Tag Update. Indicates which of the Allocation Tags must be updated.

- Applicable on the DAT channel only.
- Size is Data_Width/128 bits.

See *TU* on page 13-421.

TagOp Tag Operation. Indicates the operation to be performed on the tags present in the corresponding DAT channel.

- Applicable on the REQ, DAT, and RSP channels.
- Size is 2 bits.
- Value encodings are:

TagOp[1:0]	Tag operation
`0b00`	Invalid
`0b01`	Transfer
`0b10`	Update
`0b11`	Match or Fetch

See *TagOp* on page 13-421.

Note

For clarity, in the following sections TagOp values are italicized to differentiate them from data and cache line values.

12.3 Tag coherency

This section summarizes the tag coherency features.

Allocation Tags that are cached are kept hardware coherent. The coherence mechanism is the same as data coherence.

Applicable tag cached states are: Invalid, Clean, and Dirty. A cache 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 cache line is in a Unique state, it applies to both data and tags.
- When a cache line is in a Shared state, it applies to both data and tags.
- When a cache line with Dirty tags is evicted then:
    - Both data and tags must be treated as Dirty.
    - The tags must be either written back to memory or passed Dirty [_PD] by Home to another cache.
- When Clean tags are evicted from a cache they can be sent to other caches or dropped silently.
- When Clean tags are evicted with Dirty data, Clean tags can be transferred downstream of PoC along with Dirty data.

12.4 Read transaction rules

A read can optionally fetch tags along with the data. The need to return tags along with read data is determined from the value of TagOp in the request.

12.4.1 TagOp values

When TagOp value in the request is *Transfer*:

- Tags are required to be returned along with the data.
- The state of the returned tags must be the appropriate permitted cache state for the request being used.
- The number of tags to be returned is determined by the size of Data that is returned. For all Snoopable requests four tags per access must be returned.
- When required by the access, matching of Physical Tags against the Allocation Tags that are received along with read data is done at the Requester.

When TagOp value in the Request is *Fetch*:

- Tags are required to be returned along with data.
- Returned data is permitted to be dummy data. The Requester must ignore the received data irrespective of the Tag Match result.
- All tags corresponding to a cache line must be returned.
- The state of the returned tags must be Clean or Dirty.
- If Dirty tags are returned, they must be preserved, unless updated, and written back to memory.

When TagOp value in the request is *Invalid*:

- It is permitted, but not required, that tags are returned along with the data.
- If tags are returned along with the data, then they must be Clean.

Converting tags from Shared to Unique

When both the data and the tags are present at the Requester in Shared state, and the Requester requires to move the cache line to a Unique state, to update either the data or the tags or both, then the MakeReadUnique transaction with a TagOp value of *Transfer* is expected to be used. The Requester is permitted to use ReadUnique with a TagOp value of *Transfer*.

Fetching tags when Data is present

If a Requester has a cached copy of a cache line, with data Valid but Allocation tags are not Valid, and the Requester requires a Tag Match to be performed then the Requester must use a Read request to fetch the required tags. The type of the Read request and TagOp value in the request depends on:

- The operation being performed by the Requester, that is, a Load or a Store operation.
- The size of the Store operation.
- The target memory location.

A Requester in the above scenario must use:

- ReadClean with *Transfer* if the operation is a Load operation. The request can be sent from any initial data state. The initial MTE tag state is expected to be invalid but is permitted to be any state. See Table 4-14 on page 4-203 for Requester cache state transitions.
- ReadUnique with *Transfer* if the operation is a Store operation. The request can be sent from any initial data state. The initial MTE tag state is expected to be invalid but is permitted to be any state. ReadUnique, irrespective of the existence of MTE can be sent from any initial data state.

- ReadNoSnp with *Fetch* if the target memory location is Non-snoopable and the Requester guarantees to write a full cache line irrespective of the Tag Match result. The returned data must be dropped. The returned data is permitted to be dummy data. Clean tags must be returned. All tags within the cache line must be Valid.
- ReadUnique with *Fetch*, if the target memory location is Snoopable and the Requester guarantees to write a full cache line irrespective of result of the Tag Match result. The returned data must be dropped. The returned data is permitted to be dummy data. Clean or Dirty tags must be returned. All tags within the cache line must be Valid.

When responding to a ReadClean, a Home that uses a Snoop Filter to track the cached state at the Requester, must not downgrade the state of the cache line in the Snoop Filter based on the state in the response to the Requester.

Note

In prior issues of this specification, I and UCE were the only permitted initial cache line states for the ReadClean transaction. The Home that uses a Snoop Filter to track the cached states was permitted to set the state of the cached line based on the state in the response.

Permitted responses and tag state

The data and state of the cache line received with the Allocation Tags must be appropriately handled to not break coherency.

When the request TagOp value is *Transfer*, the permitted response field values are:
- *Transfer*. Indicates the returned tags are Clean.
- *Update*. Indicates the returned tags are Dirty. Data response must pass Dirty [_PD].

When the request TagOp value is *Fetch*, the permitted response field values are:
- *Transfer*. Indicates the returned tags are Clean.
- *Update*. Indicates the returned tags are Dirty. Data response must pass Dirty [_PD].

When the request TagOp value is *Invalid*, the permitted response field values are:
- *Invalid*. Indicates the returned tags are Invalid.
- *Transfer*. Indicates the returned tags are Clean.

When data and response are separately sent in a Read transaction, the TagOp field is only applicable in the Data-only message. It is inapplicable in the non-Data response message and must be set to zero.

The cache state that the tags must be held in is consistent with the type of the Read request:

- For all Read requests with a TagOp value of *Invalid*, Invalid or Clean tags must be returned.
- For ReadNoSnp with a TagOp value of *Transfer* or *Fetch*, Clean tags must be returned.
- For ReadOnce and ReadClean with a TagOp value of *Transfer*, Clean tags must be returned.
- For ReadNotSharedDirty with a TagOp value of *Transfer*, Clean or Dirty tags must be returned. Dirty tags are permitted to be returned only if the cache line state is Unique.
- For ReadShared with a TagOp value of *Transfer*, Clean or Dirty tags must be returned.
- For ReadUnique with a TagOp value of *Transfer* or *Fetch*, Clean or Dirty tags must be returned. The returned cache line state must be Unique.
- For MakeReadUnique with a TagOp value of *Transfer*, if data is included in the response then Clean or Dirty tags must be returned.
- In the case where Dirty tags are returned, the cache line returned must include pass Dirty [_PD].

When MTE is not supported for the targeted address, the response must use TagOp of *Invalid*.

For an Exclusive access sequence, it is important that the fetching of tags must avoid any form of request that will Invalidate other copies of the cache line before the Exclusive Store transaction is performed. This is typically achieved by fetching tags at the same time the Exclusive Load transaction is performed.

12.4.2 Permitted initial MTE tag states

Table 12-1 shows the permitted initial data state along with tag state for different Read transactions and permitted TagOp value in the corresponding request. The combination of tag and data states must obey the coherency rules described in *Tag coherency* on page 12-364.

Table 12-1 Permitted initial Tag states and request TagOp values in Read transactions

Request	TagOp	Data state	Tag state
ReadNoSnp	*Invalid*, *Transfer*, *Fetch*	I	Invalid
ReadOnce ReadNotSharedDirty ReadShared	*Invalid*, *Transfer*	I, UCE	Invalid
ReadClean	*Invalid*	I, UCE	Invalid
	Transfer	I, UCE	Invalid
		SC, UC	Invalid, Clean
		SD, UD	Invalid, Clean, Dirty
ReadPreferUnique	*Invalid*, *Transfer*	I, UCE	Invalid
		SC	Invalid, Clean
		SD	Invalid, Clean, Dirty
ReadUnique	*Invalid*, *Transfer*, *Fetch*	I, UCE	Invalid
		SC, UC	Invalid, Clean
		SD, UD	Invalid, Clean, Dirty
MakeReadUnique	*Invalid*	SC	Invalid, Clean
		SD	Invalid, Clean, Dirty
	Transfer	SC	Clean
		SD	Clean, Dirty

12.5 Write transactions

Different fields to support MTE are distributed between the Request and Data message of a Write transaction. The TagOp field, which indicates the operation to be performed on the tags in the WriteData message, is included in both the Request and WriteData message. The Request also includes the TagGroupID field to provide an identifier for the pass/fail response for a request that requires a Tag Match operation.

Note

The use of the TagGroupID field is IMPLEMENTATION DEFINED, typically it can be used to identify the Exception level and TTBR which a response relates to.

The TagOp value in the WriteData message is typically the same as the value in the Request message, except when either the write data is snooped out or the write is canceled. When the TagOp values in the WriteData and Write request are different, whether or not to perform a Tag Match must be decided based on the TagOp value in the WriteData request.

The WriteData message also includes a Tag Update (TU) bit per tag, which is applicable when TagOp is *Update*.

12.5.1 Permitted TagOp values

This section describes the permitted WriteData TagOp values for each of the permitted TagOp values in the Write request message.

When the TagOp field in the Request is *Invalid*, the Memory Tagging fields in the WriteData must be set to zero and ignored by the Completer.

When the TagOp field in the Request is *Transfer*, the TagOp field in the WriteData can be:

- *Transfer*: The Tags are Clean and must be handled appropriately by the Completer.
- *Invalid*: This is possible only if either the cached copy is invalidated, or the Write transaction is canceled.

When the TagOp field in the request is *Update*, the TagOp field in the WriteData can be:

- *Update*: The Dirty tags must be cached or written to memory.
- *Transfer*: The Tags are Clean. This is possible if the Dirty tags have been snooped out.
- *Invalid*: This is possible only if either the cached copy is invalidated or the Write transaction is canceled.

When the TagOp field in the Request is *Match* the TagOp field in the WriteData can be:

- *Match*: The appropriate Tag Match must be performed at the Completer.
- *Invalid*: This is possible only if the Write transaction is canceled.

12.5.2 TagOp, TU and tags relationship

This section describes the relationship between TagOp, TU and tags in different Write transactions:

- For all Write requests with TagOp *Invalid*, the Memory Tagging fields must be set to zero and ignored by the Completer.
- For WriteBackFull and WriteCleanFull with TagOp:
  - *Transfer*: Clean tags must be returned. TU bits are inapplicable and must be set to zero.
  - *Update*: All TU bits must be asserted.
  - *Match*: Is not permitted.
- For WriteBackPtl with TagOp:
  - *Transfer*: Is not permitted.
  - *Update*: Is not permitted.
  - *Match*: Is not permitted.
- For WriteNoSnpFull with TagOp:
  - *Transfer*: TU bits are inapplicable and must be set to zero. Clean tag transfer is permitted from RN to HN as well as HN to SN.
  - *Update*: All TU bits must be asserted.
  - *Match*: TU bits are inapplicable and must be set to zero.
- For WriteUniqueFull and WriteUniqueFullStash with TagOp:
  - *Transfer*: Is not permitted.
  - *Update*: All TU bits must be asserted.
  - *Match*: TU bits are inapplicable and must be set to zero.
- For WriteNoSnpPtl, WriteUniquePtl and WriteUniquePtlStash with TagOp:
  - *Transfer*: Is not permitted.
  - *Update*: Any combination of TU and BE bits, including none or all, can be asserted.
  - *Match*: TU bits are inapplicable and must be set to zero. Tag Match must be performed for only those tags that have at least one corresponding BE asserted.
- For WriteEvictFull, WriteEvictOrEvict and with TagOp:
  - *Transfer*: Clean tags must be returned. TU bits are inapplicable and must be set to zero.
  - *Update*: Is not permitted.
  - *Match*: Is not permitted.
- For WriteNoSnpZero and WriteUniqueZero only TagOp *Invalid* is permitted.

For a Write request with a TagOp of *Match* the tags within the size wrap boundary can take any value, and the tags outside of size are inapplicable and can also take any value.

12.6 Dataless transactions

MakeReadUnique and MakeUnique are the only Dataless transactions that support the use of the TagOp field. In all other Dataless transactions the TagOp field is inapplicable and must be set to all zeroes.

The permitted TagOp value in responses to MakeReadUnique with TagOp value of *Invalid* are *Invalid* and *Transfer* only. *Transfer* is permitted only in responses with data.

When a Requester uses MakeReadUnique with *Transfer* it is required that it has copies of both the data and the tags.

The permitted TagOp value in responses to MakeReadUnique with TagOp value of *Transfer* are *Invalid*, *Transfer* and *Update*. *Transfer* is permitted only in responses with data. *Update* is permitted only when the response is transferring the responsibility of updating Dirty data, that is, the response includes [_UD_PD].

TagOp value in MakeUnique request can be *Invalid* or *Update* only. A Request TagOp value of *Update* is an indication that the Requester will update the tags along with data. Home, in response to MakeUnique, is permitted to use SnpMakeInvalid only when either the Request TagOp value is *Update* or the Home knows that the Snoopee does not have Dirty tags.

The only permitted TagOp value in response to MakeUnique is *Invalid*.

Cache Maintenance Operations must operate on both the data and the corresponding memory tags. A MakeInvalid that permits dropping of Dirty data without writing to memory must write Dirty tags to memory.

12.7 Atomic transactions

TagOp is applicable in Atomic transactions. The permitted values for the field are *Invalid* and *Match*.

The Physical Tags to be matched are provided in the write data and correspond to the valid data bytes in AtomicLoad, AtomicStore, and AtomicSwap. Only one set of tag bits are applicable in these Atomic transactions because the maximum data size is 8-byte. The remaining tag bits in the set are not applicable and must be set to zero.

In AtomicCompare with a data size of up to 16 byte the valid data still corresponds to only one set of tag bits.

In AtomicCompare with data size of 32-byte the individual compare and swap data is only 16-byte. Only one set of Physical Tag bits are required to be matched when TagOp is set to *Match*. Physical Tags must be duplicated to cover both Compare and Swap data. The Completer is permitted to use either set of Physical Tags to perform Tag Match.

The permitted TagOp values in the CompData response to Non-store Atomic transactions are *Invalid* and *Transfer*.

12.8 Stash transactions

In StashOnce and StashOnceSep transactions, TagOp values of *Invalid* and *Transfer* are permitted.

For Stash snoop interaction with Memory Tagging see *Stash snoops* on page 12-374.

12.9 Snoop requests

This section describes the permitted use by Home of the following snoop types:

- *Non-Forwarding snoops*.
- *Forwarding snoops*.
- *Stash snoops* on page 12-374.

12.9.1 Non-Forwarding snoops

Home can use any applicable non-Forwarding snoop when tags are needed by the request. If the Snoop response returns data to Home but not tags, then the Home is responsible for obtaining tags before forwarding the Data response to the Requester.

If the Snoopee returns Dirty tags with TagOp value of *Update* then the snoop response state must include pass Dirty [_PD].

A Home in response to WriteUniqueFull, WriteUniqueFullStash, MakeUnique, and MakeInvalid must not use the SnpMakeInvalid snoop request unless either:

- The transaction is WriteUniqueFull, WriteUniqueFullStash or MakeUnique with a TagOp value of *Update*.
- The Home can determine that the Snoopee does not hold Dirty tags.

Note

A Requester, by sending a ReadUnique with TagOp *Fetch*, is indicating its willingness to update the full cache line but not the tags. Home must not use SnpMakeInvalid in response to such a request, to avoid losing modified tags at the Snoopee.

12.9.2 Forwarding snoops

The use by Home of the Forwarding snoop is permitted only if the request does not need to fetch tags. A Forwarding snoop is permitted to be sent even when the Snoopee has Valid tags. When responding to a Forwarding snoop, if the Snoopee has Dirty tags then it must not forward Dirty tags to the Requester nor split dirtiness or uniqueness of tags and data.

A Snoopee is always permitted to forward Clean tags to the Requester.

When tags are Clean, the Snoopee must follow the same data transfer rules as in the Non-MTE case.

When tags are Dirty the Snoopee must follow the rules:

- For the invalidating Forwarding snoop SnpUniqueFwd:
  - Must not forward data to the Requester.
  - Must return data and tags to the Home.
- For the Non-invalidating Forwarding snoops SnpCleanFwd, SnpSharedFwd, and SnpNotSharedDirtyFwd:
  - Must treat the snoop as SnpCleanFwd.

    Note

    Same as SnpNotSharedDirtyFwd.
- For SnpOnceFwd:
  - Permitted to return Dirty tags to the Home.
  - Dirty tags must be kept at the Snoopee if the data transfer to Home is not performed.
  - Dirty tags must be written back to Home if the cache line is being invalidated or Dirty data is being written back to Home.

A Snoopee is permitted to convert any Forwarding snoop to its corresponding non-Forwarding snoop.

Note

This property is similar to that in the Non-MTE case.

If the Snoopee returns Dirty tags with TagOp value of *Update*, then the Snoop response state must include pass Dirty [_PD].

12.9.3 Stash snoops

Because the SNP channel does not include a TagOp field, the Home cannot forward the Requester's TagOp intentions to the Stash target.

The permitted TagOp values in Snoop responses are:

- *Invalid* in response to SnpStash.
- *Invalid*, *Transfer*, and *Update* in response to SnpUniqueStash.
- *Invalid* in response to SnpMakeInvalidStash.

The requirements for determining the TagOp value in the Read request implied from the Data Pull request in the Snoop response are as follows:

For a Data Pull request in response to SnpStash:

- If the TagOp value in the original request is *Transfer*, then it is recommended that Clean tags are returned in response to the Data Pull request.
- If the TagOp value in the original request is *Invalid*, then it is recommended that Clean tags are returned in response to the Data Pull request if tags are available.

For a Data Pull request in response to SnpUniqueStash:

- Irrespective of the presence or absence of data in the Snoop response and for any cache state, it is recommended that, if tags are available when data is returned, Clean tags are returned in response to a Data Pull request.

12.10 Home to Slave transactions

For a Read request to the Slave:

- The permitted TagOp values are *Invalid*, *Transfer*, and *Fetch*.

For a Write request to the Slave:

- The permitted TagOp values are *Invalid*, *Transfer*, *Update*, and *Match*.

For an Atomic request to the Slave:

- The permitted TagOp values are *Invalid*, and *Match*.

A ReadNoSnp or ReadNoSnpSep with TagOp *Transfer* or *Fetch* can be used to fetch tags from the Slave. Tags can be returned from a Slave to the Home directly or sent to the Requester using DMT. When TagOp *Fetch* is used then the Slave is not required to return valid data. TagOp *Fetch* is permitted only with a data size of 64 bytes.

When memory needs to be updated with tags, WriteNoSnp with TagOp *Update* must be used. When only tags need to be updated the data byte enables can be set to all zero.

When TagOp is *Match*, the TagGroupID in WriteNoSnp and Atomic transactions to the Slave are applicable, and these values must be returned in the TagMatch response.

When Atomic operations are performed at the Slave Node, the Home is permitted to include TagOp *Match* in Atomic requests to the Slave. In both non-Store Atomic and Store Atomic transactions where the tag match is performed at the Slave, the TagMatch response TgtID must be obtained from the ReturnNID value in the request. To support this feature, this specification requires that the ReturnNID in the Atomic Store from a Home Node to a Slave Node is applicable and must be set to the same value as the SrcID in the request.

Note

Applicability of the ReturnNID in non-Store Atomic transactions is already a requirement in the previous version of this specification.

12.11 Error response

The following sub-sections describe error response handling for:

- *Tag Match*.
- *Non-Tag Match errors*.
- *MTE not supported* on page 12-377.

12.11.1 Tag Match

Write and Atomic transactions that have TagOp value of *Match* must return the results of the Tag Match operation. The results are returned using the TagMatch message. The transaction must proceed as normal irrespective of the Tag Match result.

Note

Using a separate TagMatch message adds complexity and additional messages to the Write and Atomic transactions but has the advantage that it provides a sufficiently accurate response mechanism. Using a separate response does not delay the sending of the Comp response.

The TagMatch message characteristics are:

- It is sent by the Completer performing the Tag Match operation. This can be an HN or an SN.
- The TgtID value in the message is copied from:
  - SrcID in the request if the responder is a Home Node.
  - ReturnNID in the request if the responder is a Slave Node. The ReturnNID can point to the Requester or the Home. When the Slave returns the TagMatch response to the Home then, if required by the transaction, the Home is responsible for forwarding the response to the Requester.
- The response must return the TagGroupID value from the request.
- The TraceTag field is inapplicable and can take any value.
- The Resp field value in the response indicates the Tag Match state as either pass or fail. See *Resp* on page 13-422.
- The TagMatch response can be sent as soon as the Completer can determine the result. TagMatch is permitted to be sent without waiting for data. This can occur when the Completer does not support MTE.
- The Resp value in the TagMatch response must be:
  - A Fail when MTE is not supported.
  - A Pass when MTE is supported but the Tag Match is not performed. For example, when an Exclusive write fails and the Tag Match is not performed.
  - Accurate, if the match is performed.

12.11.2 Non-Tag Match errors

Permitted Data and Non-data error responses to requests are independent of the presence or absence of Memory Tagging.

Permitted RespErr field values in the TagMatch response are OK, DERR, and NDERR. See *Error response use by transaction type* on page 9-327.

When the RespErr field in a response is NDERR then:

- The TagOp in the response to the Requester is a Don’t Care and can take any value.
- The Snoop response must be Invalid.

Poison on tags is not supported.

12.11.3 MTE not supported

When a Completer does not support MTE for the address in the request, then the Completer is permitted to send a TagOp value of *Invalid* in response to a Read transaction with a TagOp value of *Transfer* or *Fetch*. The returned tags must be set to zero. The returned tags can be cached as Clean by the Requester.

A Completer in response to a Read request to a Non-cacheable or Device memory location must set the TagOp value in the response to *Invalid*.

When a Completer does not support MTE for the address in the request, but receives a Write request with TagOp *Match*, then the Completer is still required to send a TagMatch response. The Resp field value must indicate that the Tag Match failed. The Completer is permitted, but not required, to wait for the write data before sending the TagMatch response.

12.12 Requests and permitted tag operations

Table 12-2 shows a summary of the permitted TagOp field values in the different requests.

Table 12-2 Permitted TagOp values for each request type

Requests	Tag operation				
	Invalid	**Transfer**	**Update**	**Match**	**Fetch**
ReadOnce, ReadClean, ReadShared, ReadNotSharedDirty, ReadPreferUnique	Yes	Yes	No	No	No
ReadUnique	Yes	Yes	No	No	Yes
ReadNoSnp, ReadNoSnpSep	Yes	Yes	No	No	Yes
ReadOnce*Invalid	Yes	No	No	No	No
MakeReadUnique	Yes	Yes	No	No	No
CleanShared, CleanSharedPersist, CleanSharedPersistSep	Yes	No	No	No	No
CleanInvalid, MakeInvalid	Yes	No	No	No	No
MakeUnique	Yes	No	Yes	No	No
Evict	Yes	No	No	No	No
StashOnceUnique, StashOnceSepUnique, StashOnceShared, StashOnceSepShared	Yes	Yes	No	No	No
WriteNoSnpFull	Yes	Yes	Yes	Yes	No
WriteUniqueFull, WriteUniqueFullStash	Yes	No	Yes	Yes	No
WriteNoSnpPtl, WriteUniquePtl, WriteUniquePtlStash	Yes	No	Yes	Yes	No
WriteBackFull, WriteCleanFull	Yes	Yes	Yes	No	No
WriteBackPtl	Yes	No	No	No	No
WriteEvictFull, WriteEvictOrEvict	Yes	Yes	No	No	No
WriteNoSnpFull+(P)CMO	Yes	Yes	Yes	No	No
WriteNoSnpPtl+(P)CMO	Yes	No	Yes	No	No
WriteUniqueFull+(P)CMO	Yes	No	No	No	No
WriteUniquePtl+(P)CMO	Yes	No	No	No	No
WriteBackFull+(P)CMO	Yes	Yes	Yes	No	No
WriteCleanFull+(P)CMO	Yes	Yes	Yes	No	No
WriteNoSnpZero	Yes	No	No	No	No
WriteUniqueZero	Yes	No	No	No	No

Table 12-2 Permitted TagOp values for each request type (continued)

Requests	Tag operation				
	Invalid	**Transfer**	**Update**	**Match**	**Fetch**
Atomic*	Yes	No	No	Yes	No
PrefetchTgt	Yes	Yes	No	No	No
PCrdReturn, DVMOp	Yes	No	No	No	No

The TagOp field in ReqLCrdReturn, DatLCrdReturn and RspLCrdReturn is a Don’t Care and can take any value.

The Tag and TU fields in DatLCrdReturn are a Don’t Care and can take any value.

12.13 TagOp field use summary

The following sections give a summary of the use of the TagOp field in messages in different channels:

REQ channel message:

- Read∗ and MakeReadUnique transaction:
  - TagOp field can be *Invalid*, *Transfer* or *Fetch*.
  - TagOp *Fetch* is permitted only in ReadUnique, ReadNoSnp and ReadNoSnpSep transactions.
  - TagOp field *Transfer* indicates whether Allocation Tags must be returned alongside read data.
  - TagOp field *Fetch* indicates only valid tags are required to be returned. Returned data can be dummy data.
- Write transaction:
  - TagOp field can be *Invalid*, *Transfer*, *Match* or *Update*.
  - TagOp field *Transfer* indicates Clean tags are being transferred and tags can be cached alongside the data.
  - TagOp field *Match* indicates a Match check is required between the Physical Tags in the message and the Allocation Tags at the memory location.
  - TagOp field *Update* indicates Dirty tags are being passed, which must update the Allocation Tag values.
- MakeUnique transaction:
  - TagOp field can be *Invalid* or *Update*.
  - TagOp field *Update* indicates all tags will be written to.
- Atomic transaction:
  - TagOp field can be *Invalid* or *Match*.
  - TagOp field *Match* indicates whether a Tag Match is required.
- StashOnce transaction:
  - TagOp field can be *Invalid* or *Transfer*.
  - TagOp field *Transfer* indicates whether Allocation Tags should be stashed alongside Stash data.
- For all the other REQ channel messages, the TagOp field is inapplicable and must be zero.

DAT channel message:

- For Read data the TagOp field indicates whether the Allocation Tags sent along with the data are Invalid, Clean or Dirty.
- For Snoop data, the TagOp field indicates whether the Allocation Tags sent in the Snoop response are Invalid, Clean or Dirty.
- For write data, the TagOp field indicates whether the Allocation Tags sent in the write data are Invalid, Clean, Dirty or a Match check is required. The TagOp value must be the same as in the Request message except when a snoop has downgraded the state of the tags or the write has been canceled.

RSP channel message:

- For a Comp response, the TagOp field is only used in response to a MakeReadUnique transaction and is used to indicate if responsibility for Dirty tags is being passed to the Requester.
- For all other RSP channel messages, the TagOp field is inapplicable and must be zero.

SNP message:

- The TagOp field is not present in the SNP channel.

Chapter 13
Link Layer

This chapter describes the Link layer that provides a streamlined mechanism for packet based communication between nodes and the interconnect across links. It contains the following sections:

13.1 Introduction

The Link layer provides a streamlined mechanism for packet based communication between nodes and the interconnect.

The Link layer defines:

- Packet and flit formats.
- Flow control across a link.

Figure 13-1 shows a typical system using link based communication.

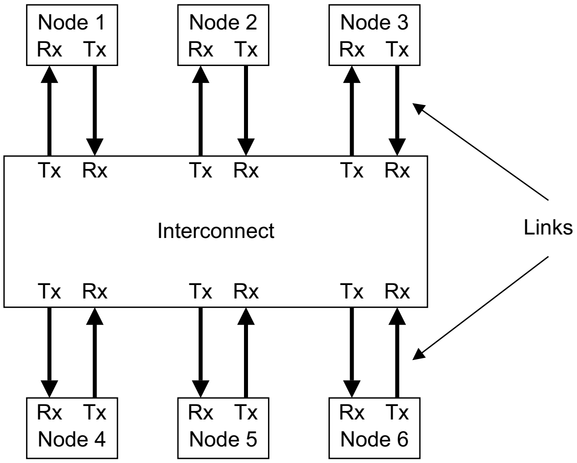


Figure 13-1 System using link based communication

Interface parity signals, which are discussed in *Use of interface parity* on page 9-339 are not included in this chapter.

13.2 Link

Flit communication occurs between a transmitter and a receiver pair.

The connection between a transmitter and a receiver is referred to as a link.

Two-way communication between a node and the interconnect requires a pair of links. Figure 13-2 shows the link requirements.

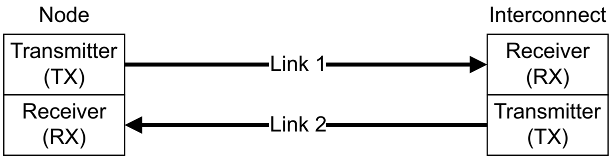


Figure 13-2 Two-way link communication

13.2.1 Outbound and inbound links

The link used by a transmitter to send packets is defined as the outbound link.

The link used by a receiver to receive packets is defined as the inbound link.

Figure 13-3 shows the outbound and inbound links at a node. The interface at the interconnect has a complementary pair of links.

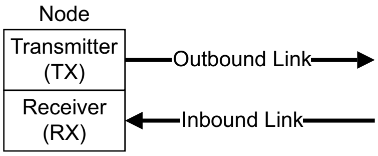


Figure 13-3 Outbound and inbound links

13.3 Flit

A flit is the basic unit of transfer in the Link layer.

Packets are formatted into flits and transmitted across a link. There are two types of flits:

Protocol flit	A Protocol flit carries a protocol packet in its payload. In this specification, every protocol packet is mapped into exactly one protocol flit.
Link flit	A Link flit carries messages associated with link maintenance. For example, a transmitter uses a Link flit to return a Link layer Credit, also referred to as an L-Credit, to the receiver during a link deactivation sequence. Link flits originate at a link transmitter and terminate at the link receiver connected at the other side of the link.

13.4 Channel

In this specification, the Link layer provides a set of channels for flit communication.

Each channel has a defined flit format that has multiple fields and some of the field widths have multiple possible values. In some cases, the defined flit format can be used on both an inbound and an outbound channel.

Table 13-1 shows the channels, and the mapping onto the RN and SN component channels.

Table 13-1 Channels' mapping onto the RN and SN component channels

Channel	Description	Usage	RN Channel	SN Channel
REQ Request	The request channel transfers flits associated with request messages such as Read requests and Write requests. See *REQ channel* on page 13-394.	All Requests	TXREQ	RXREQ
RSP Response	The response channel transfers flits associated with response messages that do not have a data payload such as write completion messages. See *RSP channel* on page 13-395.	Responses from the Completer	RXRSP	TXRSP
		Snoop Response and Completion Acknowledge	TXRSP	-
SNP Snoop	The snoop channel transfers flits associated with Snoop and SnpDVMOp Request messages. See *SNP channel* on page 13-396.	All Snoop requests	RXSNP	-
DAT Data	The data channel transfers flits associated with protocol messages that have a data payload such as read completion and WriteData messages. See *DAT channel* on page 13-397.	WriteData, and Snoop response data from an RN	TXDAT	RXDAT
		Read data	RXDAT	TXDAT

13.4.1 Channel dependencies

The following dependencies are permitted between the channels in the protocol.

For RN:

- An RN must make forward progress on the inbound SNP channel without requiring forward progress on outbound REQ channel.
- An RN is permitted to wait for forward progress on the outbound RSP channel before making forward progress on the inbound SNP channel.
- An RN is permitted to wait for forward progress on the outbound DAT channel before making forward progress on the inbound SNP channel.
- An RN must make forward progress on the inbound RSP channel without requiring forward progress on any other channel.
- An RN must make forward progress on the inbound DAT channel without requiring forward progress on any other channel.

Note

The requirement that an RN must make forward progress on the inbound RSP and DAT channel, without requiring forward progress on any other channel, means that an RN must be able to accept all Comp and CompData responses for outstanding transactions without sending any CompAck responses.

For SN:

- An SN is permitted to wait for forward progress on the outbound RSP channel before making forward progress on the inbound REQ channel.
- An SN must make forward progress on the inbound REQ channel without requiring forward progress on the outbound DAT channel.
- An SN must make forward progress on the inbound DAT channel without requiring forward progress on any other channel.

13.5 Port

A Port is defined as the set of all links at the interface of a node.

Figure 13-4 shows the relationship between links, channels, and port. See *Node interface definitions* on page 13-388 for the specific node requirements, See *Channel interface signals* on page 13-394, and Chapter 14 *Link Handshake* for signal details.

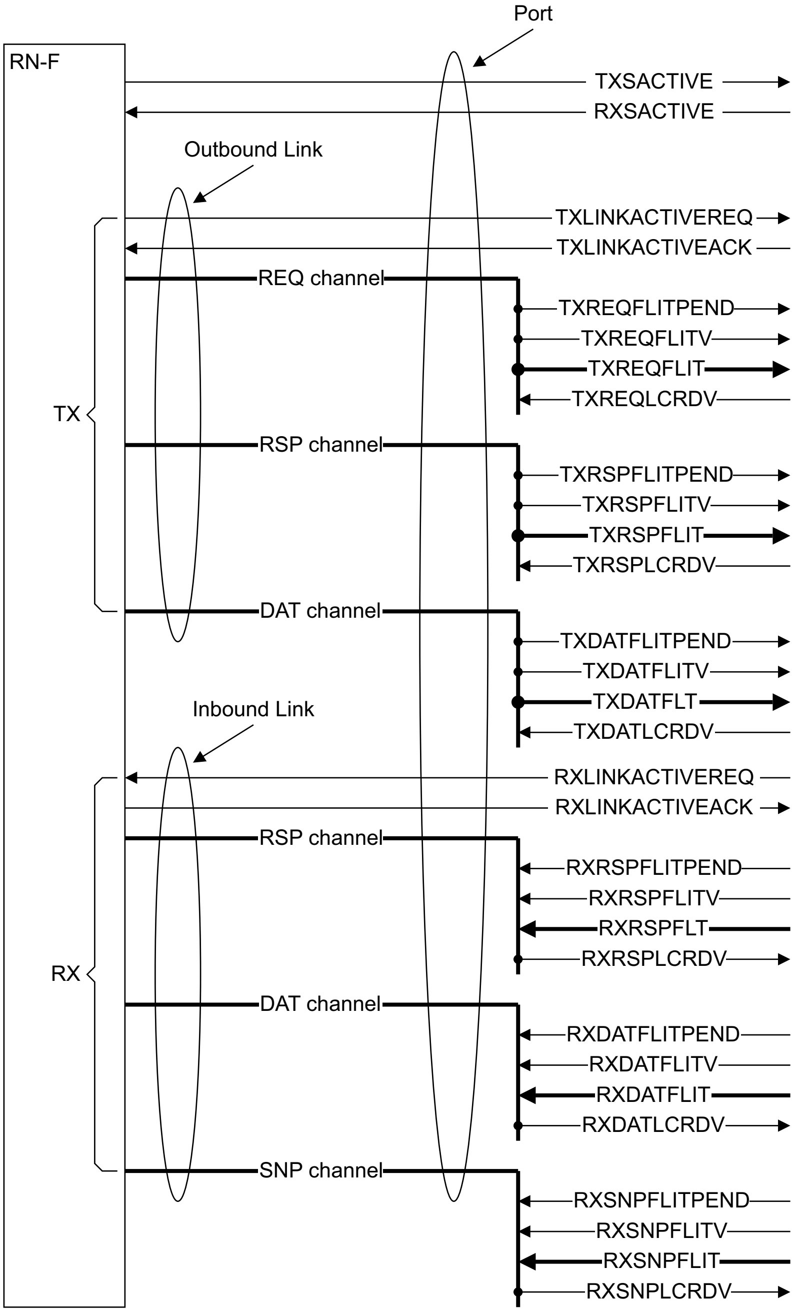


Figure 13-4 Relationship between links, channels, and port

13.6 Node interface definitions

Nodes communicate by exchanging Link flits using the node interface. This section describes the node interfaces:

- *Request Nodes*.
- *Slave Nodes* on page 13-389.

Note

The LINKACTIVE interface pins and signals used by each node for link management are described in Chapter 14 *Link Handshake*.

13.6.1 Request Nodes

This section describes the Request Node interfaces:

- *RN-F*.
- *RN-D*.
- *RN-I* on page 13-389.

RN-F

The RN-F interface uses all channels and is used by a fully coherent Requester such as a core or cluster. Figure 13-5 shows the RN-F interface.

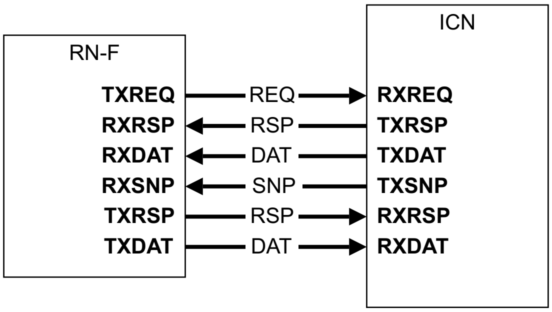


Figure 13-5 RN-F interface

RN-D

The RN-D interface uses all channels and is used by an IO coherent node that processes DVM messages. Use of the SNP channel is limited to DVM transactions. See *DVM transaction flow* on page 8-294 for details. Figure 13-6 shows the RN-D interface.

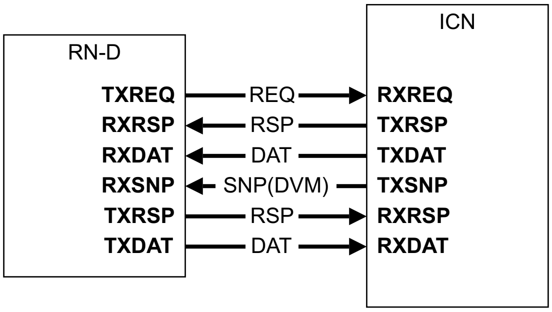


Figure 13-6 RN-D interface

RN-I

The RN-I interface uses all channels, with the exception of the SNP channel, and is used by an IO coherent Request Node such as a GPU or IO bridge. A SNP channel is not required because an RN-I node does not include a hardware-coherent cache or TLB. Figure 13-7 shows the RN-I interface.

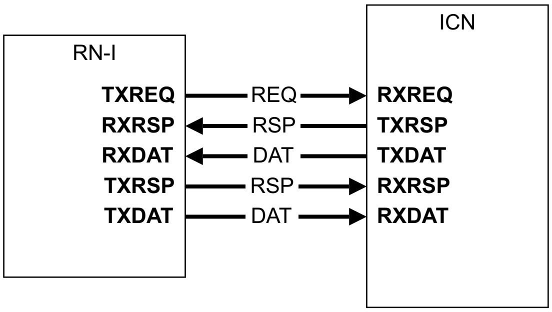


Figure 13-7 RN-I interface

13.6.2 Slave Nodes

This section describes the Slave Node interfaces:

- SN-F.
- SN-I.

SN-F and SN-I

The SN-F and SN-I interfaces are identical and use a RX request channel, a TX response channel, a TX data channel, and an RX data channel. The SN-F and SN-I receive request messages from the interconnect, and return response messages to the interconnect. However, the SN-F and SN-I receive different types of transactions. Figure 13-8 shows the SN-F and SN-I interface.

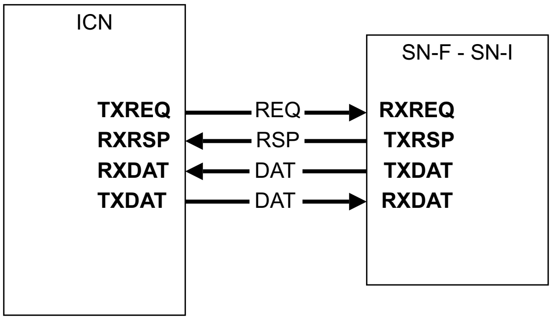


Figure 13-8 SN-F and SN-I interface

13.7 Increasing inter-port bandwidth

The available bandwidth at a node interface can be increased in several ways. Two architectural methods permitted by this specification are detailed in the following sections:

- *Multiple interfaces*.
- *Replicated channels on a single interface* on page 13-392.

13.7.1 Multiple interfaces

The simplest method for a component to increase the available bandwidth is to have multiple interfaces. A complete interface can be duplicated. The number of node interfaces that are duplicated is IMPLEMENTATION DEFINED.

Figure 13-9 shows an example of duplicated interfaces.

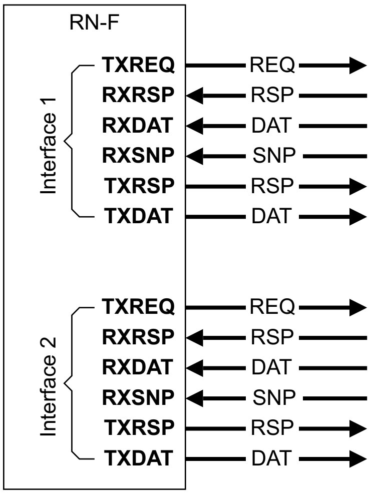


Figure 13-9 Multiple interface example

The main features of this method of increasing bandwidth across two interfaces are:

- Each interface has its own:
  - Node ID.
  - TxnID pool.
  - Set of **SActive** signals.
  - Set of **LinkActive** signals.
  - Set of **SysCo** signals.
  - Set of optional broadcast control pins.

    The presence of a physical broadcast control pin on one interface does not require it to be present on the other duplicated interfaces.
- When responding to a request, the responder must send the response on the same interface as the one used by the request.
- A snoop must be sent to the same interface that was used for the transaction that caused the allocation of a cache line.
- All channels must be replicated even when only a subset of the channels needs increased bandwidth.

Address striping

This specification permits an optional optimization where an RN can specify the address striping that it will use to select between multiple interfaces. This can be specified for any number of interfaces.

An RN is permitted to use address striping to guide its requests to the appropriate interface without declaring the striping algorithm being used.

A Home Node typically can filter snoops based on a snoop filter. If the snoop filter is precise it can track the Node ID of the Requester that cached the cache line and send a snoop for a subsequent request to the same cache line on a single interface. But a snoop filter that does not track precisely, or is sized to the number of components in the system instead of the number of RN interfaces, will not be able to isolate the single interface it needs to send the snoop on, unless it knows and uses the same address striping algorithm as the Requester.

When an RN does not declare the striping algorithm it is using, the snoop filters either need to be larger or the Home will have to send redundant snoops. This specification recommends that an RN that uses address striping advertises the striping algorithm so that the Home Nodes can make use of it.

The address striping used by an RN can be specified by a hash function.

Hash function example

This section describes a suggested hash function to be used to distribute requests across multiple REQ interfaces. The same hash function can be used to distribute snoops across multiple available SNP interfaces. The steps in generating the interface number are:

1. The cache line aligned input address is first filtered through a predefined Hash Mask. The most significant address bits that are not used must be set to all zero before filtering the address. In this example, the result of the filtering is Mask_Result.
2. The individual bits of Mask_Result are XORed to obtain the target interface.

Where the number of interfaces is a power of two:

- For 2 interfaces:

  Interfaces[0] = Mask_Result[n-1] ^ Mask_Result[n-2] ... Mask_Result[7] ^ Mask_Result[6]

- For 4 interfaces:

  Interfaces[1:0] = Mask_Result[n-1:n-2] ^ Mask_Result[n-3:n-4] ... Mask_Result[9:8] ^ Mask_Result[7:6]

- For 8 interfaces:

  Interfaces[2:0] = Mask_Result[n-1:n-3] ^ Mask_Result[n-4:n-6] ... Mask_Result[11:9] ^ Mask_Result[8:6]

This example does not cover the situation where the number of interfaces is not a power of two.

13.7.2 Replicated channels on a single interface

A more efficient method to increase the available interface bandwidth, rather than replicating a complete interface through more complex methods, is to selectively replicate the channels that require greater bandwidth.

Figure 13-10 shows an example of replicated channels.

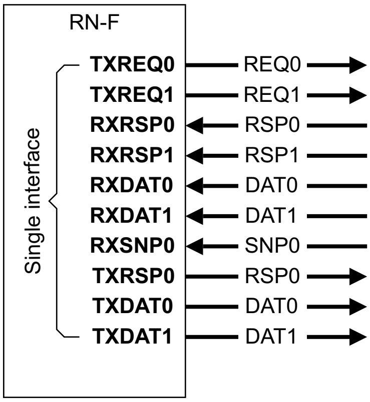


Figure 13-10 Replicated channels example

Features

The main features of this method of increasing available bandwidth are described in this section.

Each channel can be selectively replicated. There are no restrictions on which channels are replicated. Typically, replication of a channel is based on the expected bandwidth required on that channel. For example, in Figure 13-10, TXREQ is duplicated as TXREQ0, TXREQ1, whereas RXSNP is not duplicated and has only RXSNP0. The characteristics of the replicated channel interface are:

- All replicated DAT subchannels corresponding to a single DAT channel must be of the same width.
- The complete interface must use:
  - Same Node ID.
  - Single transaction ID pool.
- No cross-channel relationship:
  - Request on TXREQ0 can give a response on either RXRSP0 or RXRSP1.
  - All existing cross-channel dependencies remain.
  - Multiple response messages for a single request can come on any subchannel. For example, DBIDResp for a write transaction is received on RXRSP0 whereas the corresponding Comp can be received on RXRSP1.
- Like non-replicated channels, replicated channels do not provide any in-channel ordering guarantees.
- All link crediting is done on a subchannel basis:
  - Cannot use the credit for TXREQ0 to send the flit on TXREQ1.
- Protocol credits are for the combined TXREQ channel.
- There is no support for powering down a subchannel individually.
- The two parts of a DVM snoop can come on either subchannel. Each part can be on a different subchannels.
- The number of subchannels on two connected interfaces must match.
- Credits are required to be provided by the receiver on all subchannels.

- There must be only one set of **SActive**, **LinkActive** and **SysCo** signals, and optional broadcast control pins.
- Interface property CCF_Wrap_Order is not permitted to be set to True when the interface includes replicated DAT channels.

Note

The subchannel to use, when multiple subchannels are free, can be an implementation specific heuristic.

13.8 Channel interface signals

This section describes the channel interfaces. It contains the following sections:

- *REQ channel*.
- *RSP channel* on page 13-395.
- *SNP channel* on page 13-396.
- *DAT channel* on page 13-397.

13.8.1 REQ channel

Figure 13-11 shows the REQ channel interface pins, where R is the width of **REQFLIT**.

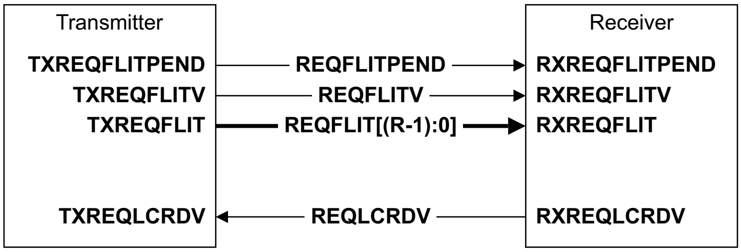


Figure 13-11 REQ channel interface pins

Table 13-2 shows the REQ channel interface signals.

Table 13-2 REQ channel interface signals

Signal	Description
REQFLITPEND	Request Flit Pending. Early indication that a request flit might be transmitted in the following cycle. See *Flit level clock gating* on page 14-433.
REQFLITV	Request Flit Valid. The transmitter sets this signal HIGH to indicate when **REQFLIT[(R-1):0]** is valid.
REQFLIT[(R-1):0]	Request Flit. See *Request flit* on page 13-398 for a description of the request flit format.
REQLCRDV	Request L-Credit Valid. The receiver sets this signal HIGH to return a request channel L-Credit to a transmitter. See *L-Credit flow control* on page 14-431.

13.8.2 RSP channel

Figure 13-12 shows the RSP channel interface pins, where T is the width of **RSPFLIT**. The same interface is used for both inbound and outbound RSP channels.

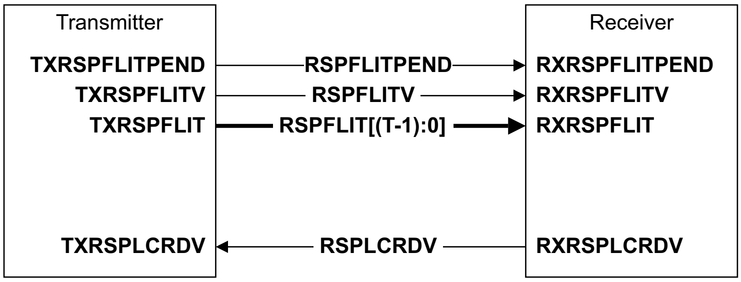


Figure 13-12 RSP channel interface pins

Table 13-3 shows the RSP channel interface signals.

Table 13-3 RSP channel interface signals

Signal	Description
RSPFLITPEND	Response Flit Pending. Early indication that a response flit might be transmitted in the following cycle. See *Flit level clock gating* on page 14-433.
RSPFLITV	Response Flit Valid. The transmitter sets this signal HIGH to indicate when **RSPFLIT[(T-1):0]** is valid.
RSPFLIT[(T-1):0]	Response Flit. See *Response flit* on page 13-400 for a description of the response flit format.
RSPLCRDV	Response L-Credit Valid. The receiver sets this signal HIGH to return a response channel L-Credit to a transmitter. See *L-Credit flow control* on page 14-431.

13.8.3 SNP channel

Figure 13-13 shows the SNP channel interface pins, where S is the width of **SNPFLIT**.

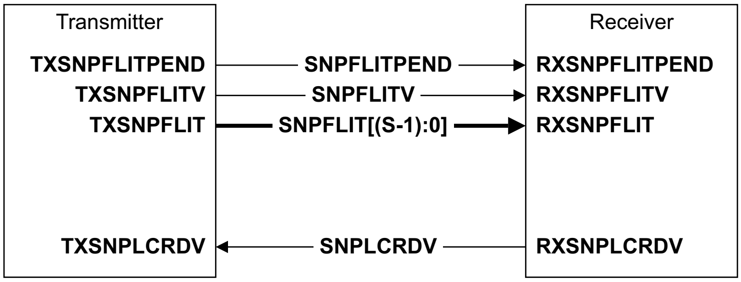


Figure 13-13 SNP channel interface pins

Table 13-4 shows the SNP channel interface signals.

Table 13-4 SNP channel interface signals

Signal	Description
SNPFLITPEND	Snoop Flit Pending. Early indication that a snoop flit might be transmitted in the following cycle. See *Flit level clock gating* on page 14-433.
SNPFLITV	Snoop Flit Valid. The transmitter sets this signal HIGH to indicate when **SNPFLIT[(S-1):0]** is valid.
SNPFLIT[(S-1):0]	Snoop Flit. See *Snoop flit* on page 13-401 for a description of the snoop flit format.
SNPLCRDV	Snoop L-Credit Valid. The receiver sets this signal HIGH to return a snoop channel L-Credit to a transmitter. See *L-Credit flow control* on page 14-431.

13.8.4 DAT channel

Figure 13-14 shows the DAT channel interface pins, where D is the width of **DATFLIT**. The same interface is used for both inbound and outbound DAT channels.

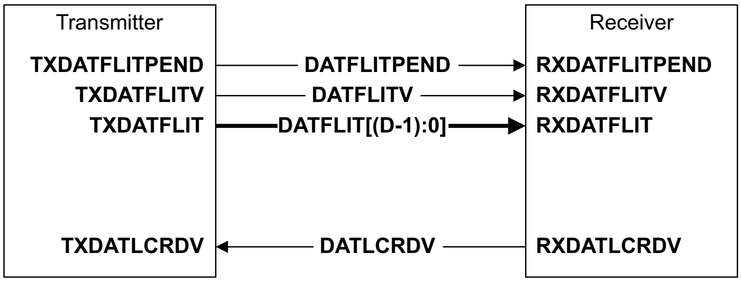


Figure 13-14 DAT channel interface pins

Table 13-5 shows the DAT channel interface signals.

Table 13-5 DAT channel interface signals

Signal	Description
DATFLITPEND	Data Flit Pending. Early indication that a data flit might be transmitted in the following cycle. See *Flit level clock gating* on page 14-433.
DATFLITV	Data Flit Valid. The transmitter sets this signal HIGH to indicate when **DATFLIT[(D–1):0]** is valid.
DATFLIT[(D-1):0]	Data Flit. See *Data flit* on page 13-402 for a description of the data flit format.
DATLCRDV	Data L-Credit Valid. The receiver sets this signal HIGH to return a data channel L-Credit to a transmitter. See *L-Credit flow control* on page 14-431.

13.9 Flit packet definitions

This section defines the flit format. See:

- *Request flit*.
- *Response flit* on page 13-400.
- *Snoop flit* on page 13-401.
- *Data flit* on page 13-402.

13.9.1 Request flit

Table 13-6 shows the Request flit format in a REQ channel packet starting at bit zero.

Table 13-6 Request flit format

REQFLIT[(R-1):0] format

Field	Field width	Comments
QoS	4	-
TgtID	7 to 11	Width determined by NodeID_Width
SrcID	7 to 11	Width determined by NodeID_Width
TxnID	12	-
ReturnNID StashNID {(NodeID_Width - 7)’b0, SLCRepHint[6:0]}	7 to 11	Used for DMT Used in Stash transactions - Used in cache line replacement algorithms
StashNIDValid Endian Deep	1	Used in Stash transactions Used in Atomic transactions Used in CleanSharedPersist* transactions
ReturnTxnID[11:0] {6’b0, StashLPIDValid, StashLPID[4:0]}	12	Used for DMT SBZ Used in Stash transactions Used in Stash transactions
Opcode	7	-
Size	3	-
Addr	RAW = 44 to 52	Width determined by Req_Addr_Width (RAW)
NS	1	-
LikelyShared	1	-
AllowRetry	1	-
Order	2	-
PCrdType	4	-
MemAttr	4	-
SnpAttr DoDWT	1	-

Table 13-6 Request flit format (continued)

REQFLIT[(R-1):0] format		
Field	**Field width**	**Comments**
{GroupIDExt[2:0], LPID[4:0]} PGroupID[7:0] StashGroupID[7:0] TagGroupID[7:0]	8	- Used in Excl, ReadNoSnp and WriteNoSnp transactions Used in CleanSharedPersistSep transaction Used in the StashOnceSep transaction Used in Memory Tagging writes
Excl SnoopMe	1	Used in Exclusive transactions Used in Atomic transactions
ExpCompAck	1	-
TagOp	2	Memory Tagging
TraceTag	1	-
MPAM	M = 0	No MPAM bus
	M = 11	-
RSVDC	Y = 0	No RSVDC bus
	Y = 4, 8, 12, 16, 24, 32	Permitted RSVDC bus widths
Total	R= (87 + RAW + M + Y) to (99 + RAW + M + Y)	

13.9.2 Response flit

Table 13-7 shows the Response flit format in a RSP channel packet starting at bit zero,

Table 13-7 Response flit format

RSPFLIT[(T-1):0] format

Field	Field width	Comments
QoS	4	-
TgtID	7 to 11	Width determined by NodeID_Width
SrcID	7 to 11	Width determined by NodeID_Width
TxnID	12	-
Opcode	5	-
RespErr	2	-
Resp	3	-
FwdState[2:0] DataPull[2:0]	3	Used for DCT Used in Stash transactions
CBusy	3	-
DBID[11:0] {4'b0, PGroupID[7:0]} {4'b0, StashGroupID[7:0]} {4'b0, TagGroupID[7:0]}	12	- - Used in PCMO transactions - Used in the StashDone response - Used in Excl, ReadNoSnp and WriteNoSnp transactions
PCrdType	4	-
TagOp	2	-
TraceTag	1	-
Total	T = 65 to 73	-

13.9.3 Snoop flit

Table 13-8 shows the Snoop flit format in a SNP channel packet starting at bit zero.

Table 13-8 Snoop flit format

SNPFLIT[(S-1):0] format

Field	Field width	Comments
QoS	4	-
SrcID	7 to 11	Width determined by NodeID_Width
TxnID	12	-
FwdNID	7 to 11	Width determined by NodeID_Width
FwdTxnID[11:0] {6'b0, StashLPIDValid, StashLPID[4:0]} {4'b0, VMIDExt[7:0]}	12	Used for DCT SBZ Used in Stash transactions Used in Stash transactions SBZ Used to extend VMID value
Opcode	5	-
Addr	SAW = 41 to 49	Width determined by Req_Addr_Width (SAW)
NS	1	-
DoNotGoToSD	1	-
RetToSrc	1	-
TraceTag	1	-
MPAM	M = 0	No MPAM bus
	M = 11	-
Total	S = 51 + SAW + M to 59 + SAW+ M	-

13.9.4 Data flit

Table 13-9 shows the Data flit format in a DAT channel packet starting at bit zero.

The number of data flits required is dependent on the number of data bytes, and the data bus width. See *Data packetization* on page 2-138.

The data channel interface supports a 128-bit, 256-bit, and 512-bit data bus width. There are three data flit formats defined, one for each of the three data bus widths supported at the data channel interface.

DataCheck (DC) field width is either zero or equal to the width of the Data field divided by 8.

Poison (P) field width is either zero or equal to the width of the Data field divided by 64.

Table 13-9 Data flit fields

DATFLIT[D-1:0] format

Field	Field Width	Comments
QoS	4	-
TgtID	7 to 11	Width determined by NodeID_Width
SrcID	7 to 11	Width determined by NodeID_Width
TxnID	12	-
HomeNID	7 to 11	Width determined by NodeID_Width
Opcode	4	-
RespErr	2	-
Resp	3	-
{1’b0, FwdState[2:0]} {1’b0, DataPull[2:0]} DataSource[3:0]	4	SBZ Used for DCT SBZ Used in Stash transactions Indicates Data source in a response
CBusy	3	-
DBID[11:0]	12	-
CCID	2	-
DataID	2	-
TagOp	2	Memory tagging
Tag	DW/32 = 4, 8, 16	
TU	DW/128 = 1, 2, 4	
TraceTag	1	-
RSVDC	Y = 0	No RSVDC bus
	Y = 4, 8, 12, 16, 24, 32	Permitted RSVDC bus widths
BE	DW/8 = 16, 32, 64	-
Data	DW = 128, 256, 512	DW = Data bus width

Table 13-9 Data flit fields (continued)

DATFLIT[D-1:0] format

Field	Field Width	Comments
DataCheck (DC)	0 or DW/8 = 16, 32, 64	-
Poison (P)	0 or DW/64 = 2, 4, 8	-
Total	D = (221 to 233) + Y + DC + P	DW = 128 bit Data
	D = (370 to 382) + Y + DC + P	DW = 256 bit Data
	D = (668 to 680) + Y + DC + P	DW = 512 bit Data

13.10 Protocol flit fields

A Protocol flit is identified by a non-zero value in the opcode field. All the flit fields defined in this section are applicable for a Protocol flit. The following sections describe the encoding of the Protocol flit fields:

- *TgtID* on page 13-405.
- *SrcID* on page 13-405.
- *HomeNID* on page 13-405.
- *ReturnNID* on page 13-405.
- *FwdNID* on page 13-405.
- *LPID* on page 13-406.
- *PGroupID* on page 13-406.
- *GroupIDExt* on page 13-406.
- *StashNID* on page 13-406.
- *StashNIDValid* on page 13-407.
- *StashLPID* on page 13-407.
- *StashLPIDValid* on page 13-407.
- *StashGroupID* on page 13-408.
- *TxnID* on page 13-408.
- *ReturnTxnID* on page 13-408.
- *FwdTxnID* on page 13-408.
- *DBID* on page 13-408.
- *Opcode* on page 13-409.
- *Deep* on page 13-414.
- *Addr* on page 13-415.
- *NS* on page 13-415.
- *Size* on page 13-415.
- *MemAttr* on page 13-416.
- *SnpAttr* on page 13-416.
- *DoDWT* on page 13-417.
- *LikelyShared* on page 13-417.
- *Order* on page 13-417.
- *Excl* on page 13-418.
- *Endian* on page 13-418.
- *AllowRetry* on page 13-418.
- *ExpCompAck* on page 13-419.
- *SnoopMe* on page 13-419.
- *RetToSrc* on page 13-419.
- *DataPull* on page 13-419.
- *DoNotGoToSD* on page 13-420.
- *QoS* on page 13-420.
- *PCrdType* on page 13-420.
- *TagOp* on page 13-421.
- *Tag* on page 13-421.
- *TU* on page 13-421
- *TagGroupID* on page 13-421.
- *TraceTag* on page 13-422.
- *MPAM* on page 13-422.
- *VMIDExt* on page 13-422.
- *Resp* on page 13-422.
- *FwdState* on page 13-424.
- *CBusy* on page 13-424.

- *RespErr* on page 13-425.
- *Data* on page 13-425.
- *CCID* on page 13-425.
- *DataID* on page 13-426.
- *BE* on page 13-426.
- *DataCheck* on page 13-426.
- *Poison* on page 13-426.
- *DataSource* on page 13-427.
- *SLCRepHint* on page 13-427.
- *RSVDC* on page 13-427.

13.10.1 TgtID

Target Identifier associated with the message. The node ID of the component to which the message is targeted. This is used by the interconnect to determine the port to which the message is sent. See *Details of transaction identifier fields* on page 2-89.

13.10.2 SrcID

Source Identifier associated with the message. The node ID of the component from which the message is sent. This is used by the interconnect to determine the port from which the message has been sent. See *Details of transaction identifier fields* on page 2-89.

13.10.3 HomeNID

Home Identifier associated with the original request. The Requester uses the value in this field to determine the TgtID of the CompAck to be sent in response to CompData. See *Details of transaction identifier fields* on page 2-89.

Applicable in CompData and DataSepResp from the Slave and Home.

Inapplicable and must be zero in all other Data messages.

13.10.4 ReturnNID

Return NID. Identifies the node to which the Slave sends a CompData response, or a DataSepResp response, or a Persist response. The value can be either the NID of Home or the Requester that originated the transaction. See *Details of transaction identifier fields* on page 2-89.

Applicable from Home to Slave in ReadNoSnp, ReadNoSnpSep, CleanSharedPersistSep, WriteNoSnp, Combined Write, and Atomic requests.

Inapplicable and must be zero for all other requests. For Stash requests, the same bits in the packet are used for StashNID.

13.10.5 FwdNID

Forward NID. Identifies the Requester to which the CompData response can be forwarded. The value must be the NID of the Requester that initiated the transaction. See *Details of transaction identifier fields* on page 2-89.

Applicable in Forward type snoops.

Inapplicable and must be zero in all other Snoop requests.

13.10.6 LPID

Logical Processor Identifier. Used in conjunction with the SrcID to uniquely identify the logical processor that generated the request. See *Logical Processor Identifier* on page 2-115.

Applicable in the following requests:

- ReadNoSnp for Exclusive, Non-snoopable, Non-cacheable, or Device access.
- WriteNoSnp for Exclusive, Non-snoopable, Non-cacheable, or Device access.
- ReadClean, ReadNotSharedDirty, ReadShared, and ReadPreferUnique for Exclusive Access.
- CleanUnique and MakeReadUnique for Exclusive Access.

In requests, when applicable, the same bits in the packet are used for TagGroupID, PGroupID and StashGroupID.

13.10.7 PGroupID

Persistence Group ID. Used by a Requester to process different sets of CleanSharedPersistSep transactions by grouping them together and identifying each using PGroupID.

Applicable in the CleanSharedPersistSep request and the Persist and CompPersist responses.

Inapplicable and must be set to zero in all other requests and responses.

In requests, when applicable, the same bits in the packet are used for LPID, TagGroupID and StashGroupID.

In responses, when applicable, the same bits in the packet are used for DBID, TagGroupID and StashGroupID.

13.10.8 GroupIDExt

Group ID Extension. A 3-bit REQ channel field used in the following manner in Request transactions:

- Persistent CMO transactions:
  - In Persistent CMO requests, used to extend the persistent group ID size to 8 bits;
    PGroupID[7:0] = {GroupIDExt[2:0], LPID[4:0]}.
- StashOnce transactions:
  - In StashOnceSep requests, used to extend the stash group ID value size to 8bits:
    StashGroupID[7:0] = {GroupIDExt[2:0], LPID[4:0]}.
- Write transactions and Atomic transactions:
  - In Write and Atomic requests that have the TagOp set to *Match*, used to extend the Tag group ID value size to 8 bits:
    TagGroupID[7:0] = {GroupIDExt[2:0], LPID[4:0]}.
  - The precise contents of the TagGroupID are IMPLEMENTATION DEFINED. Typically, the fields are expected to contain Exception Level, TTBR value, and CPU identifier.

13.10.9 StashNID

Stash NID. Identifies the target of the stash request. Provides a valid stash target value when the corresponding StashNIDValid bit is asserted. See *Stash target identifiers* on page 7-291.

Applicable in Stash requests.

Inapplicable and must be zero for all other requests. For ReadNoSnp and ReadNoSnpSep requests, the same bits in the packet are used for ReturnNID.

13.10.10 StashNIDValid

Stash NID valid. Indicates if StashNID field has a valid value. See *Stash target identifiers* on page 7-291.

Applicable in Stash requests, inapplicable and must be set to zero in all other requests.

Table 13-10 shows the StashNIDValid value encoding.

Table 13-10 StashNIDValid value encoding

StashNIDValid	Description
0	StashNID field value is inapplicable and must be set to zero.
1	The StashNID field in the Request has a valid Stash target.

13.10.11 StashLPID

Stash Logical Processor ID. Provides a valid Logical Processor target value within the Request Node specified by StashNID. See *Stash target identifiers* on page 7-291.

Applicable in Stash requests and Stash type snoop requests.

Inapplicable and must be zero for all other requests. For ReadNoSnp requests the same bits in the packet are used for ReturnTxnID.

Inapplicable and must be zero for all other snoop requests. For Fwd snoops the same bits in the packet are used for FwdTxnID and for SnpDVMOp snoops the same bits in the packet are used for VMIDExt.

13.10.12 StashLPIDValid

Stash LPID valid. Indicates if the StashLPID field has a valid value. See *Stash target identifiers* on page 7-291.

Applicable in Stash requests and Stash type snoop requests, inapplicable and must be set to zero in all other requests.

Table 13-11 shows the StashLPIDValid value encoding.

Table 13-11 StashLPIDValid value encoding

StashLPIDValid	Description
0	StashLPID field value is inapplicable and must be set to zero.
1	The StashLPID field in the Request has a valid Stash target.

Table 13-12 shows the valid StashNIDValid and StashLPIDValid encodings.

Table 13-12 Valid StashNIDValid and StashLPIDValid encodings

StashNIDValid	StashLPIDValid	Comments
0	0	Stash target is not specified
0	1	Reserved
1	0	Only a target RN is specified
1	1	Both target RN and LPID are specified

13.10.13 StashGroupID

Stash Group ID. Used by a Requester to process different sets of StashOnceSep transactions by grouping them together and identifying each using StashGroupID.

Applicable only in the StashOnceSep request and StashDone response.

Inapplicable and must be set to zero in all other requests and responses.

In requests, when applicable, the same bits in the packet are used for LPID, TagGroupID and PGroupID.

In responses, when applicable, the same bits in the packet are used for DBID, TagGroupID and PGroupID.

13.10.14 TxnID

Transaction Identifier associated with the message. When there are multiple outstanding transactions from a given source node they will each use a unique transaction ID. See *Details of transaction identifier fields* on page 2-89.

Link flits do not have a unique ID. Table 13-13 shows the link flit TxnID field value encodings.

Table 13-13 Link flit TxnID Encodings

TxnID	Description
`0x000`	L-Credit return
`0x001` to `0x3FF`	Reserved

13.10.15 ReturnTxnID

Return TxnID. Identifies the value the Slave must use in the TxnID field of the CompData, and DataSepResp response. It can be either the TxnID generated by Home for this transaction or the TxnID in the Request packet from the Requester that originated the transaction. See *Details of transaction identifier fields* on page 2-89.

Applicable only in ReadNoSnp, ReadNoSnpSep, WriteNoSnp, Combined Write, and Atomic requests from Home to Slave.

Inapplicable and must be set to zero for all other requests. For Stash requests the same bits in the packet are used for StashLPID.

13.10.16 FwdTxnID

Identifies the TxnID of the original Request associated with the Snoop transaction. See *Details of transaction identifier fields* on page 2-89.

Applicable in Forward type snoops.

Inapplicable and must be set to zero in all other snoop requests. For Stash snoops the same bits in the packet are used for StashLPID and for SnpDVMOp snoops the same bits in the packet are used for VMIDExt.

13.10.17 DBID

Data Buffer Identifier. The DBID field value in the response packet from a Completer is used as the TxnID for CompAck or WriteData sent from the Requester.

In Snoop responses with Data Pull, this field value indicates the value to be used in the TxnID field of Data Pull response messages. See *Details of transaction identifier fields* on page 2-89.

In responses, when applicable, the same bits in the packet are used for PGroupID, StashGroupID and TagGroupID.

13.10.18 Opcode

Specifies the operation to be carried out. The Opcode encodings are specific to each channel. See:

- *REQ channel opcodes*.
- *RSP channel opcodes* on page 13-412.
- *SNP channel opcodes* on page 13-413.
- *DAT channel opcodes* on page 13-414.

REQ channel opcodes

Table 13-14 shows the opcodes for the request channel.

Table 13-14 REQ channel opcodes

Opcode[5:0]	Request command	
	Opcode[6] = 0	**Opcode[6] = 1**
`0x00`	ReqLCrdReturn	Reserved
`0x01`	ReadShared	MakeReadUnique
`0x02`	ReadClean	WriteEvictOrEvict
`0x03`	ReadOnce	WriteUniqueZero
`0x04`	ReadNoSnp	WriteNoSnpZero
`0x05`	PCrdReturn	Reserved
`0x06`	Reserved	Reserved
`0x07`	ReadUnique	StashOnceSepShared
`0x08`	CleanShared	StashOnceSepUnique
`0x09`	CleanInvalid	Reserved
`0x0A`	MakeInvalid	Reserved
`0x0B`	CleanUnique	Reserved
`0x0C`	MakeUnique	ReadPreferUnique
`0x0D`	Evict	Reserved
`0x0E`	Reserved (EOBarrier)	Reserved
`0x0F`	Reserved (ECBarrier)	Reserved
`0x10`	Reserved	WriteNoSnpFullCleanSh
`0x11`	ReadNoSnpSep	WriteNoSnpFullCleanInv
`0x12`	Reserved	WriteNoSnpFullCleanShPerSep
`0x13`	CleanSharedPersistSep	Reserved
`0x14`	DVMOp	WriteUniqueFullCleanSh
`0x15`	WriteEvictFull	Reserved
`0x16`	Reserved (WriteCleanPtl)	WriteUniqueFullCleanShPerSep
`0x17`	WriteCleanFull	Reserved

Table 13-14 REQ channel opcodes (continued)

Opcode[5:0]	Request command	
	Opcode[6] = 0	**Opcode[6] = 1**
`0x18`	WriteUniquePtl	WriteBackFullCleanSh
`0x19`	WriteUniqueFull	WriteBackFullCleanInv
`0x1A`	WriteBackPtl	WriteBackFullCleanShPerSep
`0x1B`	WriteBackFull	Reserved
`0x1C`	WriteNoSnpPtl	WriteCleanFullCleanSh
`0x1D`	WriteNoSnpFull	Reserved
`0x1E`	Reserved	WriteCleanFullCleanShPerSep
`0x1F`	Reserved	Reserved
`0x20`	WriteUniqueFullStash	WriteNoSnpPtlCleanSh
`0x21`	WriteUniquePtlStash	WriteNoSnpPtlCleanInv
`0x22`	StashOnceShared	WriteNoSnpPtlCleanShPerSep
`0x23`	StashOnceUnique	Reserved
`0x24`	ReadOnceCleanInvalid	WriteUniquePtlCleanSh
`0x25`	ReadOnceMakeInvalid	Reserved
`0x26`	ReadNotSharedDirty	WriteUniquePtlCleanShPerSep
`0x27`	CleanSharedPersist	Reserved
`0x28` - `0x2F`	AtomicStore	Reserved
`0x30` - `0x37`	AtomicLoad	Reserved
`0x38`	AtomicSwap	Reserved
`0x39`	AtomicCompare	Reserved
`0x3A`	PrefetchTgt	Reserved
`0x3B` - `0x3F`	Reserved	Reserved

Note

Prior to CHI Issue E, the following requests were not supported:

- ReadPreferUnique.
- MakeReadUnique.
- StashOnceSep.
- WriteEvictOrEvict.
- WriteNoSnpZero.
- WriteUniqueZero.
- Combined Write.

Table 13-15 shows the sub-opcodes for AtomicStore and AtomicLoad.

Table 13-15 Sub-codes for AtomicStore and AtomicLoad

Opcode[5:3]		Opcode[2:0]	Operation
AtomicStore	AtomicLoad		
`101`	`110`	`000`	ADD
		`001`	CLR
		`010`	EOR
		`011`	SET
		`100`	SMAX
		`101`	SMIN
		`110`	UMAX
		`111`	UMIN

RSP channel opcodes

Table 13-16 shows the opcodes for the response channel.

Table 13-16 RSP channel opcodes encoding

Opcode[4:0]	Response
`0x0`	RespLCrdReturn
`0x1`	SnpResp
`0x2`	CompAck
`0x3`	RetryAck
`0x4`	Comp
`0x5`	CompDBIDResp
`0x6`	DBIDResp
`0x7`	PCrdGrant
`0x8`	ReadReceipt
`0x9`	SnpRespFwded
`0xA`	TagMatch
`0xB`	RespSepData
`0xC`	Persist
`0xD`	CompPersist
`0xE`	DBIDRespOrd
`0xF`	Reserved
`0x10`	StashDone
`0x11`	CompStashDone
`0x12-0x13`	Reserved
`0x14`	CompCMO
`0x15-0x1F`	Reserved

Note

Prior to CHI Issue E, the following responses were not supported:

- DBIDRespOrd.
- TagMatch.
- StashDone.
- CompStashDone.
- CompCMO.

SNP channel opcodes

Table 13-17 shows the opcodes for the snoop channel.

Table 13-17 SNP channel opcodes encoding

Opcode[4:0]	Snoop command
`0x00`	SnpLCrdReturn
`0x01`	SnpShared
`0x02`	SnpClean
`0x03`	SnpOnce
`0x04`	SnpNotSharedDirty
`0x05`	SnpUniqueStash
`0x06`	SnpMakeInvalidStash
`0x07`	SnpUnique
`0x08`	SnpCleanShared
`0x09`	SnpCleanInvalid
`0x0A`	SnpMakeInvalid
`0x0B`	SnpStashUnique
`0x0C`	SnpStashShared
`0x0D`	SnpDVMOp
`0x0E` - `0x0F`	Reserved
`0x10`	SnpQuery
`0x11`	SnpSharedFwd
`0x12`	SnpCleanFwd
`0x13`	SnpOnceFwd
`0x14`	SnpNotSharedDirtyFwd
`0x15`	SnpPreferUnique
`0x16`	SnpPreferUniqueFwd
`0x17`	SnpUniqueFwd
`0x18` - `0x1F`	Reserved

Note

Prior to CHI Issue E, the following Snoop requests were not supported:

- SnpPreferUnique.
- SnpPreferUniqueFwd.
- SnpQuery.

DAT channel opcodes

Table 13-18 shows the opcodes for the data channel.

Table 13-18 DAT channel opcodes encoding

Opcode[3:0]	Data command
`0x0`	DataLCrdReturn
`0x1`	SnpRespData
`0x2`	CopyBackWrData
`0x3`	NonCopyBackWrData
`0x4`	CompData
`0x5`	SnpRespDataPtl
`0x6`	SnpRespDataFwded
`0x7`	WriteDataCancel
`0x8` - `0xA`	Reserved
`0xB`	DataSepResp
`0xC`	NCBWrDataCompAck
`0xD-0xF`	Reserved

13.10.19 Deep

Deep persistence. Used by the Requester to indicate that the Persist response must not be sent until all earlier writes are written to the final destination.

Applicable in the CleanSharedPersist* request.

Inapplicable and must be set to zero in all other requests.

When Deep is deasserted:

- The Completer must send Persist response after all the earlier writes have reached the *Point of Persistence*(PoP).
  - — The PoP is the point at which it is guaranteed that sufficient time is available to make the data persistent after loss of power.

When Deep is asserted:

- The Completer must send the Persist response only after all earlier writes are written to the final destination, not just the PoP.
  - — The final destination is the point at which no time is required to make the data persistent after loss of power, thus preserving data even when battery failure occurs.

If the receiver of the request does not support the Deep attribute, then it can ignore the attribute value and treat the request as having the Deep attribute deasserted.

An error response must not be given to indicate that deep persistence is not supported.

13.10.20 Addr

Address. Specifies the address associated with the message.

This specification supports a *Physical Address* (PA) of 44 to 52 bits. This permits the REQ and SNP packets to support 49 to 53 bits of *Virtual Address* (VA) in DVM operations.

- Request messages support a 44 to 52 bit address field, Addr[(43-51):0].
- Snoop messages support a 41 to 49 bit address field, Addr[(43-51):3]:
  - Addr[(43-51):6] specifies the aligned address of the 64-byte cache line.
  - Addr[5:4] indicates the 16-byte critical chunk within the cache line. See *Critical Chunk Identifier* on page 2-141.
  - Addr[3] is relevant in SnpDVMOp, for all other Snoop packets it is Don't Care and can take any value.

13.10.21 NS

Non Secure. Indicates a Non-secure access or a Secure access. See *Non-secure bit* on page 2-128.

Table 13-19 shows the NS field value encoding.

Table 13-19 NS value encoding

NS	Description
0	Secure access
1	Non-secure access

13.10.22 Size

Size. Specifies the size of the data associated with the transaction. See *Data size* on page 2-136.

Table 13-20 shows the Size field value encodings.

Table 13-20 Size field value encodings

Size[2:0]	Bytes
`0b000`	1
`0b001`	2
`0b010`	4
`0b011`	8
`0b100`	16
`0b101`	32
`0b110`	64
`0b111`	Reserved

13.10.23 MemAttr

Memory Attribute. Memory attribute associated with the transaction.

Table 13-21 shows the MemAttr value encodings.

Table 13-21 MemAttr value encodings

MemAttr[3:0]	Description
[3]	Allocate hint bit. Indicates whether or not the cache receiving the transaction is recommended to allocate the transaction: **0** Recommend that it does not allocate. **1** Recommend that it allocates.
[2]	Cacheable bit. Indicates a Cacheable transaction for which the cache, when present, must be looked up in servicing the transaction: **0** Non-cacheable. Looking up a cache is not required. **1** Cacheable. Looking up a cache is required.
[1]	Device bit. Indicates if the memory type associated with the transaction is Device or Normal: **0** Normal memory type. **1** Device memory type.
[0]	Early Write Acknowledge bit. Specifies the Early Write Acknowledge status for the transaction: **0** Early Write Acknowledge not permitted. **1** Early Write Acknowledge permitted.

See *Memory Attributes* on page 2-128.

13.10.24 SnpAttr

Snoop Attribute. Specifies the snoop attribute associated with the transaction.

Table 13-22 shows the SnpAttr value encoding.

Table 13-22 SnpAttr value encoding

SnpAttr	Snoop attribute
0	Non-snoopable
1	Snoopable

See *Snoop Attribute* on page 2-134.

13.10.25 DoDWT

Supports DWT. The characteristics are:

- Only applicable in WriteNoSnpFull, WriteNoSnpPtl and Combined Write requests from Home to Slave.
- It is inapplicable and must be set to zero in all other requests.
- The bit shares the same field as SnpAttr.

Table 13-23 shows the DoDWT value encoding.

Table 13-23 DoNotDWT value encoding

DoDWT	DBIDResp.TgtID value	DBIDResp.TxnID value
0	Set to the SrcID value of the request	Set to TxnID value in the request
1	Set to the ReturnNID value in the request	Set to the ReturnTxnID value in the request

See *Direct Write-data Transfer* on page 2-70 and *Write and CMO combined request* on page 2-72.

13.10.26 LikelyShared

Likely Shared. Indicates whether the requested data is likely to be shared with another RN. See *Likely Shared* on page 2-133.

Table 13-24 shows the LikelyShared field value encoding.

Table 13-24 LikelyShared value encoding

LikelyShared	Description
0	Not likely to be shared by another RN
1	Likely to be shared by another RN

13.10.27 Order

Specifies the ordering requirements for a transaction. See *Ordering* on page 2-116 for more information on the ordering requirements.

Table 13-25 shows the Order field value encodings.

Table 13-25 Order value encodings

Order[1:0]	Description	Note
`0b00`	No ordering required	
`0b01`	Request accepted	Applicable in Read request from HN-F to SN-F, and HN-I to SN-I.
	Reserved	Reserved in all other cases
`0b10`	Request Order/Ordered Write Observation	Not applicable in requests from HN-F to SN-F.
`0b11`	Endpoint Order, which also includes Request Order	Not applicable in requests from HN-F to SN-F.

13.10.28 Excl

Exclusive. Indicates that the corresponding transaction is an Exclusive type transaction. The Exclusive bit must only be used with the following transactions:

- ReadNotSharedDirty.
- ReadShared.
- ReadClean.
- ReadPreferUnique.
- CleanUnique.
- MakeReadUnique.
- ReadNoSnp.
- WriteNoSnp.

Table 13-26 shows the Excl value encoding.

Table 13-26 Excl value encoding

Excl	Description
0	Normal transaction
1	Exclusive transaction

See *Exclusive transactions* on page 6-278.

13.10.29 Endian

Endian. Indicates the endianness of Data in an Atomic transaction. See *Endianness* on page 2-140.

Applicable in Atomic requests, inapplicable in all other requests and must be set to zero.

Table 13-27 shows the Endian value encoding.

Table 13-27 Endian value encoding

Endian	Description
0	Little Endian
1	Big Endian

13.10.30 AllowRetry

Allow Retry. Specifies that the request is being sent without a P-Credit and that the target can determine if a retry response is given. See *Transaction Retry mechanism* on page 2-149.

Table 13-28 shows the AllowRetry value encoding.

Table 13-28 AllowRetry value encodings

AllowRetry	Description
0	RetryAck response not permitted
1	RetryAck response permitted

13.10.31 ExpCompAck

Expect CompAck. Indicates that the transaction will include a CompAck response.

Table 13-29 shows the ExpCompAck value encoding.

Table 13-29 ExpCompAck value encoding

ExpCompAck	Description
0	Transaction does not include a CompAck response
1	Transaction includes a CompAck response

13.10.32 SnoopMe

SnoopMe. Indicates that Home must determine whether to send a snoop to the Requester. See *Atomics* on page 2-78.

Only applicable in Atomic requests.

Table 13-30 shows the SnoopMe value encoding.

Table 13-30 SnoopMe value encoding

SnoopMe	Description
0	Home does not need to send a snoop to the Requester.
1	Home must send a Snoop to the Requester if it determines the cache line might be present at the Requester.

13.10.33 RetToSrc

Return to Source. Requesting Snoopee to return a copy of the cache line to Home.

Applicable in all Snoops except SnpDVMOp.

For RetToSrc bit semantics see *Returning Data with Snoop response* on page 4-232.

13.10.34 DataPull

Data Pull. Indicates the inclusion of a Read request, also referred to as a Data Pull, in the Snoop response. See *Snoop requests and Data Pull* on page 7-286.

Applicable in SnpResp and SnpRespData response to a Stash request, not applicable in all other Snoop responses.

When the DataPull bit is set in a SnpRespData message it must be set in all packets of that response message.

Table 13-31 shows the DataPull field value encodings.

Table 13-31 DataPull value encodings

DataPull[2:0]	Description	Comment
`0b000`	No Read	Inclusion of Data Pull in the Snoop response
`0b001`	Read	
`0b010`- `0b111`	-	Reserved

13.10.35 DoNotGoToSD

Do not transition to SD state. An attribute in a snoop request indicating if a Snoopee is required to not transition to SD state. See *Do not transition to SD* on page 4-233.

Applicable, and can take any value in:

- SnpOnce, SnpOnceFwd.
- SnpClean, SnpCleanFwd.
- SnpNotSharedDirty, SnpNotSharedDirtyFwd.
- SnpShared, SnpSharedFwd.
- SnpPreferUnique, SnpPreferUniqueFwd.

Applicable, and must be set to 1 in:

- SnpStashShared, SnpStashUnique.
- SnpUnique, SnpUniqueFwd. SnpUniqueStash.
- SnpCleanShared.
- SnpCleanInvalid.
- SnpMakeInvalid, SnpMakeInvalidStash.
- SnpQuery.

The field is inapplicable, and must be set to 0 in SnpDVMOp.

Table 13-32 shows the DoNotGoToSD value encoding.

Table 13-32 DoNotGoToSD value encoding

DoNotGoToSD	Description
0	Permitted to transition to SD state.
1	Transitioning to SD state is not permitted. If already in SD state, then must exit SD state in response to the snoop. The bit value can be ignored by the Snoopee if the Snoop request is SnpOnce or SnpOnceFwd.

13.10.36 QoS

Quality of Service priority level. Ascending values of QoS indicate higher priority levels. See Chapter 10 *Quality of Service* for more information.

13.10.37 PCrdType

Protocol Credit Type. Indicates the type of credit being granted or returned. See *Transaction Retry mechanism* on page 2-149.

Table 13-33 shows the PCrdType value encodings.

Table 13-33 PCrdType value encodings

PCrdType	Description
`0x0` - `0xF`	P-Credit type 0 to 15 respectively.

13.10.38 TagOp

Tag Operation. Indicates the operation to be performed on the tags present in the corresponding DAT channel.

Table 13-34 shows the TagOp value encodings.

Table 13-34 TagOp value encodings

TagOp[1:0]	Tag Operation	Description
`0b00`	Invalid	The tags are not valid.
`0b01`	Transfer	• The tags are Clean. • Tag Match does not need to be performed. • All tags corresponding to data in the packet must be transferred. For Snoopable transactions, partial tag transfer is not supported. • TU field is not applicable and must be set to zero.
`0b10`	Update	• The Allocation Tag values have been updated and are Dirty. • The Tags in memory should be updated. Only the Tags that have TU asserted must be updated.
`0b11`	Match Fetch	• Match: — The Physical Tags in the write must be checked against the Allocation Tag values obtained from memory. — The Match Tag operation must be enabled for only those tags that have BE asserted. — TU field is not applicable and must be set to zero. • Fetch: — Tags must be fetched. — All tags must be fetched. — Permitted, but not required, to fetch Valid data.

TagOp value in a Retry request must be the same as in the original request.

13.10.39 Tag

Tag[4∗n-1:0]. Provides n sets of 4-bit tags, each associated with a 16-byte, aligned address location.

[Tag[((4∗n)-1) : 4∗(n-1)] corresponds to Data[((128∗n)-1) : 128∗(n-1)]

13.10.40 TU

TU(n-1:0). Tag Update. Indicates which of the Allocation Tags must be updated. There is one TU bit for each tag. Only valid in Snoop responses and Write transactions which update the Allocation Tags. Must be set to zero in all other transactions.

TU[n-1] corresponds to Tag[(4∗n)-1 : 4∗(n-1)]

13.10.41 TagGroupID

Tag Group ID. Provides an identifier for the pass/fail response for a request that requires a Tag Match operation. Precise contents is IMPLEMENTATION DEFINED. Typically expected to contain Exception Level, TTBR value, and CPU identifier.

13.10.42 TraceTag

Trace Tag. A bit in a packet used to tag the packets associated with a transaction for tracing purposes.

Table 13-35 shows the TraceTag field value encoding.

Table 13-35 TraceTag value encoding

TraceTag	Description
0	Packet is not tagged.
1	Packet is tagged.

See Chapter 11 *System Debug, Trace, and Monitoring*.

13.10.43 MPAM

Memory System Performance Resource Partitioning and Monitoring. It is used to efficiently utilize the memory resources among users and to monitor their use. See *MPAM* on page 11-355.

13.10.44 VMIDExt

Virtual Machine Identifier extension. It is used to extend VMID value from 8-bits to 16-bits. See *DVMOp payload* on page 8-305.

13.10.45 Resp

Response Status. The Resp field must have the same value in all data flits of a multi-flit data transfer.

Table 13-36 shows the Resp value encodings.

Table 13-36 Resp value encodings

Resp[2:0]	Description
Resp[2]	PassDirty. Indicates that the data included in the response message is Dirty with respect to memory and that the responsibility of writing back the cache line is being passed to the recipient of the response message. **0** Returned data is not Dirty. **1** Returned data is Dirty and the responsibility of writing back the cache line is being passed on.
Resp[1:0]	For snoop responses, this field indicates the final state of the snooped RN-F. For completion responses, this field indicates the final state in the RN. For WriteData responses, this field indicates the state of the data in the RN when the data is sent. For TagMatch responses, Resp[0] alone indicates a Tag Match pass or fail.

Table 13-37 shows the valid Resp value encodings.

Table 13-37 Valid Resp value encodings for different message types

Response Type	Resp[2:0]	State	Notes
Snoop responses	`0b000`	I	Final state of the snooped RN-F.
	`0b001`	SC	
	`0b010`	UC, UD	
	`0b011`	SD	
	`0b100`	I_PD	Final state of the snooped RN-F. Responsibility for updating memory is passed to Home.
	`0b101`	SC_PD	
	`0b110`	UC_PD	
	`0b111`	-	Reserved.
Comp responses	`0b000`	I	Final state of the requesting RN-F.
	`0b001`	SC	
	`0b010`	UC	
	`0b011`	-	Reserved.
	`0b100`	-	
	`0b101`	-	
	`0b110`	UD_PD	Final state of the requesting RN-F. Responsibility for updating memory is passed to the Requester.
	`0b111`	SD_PD	
WriteData responses	`0b000`	I	State of the cache line at the RN-F when data is sent.
	`0b001`	SC	
	`0b010`	UC	
	`0b011`	-	Reserved.
	`0b100`	-	
	`0b101`	-	
	`0b110`	UD_PD	State of the cache line at the RN-F when data is sent. Responsibility for updating memory is passed to Home.
	`0b111`	SD_PD	
TagMatch responses	`0b000`	Fail	Part of the Tag Match operation.
	`0b001`	Pass	
	`0b010` - `0b111`	-	Reserved.

13.10.46 FwdState

Forward State. Indicates the state in the CompData sent from the Snoopee to the Requester. Applicable in SnpRespFwded and SnpRespDataFwded, inapplicable in all other Snoop responses and must be set to zero.

Table 13-38 shows the FwdState value encodings.

Table 13-38 FwdState value encodings

FwdState[2:0]	Description
FwdState[2]	PassDirty. **0** Forwarded data is not Dirty. **1** Forwarded data is Dirty and the responsibility of writing back the cache line is passed on to the Requester.
FwdState[1:0]	Indicates the final state at the Requester. See Table 13-39.

Table 13-39 enumerates the FwdState value encodings.

Table 13-39 Valid FwdState value encodings

FwdState[2:0]	State	Comment
`0b000`	I	Final state at the Requester.
`0b001`	SC	
`0b010`	UC	
`0b011`	-	Reserved.
`0b100`	-	
`0b101`	-	
`0b110`	UD_PD	Final state at the Requester. Responsibility for updating memory is passed to the Requester.
`0b111`	SD_PD	

13.10.47 CBusy

Completer Busy. A mechanism for the Completer of a transaction to indicate its current level of activity. The CBusy value encodings are IMPLEMENTATION DEFINED. See *Completer Busy* on page 11-357.

13.10.48 RespErr

Response Error. This field indicates the error status of the response. See Chapter 9 *Error Handling*.

Table 13-40 shows the RespErr value encodings.

Table 13-40 RespErr value encodings

RespErr[1:0]	Description
`0b00`	Normal Okay. Indicates that either: • The Normal access was successful. • The Exclusive access failed.
`0b01`	Exclusive Okay. Indicates that either the read or write portion of an Exclusive access was successful.
`0b10`	Data Error.
`0b11`	Non-data Error.

13.10.49 Data

Data payload. This is the data payload that is being transported in a Data packet.

The following data bus widths are supported:

- 128-bit.
- 256-bit.
- 512-bit.

See *Data packetization* on page 2-138.

13.10.50 CCID

Critical Chunk Identifier. The CCID indicates the critical 128-bit chunk of the data that is being requested. See *Critical Chunk Identifier* on page 2-141.

Table 13-41 shows the CCID value encodings.

Table 13-41 CCID value encodings

CCID[1:0]	Critical data chunk
`0b00`	Data[127:0]
`0b01`	Data[255:128]
`0b10`	Data[383:256]
`0b11`	Data[511:384]

13.10.51 DataID

Data Identifier. The DataID indicates the relative position of the data chunk within the 512-bit cache line that is being transferred. See *Data packetization* on page 2-138.

Table 13-42 shows the DataID value encodings.

Table 13-42 DataID and the bytes within a packet for different data widths

DataID	Data Width		
	128-bit	256-bit	512-bit
`0b00`	Data[127:0]	Data[255:0]	Data[511:0]
`0b01`	Data[255:128]	Reserved	Reserved
`0b10`	Data[383:256]	Data[511:256]	Reserved
`0b11`	Data[511:384]	Reserved	Reserved

13.10.52 BE

Byte Enable. Indicates if the byte of data corresponding to this byte enable bit is valid. The BE field is defined for write data, DVM payload, and snoop response data transfers. For read response data transfers, this field is inapplicable and can take any value. It consists of a bit for each data byte in the DAT flit. See *Byte Enables* on page 2-137.

Table 13-43 shows the BE value encodings.

Table 13-43 BE value encodings

BE	Byte enable
0	Corresponding byte of data is not valid.
1	Corresponding byte of data is valid.

13.10.53 DataCheck

Data Check. Used to supply the DataCheck bit for the corresponding byte of Data. See *Data Check* on page 9-338.

13.10.54 Poison

Indicates if the 64-bit chunk of data corresponding to a Poison bit is poisoned, that is, has an error, and must not be consumed. See *Poison* on page 9-337.

Table 13-44 shows the Poison value encodings.

Table 13-44 Poison value encodings

Poison	64-bit chunk poisoned
0	Corresponding 64-bit chunk is not poisoned.
1	Corresponding 64-bit chunk is poisoned.

13.10.55 DataSource

Data Source. Identifies the Sender of the data response. See *DataSource value assignment* on page 11-350.

Applicable in CompData and DataSepResp responses in Read and Atomic transactions, and SnpRespData and SnpRespDataPtl responses in Non-stash type Snoop transactions. DataSource is not applicable, and must be set to zero, in all other responses.

Table 13-45 shows the DataSource value encodings.

Table 13-45 DataSource value encodings

DataSource	Description	Comment
`0b0000`	DataSource is not supported	Applicable only to a non-memory component
`0b0001` - `0b0101`	IMPLEMENTATION DEFINED	See *Suggested DataSource values* on page 11-351
`0b0110`	PrefetchTgt was useful	Indication from memory that the earlier sent prefetch was useful or memory received a prefetch.
`0b0111`	PrefetchTgt was not useful	Indication from memory that the earlier sent prefetch was not useful or memory did not receive a prefetch.
`0b1000` - `0b1111`	IMPLEMENTATION DEFINED	-

13.10.56 SLCRepHint

SLC Replacement Hint. Forwards cache replacement hints from the Requesters to the *System Level Caches* (SLC) in the interconnect, See *SLC replacement hint* on page 11-353.

13.10.57 RSVDC

Reserved for customer use. Any value is valid in a Protocol flit. Propagation of this field through the interconnect is IMPLEMENTATION DEFINED.

This field is applicable to the REQ and DAT channels as follows:
- The presence of this field is optional.
- The permitted field widths are 4-bit, 8-bit, 12-bit, 16-bit, 24-bit and 32-bit.
- The field widths:
  - Can be different between REQ and DAT channels.
  - Need not be the same across all REQ channels in the system.
  - Need not be the same across all DAT channels in the system.

When connecting TX and RX flit interfaces that have mismatched RSVDC widths:

- The corresponding lower-order bits of the RSVDC field must be connected at each side of the interface.
- The higher-order RSVDC bits at the RX interface that do not have corresponding bits at the TX interface must be tied LOW.

13.11 Link flit

A link flit is used to return L-Credits to the receiver during a link deactivation sequence. Link flits originate at a link transmitter and terminate at the link receiver on the other side of the link.

A link flit is identified by a zero value in the Opcode field. The TxnID field of the link flit is required to be zero. The remaining fields are not used and can be any value. See *Opcode* on page 13-409 for the link flit type encoding.

Chapter 14
Link Handshake

This chapter describes the link handshake requirements. It contains the following sections:

14.1 Clock, and initialization

This section specifies the AMBA 5 CHI requirement for global clock and reset signals.

14.1.1 Clock

This architecture specification does not define a specific clocking microarchitecture, but it is expected that all devices, interconnects, etc. will include one or more clocks that can be relied upon by other Link layer functions that require synchronous communication. A generic clock signal is referred to as **CLK** in the following sections, where applicable.

14.1.2 Reset

This architecture specification does not define a specific reset microarchitecture, but it is expected that all devices, interconnects, etc will include a specific reset deassertion event that can be relied upon by other Link layer functions. A generic reset signal is referred to as **RESETn** in the following sections, where applicable.

14.1.3 Initialization

During reset the following interface signals must be deasserted by the component:

- **TX***LCRDV**.
- **TX***FLITV**.
- **TXLINKACTIVEREQ** and **RXLINKACTIVEACK**.

The earliest point after reset that it is permitted to begin driving these signals HIGH is at a rising **CLK** edge after **RESETn** is HIGH.

All other signals can be any value.

14.2 Link layer Credit

This section describes the *Link layer Credit* (L-Credit) mechanism. Information is transferred across an interface channel by the use of L-Credits. To transfer one flit from the transmitter to the receiver the transmitter must have obtained an L-Credit.

14.2.1 L-Credit flow control

An L-Credit is sent from the receiver to the transmitter by asserting the appropriate **LCRDV** signal for a single clock cycle. There is one **LCRDV** signal for each channel. See *Channel interface signals* on page 13-394 for the **LCRDV** signal naming for each channel.

Each transfer of a flit from the transmitter to the receiver consumes one L-Credit.

The minimum number of L-Credits that a receiver can provide is one. The maximum number of L-Credits that a receiver can provide is 15.

A receiver must guarantee that it can accept all the flits for which it has issued L-Credits.

When the link is active, the receiver must provide L-Credits in a timely manner without requiring any action on the part of the transmitter.

Note

An L-Credit cannot be used in the cycle it is received.

14.3 Low power signaling

This section describes the signaling used to enhance the low power operation of the interface. There are several different levels of operation:

Flit Level Clock Gating

This technique is used to provide a cycle by cycle indication of the activity of each of the channels of the interface. For each channel an additional signal is provided to indicate if a transfer might occur in the following cycle. This signaling permits local clock gating of certain registers associated with the interface.

Link Activation

Link activation and deactivation is supported to permit the interface to be taken to a safe state, so that both-sides of the interface can enter a low power state that permits them to be either clock-gated or power-gated.

Protocol Activity Indication

The Protocol layer activity indication is used by components to indicate if there are ongoing transactions in progress. The Protocol layer activity indication can be used to influence the decision to use other low power techniques.

14.4 Flit level clock gating

The **FLITPEND** signal associated with a channel is used to indicate if a valid flit is going to be sent in the next clock cycle. There is one **FLITPEND** signal for each channel. See *Channel interface signals* on page 13-394 for the **FLITPEND** signal naming for each channel.

The requirements for the use of **FLITPEND** are:

- It is required that the signal is asserted exactly one cycle before a flit is sent from the transmitter.
- When asserted it is permitted, but not required, that the transmitter sends a flit in the next cycle.
- When deasserted, it is required that the transmitter does not send a flit in the next cycle.
- A transmitter is permitted to keep the signal permanently asserted. It might do this, for example, if it is unable to determine in advance when a flit is to be sent.
- A transmitter is permitted to assert this signal when it does not have an L-Credit.
- A transmitter is permitted to assert and then deassert this signal without sending a flit.

Figure 14-1 shows an example of the use of the **FLITPEND** signal.

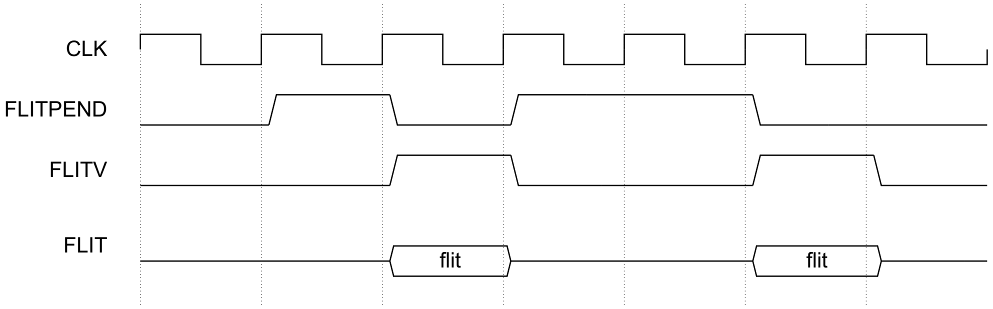


Figure 14-1 FLITPEND indicating a valid flit in next cycle

14.5 Interface activation and deactivation

A mechanism is provided for an entire interface to move between a full running operational state and a low power state. When moving between operational states, including when exiting from reset, it is important that the exchange of L-Credits, and also the exchange of Link flits, is carefully controlled to avoid the loss of flits or credits.

On exit from reset, or when moving to a full running operational state, the interface will start in an idle state and the transfer of flits can only commence when L-Credits have been exchanged. L-Credits can only be exchanged when the Sender of the credits knows the receiver is ready to receive them.

A two signal, four phase, handshake mechanism is used. This two signal interface is used for all channels traveling in the same direction, rather than being required for each individual channel. An entire interface uses a total of four signals, two signals are used for all the transmit channels and two signals are used for all the receive channels.

14.5.1 Request and Acknowledge handshake

For the purposes of description, the two signal Request and Acknowledge signaling is described using the signal names **LINKACTIVEREQ** and **LINKACTIVEACK**.

This section describes the operation of the **LINKACTIVEREQ** and **LINKACTIVEACK** handshake pairs for all channels moving in one direction. *Transmit and receive link Interaction* on page 14-440 describes the interaction between the handshake pairs for the transmit channels and those for the receive channels.

For a single channel, or group of channels traveling in the same direction, Figure 14-2 shows the relationship between the Payload, Credit, **LINKACTIVEREQ** and **LINKACTIVEACK** signals.

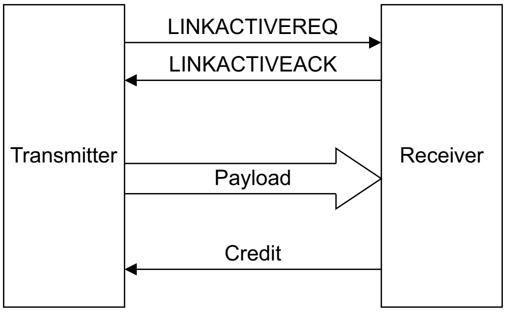


Figure 14-2 Relationship between Payload, Credit and LINKACTIVE signals

As Figure 14-2 shows, during normal operation the transmitter, which sends the payload flits, requires a credit before it can send a flit. A credit is passed from the receiver when it has resources available to accept a flit:

- On exit from reset, credits are held by the receiver and must be passed to the transmitter before flit transfer can begin.
- During normal operation, there is an ongoing exchange of flits and credits between the two sides of the interface.
- Before entering a low power state, the sending of payload flits must be stopped and all credits must be returned to the receiver, this effectively returns the interface to the same state that it was at immediately after reset.

Four states are defined for the interface operation:

RUN	There is an ongoing exchange of flits and credits between the two components.
STOP	The interface is in a low power state and it is not operational. All credits are held by the receiver and the transmitter is not permitted to send any flits.
ACTIVATE	This state is used when moving from the STOP state to the RUN state.
DEACTIVATE	This state is used when moving from the RUN state to the STOP state.

RUN and STOP are stable states and when one of these states is entered a channel can remain in this state for an indefinite period of time.

DEACTIVATE and ACTIVATE are transient states and it is expected that when one of these states is entered a channel will move to the next stable state in a relatively short period of time.

Note

The specification does not define a maximum period of time in a transient state, but it is expected that for any given implementation it is deterministic.

The state is determined by the **LINKACTIVEREQ** and **LINKACTIVEACK** signals. Figure 14-3 shows the relationship between the four states.

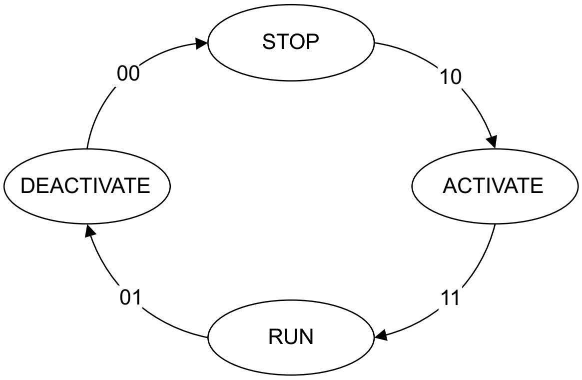


Figure 14-3 Request and Acknowledge handshake states

Table 14-1 shows the mapping of the states to the **LINKACTIVEREQ** and **LINKACTIVEACK** signals.

Table 14-1 Mapping of states to the LINKACTIVE signals

	LINKACTIVEREQ	LINKACTIVEACK
STOP	0	0
ACTIVATE	1	0
RUN	1	1
DEACTIVATE	0	1

Table 14-2 on page 14-436 describes the behavior of both the transmitter and the receiver of a single link for each of the four states.

Table 14-2 Behavior for each Request and Acknowledge state

State	Transmitter Behavior	Receiver Behavior
STOP	The transmitter has no credits and must not send any flits. The transmitter is guaranteed not to receive any credits. The transmitter must assert **LINKACTIVEREQ** to move to the ACTIVATE state if it has flits to send.	The receiver is guaranteed not to receive any flits. The receiver must not send any credits.
ACTIVATE (ACT)	The transmitter must not send any flits. The transmitter must be prepared to receive credits in this state, although it must not use them until in the RUN state. The transmitter remains in the ACTIVATE state while it is waiting for the receiver to acknowledge the move to the RUN state. **Note** The transmitter will only receive credits in the ACTIVATE state when there is a race between the receiver sending credits and asserting **LINKACTIVEACK** to move to the RUN state.	The receiver is guaranteed not to receive any flits. The receiver must not send any credits. The ACTIVATE state is a transient state and the receiver controls the move to the RUN state by asserting **LINKACTIVEACK**. The receiver must assert **LINKACTIVEACK** and move to the RUN state before sending credits. It is permitted to assert **LINKACTIVEACK** and send a credit in the same cycle. **Note** It can appear that a receiver has sent credits in the ACTIVATE state if there is a race between the receiver sending credits and asserting **LINKACTIVEACK** to move to the RUN state.
RUN	The transmitter can receive credits. The transmitter can send flits when it has credits available. The transmitter deasserts **LINKACTIVEREQ** to exit from this state if it wants to move to a low power state.	The receiver can receive flits corresponding to the credits it has sent. The receiver sends credits when it has resources available to accept further flits. The receiver must remain in the RUN state until it observes the deassertion of **LINKACTIVEREQ**.
DEACTIVATE (DEACT)	The transmitter must return credits using Protocol flits or L-Credit return flits. It is recommended that the transmitter enters the DEACTIVATE state only when it has no more Protocol flits to send. Therefore, it is expected that the transmitter will return credits using only L-Credit return flits. The transmitter must be prepared to continue receiving credits. For each additional credit received it must send an L-Credit return flit to return the credit. The transmitter remains in the DEACTIVATE state while it is waiting for the receiver to acknowledge the move to the STOP state. At this point, it will be guaranteed to receive no more credits.	During this state the receiver stops sending credits and collects all returned credits. The receiver must be prepared to receive flits, other than Link flits to return credits, in this state. This is not expected, but can occur. The receiver is permitted to send credits when first entering this state. However, it must have stopped sending credits and had all credits returned before exiting this state. The receiver will receive L-Credit return flits until all credits are returned. The receiver must wait for all credits to be returned before deasserting **LINKACTIVEACK**. **Note** The receiver will only receive flits in the DEACTIVATE state when there is a race between the transmitter sending the last remaining flits and deasserting **LINKACTIVEREQ** to move to the DEACTIVATE state.

Table 14-5 on page 14-444 summarizes the required behavior described in detail in Table 14-2 on page 14-436.

Table 14-3 Summary of behavior for each Request and Acknowledge state

	Transmitter	Receiver
STOP	Must not send flits. Will not receive credits.	Must not send credits. Will not receive flits.
ACT	Must not send flits. Must accept credits.	Must not send credits. Will not receive flits.
RUN	Can send flits. Must accept credits.	Must accept flits. Can send credits.
DEACT	Must not send flits, except for credit return flits. Must accept credits. Must return credits.	Must accept flits. Must stop sending credits.

Race conditions

There are two situations where one side of the interface performs two actions at or around the same time:

- Changing the **LINKACTIVEREQ** or **LINKACTIVEACK** signal to change the state of the interface.
- Sending an associated credit or flit around the time of the state change.

This occurs in the following situations:

- When the receiver is asserting **LINKACTIVEACK**, to move from ACTIVATE to RUN, it is also permitted to start sending credits:
  - A race can occur between the sending of a credit, which is expected in the new state, and the assertion of the **LINKACTIVEACK** signal indicating the state change.
  - This is acceptable because the transmitter is required to be able to accept the credit in the previous state as well as in the new state.
  - For the receiver, it is permitted to send a credit in the same cycle that **LINKACTIVEACK** is asserted.
  - For the transmitter, it is required to accept a credit both before and after the assertion of **LINKACTIVEACK**.

- When the transmitter is deasserting **LINKACTIVEREQ**, to move from RUN to DEACTIVATE, it must stop sending flits, other than L-Credit return flits:
  - A race can occur between the last flit sent, which is expected in the previous state, and the deassertion of the **LINKACTIVEREQ** signal indicating the state change.
  - This is acceptable because the receiver is required to be able to accept the flit in the next state, as well as in the previous state.
  - For the transmitter, it is permitted to send a flit in the last cycle that **LINKACTIVEREQ** is asserted.
  - For the receiver, it is required to accept flits both before and after the deassertion of **LINKACTIVEREQ**.

Response to new state

When moving to a new state, where the state change has been initiated by the other-side of the interface, a component might be required to change its behavior.

If the state change requires a component to start sending flits or credits, then there is no defined limit on the time taken for the component to start the new behavior. This new behavior will only occur in the new state.

If the state change requires a component to stop sending flits or credits, then the component is permitted to take some time to respond. In this scenario, it is possible to see behavior when first entering a new state which is not expected within that state.

The state change from RUN to DEACTIVATE is the point at which flits and credits stop being sent.

Flits are sent by the transmitter, which is also the component that determines the state change, and therefore the transmitter can ensure flits are not sent after the state change. However, a race condition might still occur as described in *Race conditions* on page 14-437.

Credits are sent by the receiver, but that component does not determine the state change. The receiver might take some time to react to the state change and therefore it is possible for credits to be sent when first entering the DEACTIVATE state.

The protocol requires that the receiver has stopped sending credits and has had all credits returned before it signals the change from DEACTIVATE to STOP.

Determining when to move to ACTIVATE or DEACTIVATE

For a given channel, or set of channels in the same direction, the transmitter is always responsible for initiating the state change from RUN to STOP, or from STOP to RUN.

The transmitter itself can determine that a state change is needed. This can happen through a number of mechanisms. The following examples are not exhaustive:

- The transmitter can determine that it has flits to send, so must move from STOP to RUN.
- The transmitter can determine that it has no activity to perform for a significant period of time, so can move from RUN to STOP.
- The transmitter can observe an independent sideband signal that indicates it should move either from RUN to STOP, or from STOP to RUN.
- The transmitter can determine that a transaction is not fully complete and therefore the channels should remain in RUN state until all activity has completed.
- The transmitter can observe a state change on the channel, or set of channels, that are used in the opposite direction. See *Transmit and receive link Interaction* on page 14-440.

Multiple channels in the same direction

Figure 14-4 shows an example of a multiple channel interface, also referred to as a link, that transfers payload flits in the same direction. A single pair of **LINKACTIVEREQ** and **LINKACTIVEACK** signals are used for all channels.

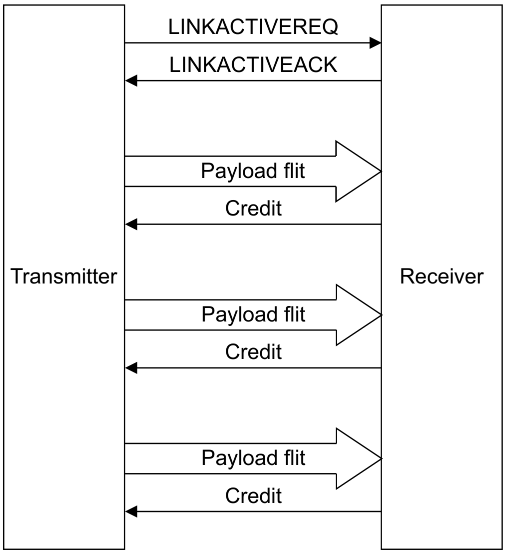


Figure 14-4 Example of a multiple channel unidirectional interface

The rules regarding the relationship between the **LINKACTIVEREQ** and **LINKACTIVEACK** signals must be applied appropriately across all channels:

- When a state change requires the transmitter to be able to accept credits it must be able to accept credits on all channels.
- When a state change requires the receiver to be able to accept flits it must be able to accept flits on all channels.
- When the sending of flits must stop before a state change the sending of flits must stop on all channels.
- When the sending of credits must stop before a state change the sending of credits must stop on all channels.
- A credit can only be associated with a flit on the same channel.

14.6 Transmit and receive link Interaction

This section describes the interaction between a link transmitter and receiver. It contains the following subsections:

- *Introduction*.
- *Tx and Rx state machines* on page 14-441.
- *Expected transitions* on page 14-443.

14.6.1 Introduction

A single component has a number of different channels, some of which are inputs and some of which are outputs.

For a single component:

- All the channels where the Payload is an output are defined to be the *Transmit Link* (TXLINK).
- All the channels where the Payload is an input are defined to be the *Receive Link* (RXLINK).

This specification requires that the activation and deactivation of the TXLINK and RXLINK are coordinated.

When the TXLINK and RXLINK are both in the stable STOP state:

- If the RXLINK moves to the ACTIVATE state, which is controlled by the component on the other side of the interface, then it is required that the TXLINK also moves to the ACTIVATE state, in a timely manner.
- If a component moves the TXLINK to the ACTIVATE state, which it controls, then it can expect the RXLINK to also move to the ACTIVATE state, in a timely manner.

When the TXLINK and RXLINK are both in the stable RUN state:

- If the RXLINK moves to the DEACTIVATE state, which is controlled by the component on the other side of the interface, then it is required that the TXLINK also moves to the DEACTIVATE state, in a timely manner.
- If a component moves the TXLINK to the DEACTIVATE state, which it controls, then it can expect the RXLINK to also move to the DEACTIVATE state, in a timely manner.

When the TXLINK and RXLINK are changing states, the rules about the sending and receiving of credits and flits can be considered independently for each link.

14.6.2 Tx and Rx state machines

Figure 14-5 shows the permitted relationships between the Tx and Rx state machines. It is formatted so that the independent nature of the Tx and Rx state machines can be seen.

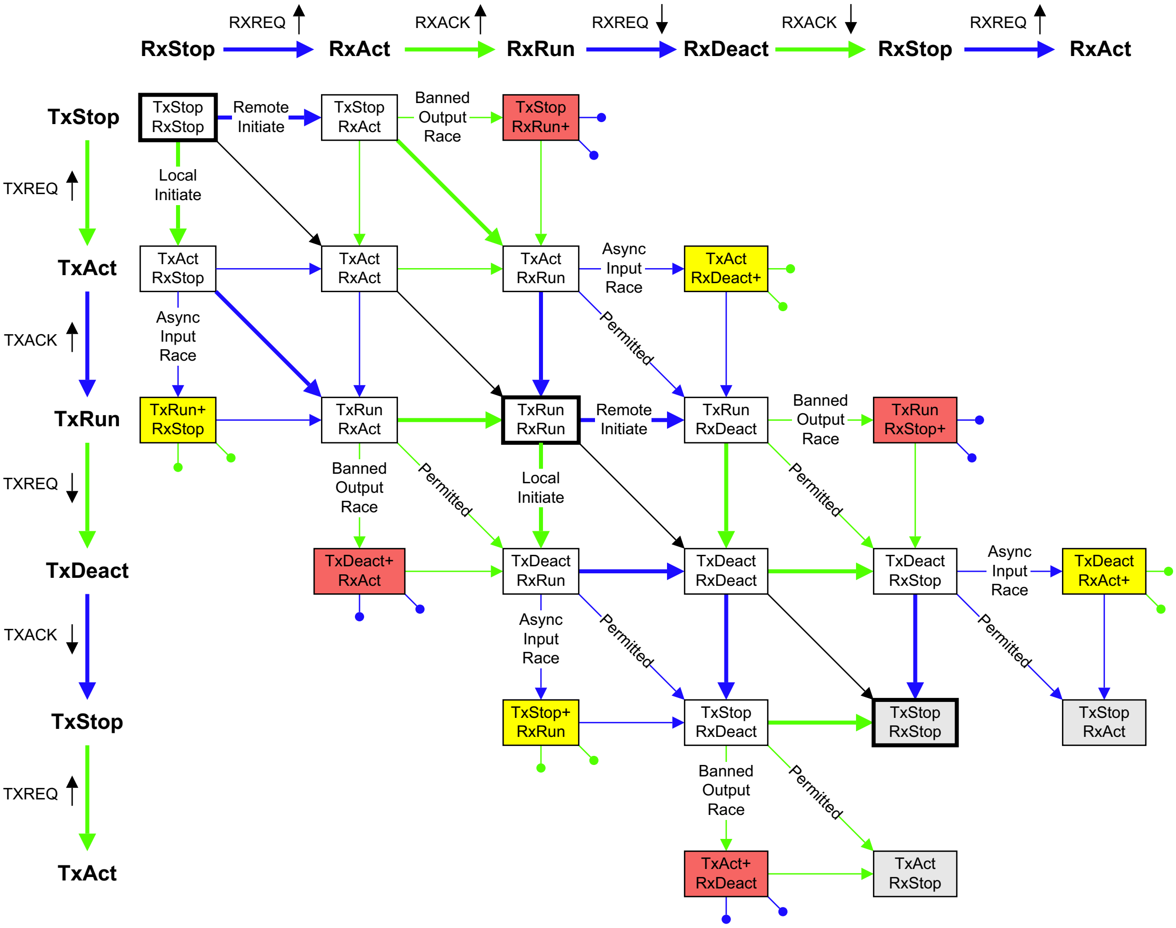


Figure 14-5 Combined Tx and Rx state machines

Figure 14-5 shows the combined Tx and Rx state machines for a single component:

- For clarity, shortened state names and signal names are used.
- A green arrow represents a transition that the local agent can control.
- A blue arrow represents a transition that is under the control of the remote agent on the other side of the interface.
- A black arrow represents a transition that is made when both the local and remote agents make a transition at the same time.
- Around the edge of Figure 14-5 is an indication of the individual Tx and Rx states, the green and blue arrows show which agent controls the transition. There is also an indication of the signal change that causes the state transition.
- A vertical or horizontal arrow is a state change caused by just one signal change, that is, only the Rx state machine or the Tx state machine changes state, not both.
- A diagonal arrow is a state change caused by two signals changing at the same time. If the diagonal arrow is green or blue then the same agent is changing both signals.

- There are a few cases where, by coincidence, a state change occurs due to two events, one on each side of the link, occurring at the same time. This is always a diagonal path and is shown by a black arrow.
- The stub-lines show dead-end paths where an exit from a state is not permitted. The color of a stub-line indicates which agent is responsible for ensuring that the path is not taken.
- The TxStop/RxStop and TxRun/RxRun states are expected to be stable states, and are typically the states where the state machines stay for long periods of time. These states are highlighted with a bold outline. All other states are considered transient states that are exited in a timely manner.
- The grey states, on the bottom right of Figure 14-5 on page 14-441, are replications of those on the top left. They are shown to aid clarity and maintain the symmetry of the diagram.
- The yellow states can only be reached by observing a race between two input signals. The transition into these states is labeled with *Async Input Race*. See *Asynchronous race condition* on page 14-444.
- The red states can only be reached by observing a race between two output signals. A race between two outputs is not permitted at the edge of a component and therefore the transition into these states is labeled with *Banned Output Race*. These states can only be observed at a midpoint between two components. See *Asynchronous race condition* on page 14-444.
- The bold arrows are used to indicate the expected transitions around the state machine. These are described in more detail in *Expected transitions* on page 14-443.
- The arrows labeled *Permitted* are state transactions that would not normally be expected, but they are permitted by the protocol.

State naming

Figure 14-5 on page 14-441 shows the full set of states, including those that can only be reached through race conditions. A more detailed discussion of race conditions can be found in *Asynchronous race condition* on page 14-444.

There are two different TxStop/RxRun states, and two different TxRun/RxStop states. These states differ in how they are reached and how it is permitted to exit from them. To differentiate between these states, a [+] suffix is used to indicate which state machine, that is, Tx or Rx, is running ahead. For example:

- TxStop/RxRun+ indicates that the Tx state machine has remained in the previous Stop state, while the Rx state machine has advanced to the next Run state.
- TxStop+/RxRun indicates that the Tx state machine has advanced to the next Stop state, while the Rx state machine remains in the previous Run state.

14.6.3 Expected transitions

Figure 14-6 shows the expected state transitions.

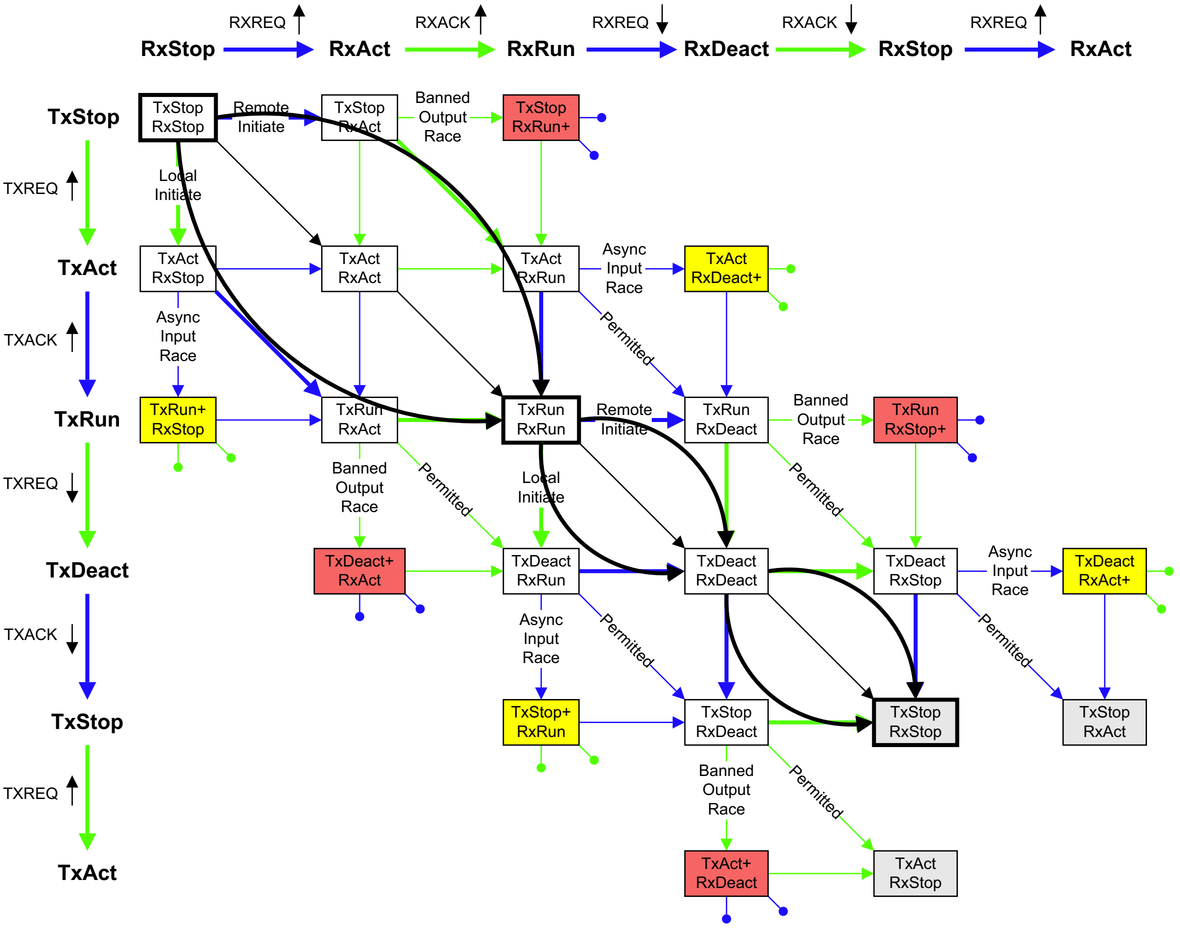


Figure 14-6 Expected Tx and Rx state machines transitions

Figure 14-6 shows, using bold arrows, the routes between the stable TxStop/RxStop and TxRun/RxRun states, and between the stable TxRun/RxRun and the TxStop/RxStop states.

The difference between the two routes moving from TxStop/RxStop to TxRun/RxRun states compared to moving from TxRun/RxRun to TxStop/RxStop states is due to the requirement to return *Link layer Credits* (L-Credits) in the latter case. The differences are detailed in the following sections.

Expected transitions from TxStop/RxStop to TxRun/RxRun

There are two expected routes from a stable Stop/Stop to Run/Run state. Table 14-4 shows, in terms of the state transitions, the two expected paths.

Table 14-4 Stop/Stop to Run/Run state paths

	State 1	State 2	State 3	State 4
Path 1	TxStop/RxStop	TxStop/RxAct	TxAct/RxRun	TxRun/RxRun
Path 2	TxStop/RxStop	TxAct/RxStop	TxRun/RxAct	TxRun/RxRun

The annotations on the diagram arrows in Figure 14-6 on page 14-443 are:

Local Initiate	Indicates that the local agent has initiated the process of leaving one stable state towards the other stable state.
Remote Initiate	Indicates that the remote agent on the other side of the interface has initiated the process of leaving one stable state towards the other stable state.

Expected transitions from TxRun/RxRun to TxStop/RxStop

A transition from a Run/Run state to a Stop/Stop state requires that L-Credits are returned. A link must remain in the DEACTIVATE state until all L-Credits are returned.

There are four expected routes from a stable Run/Run to Stop/Stop state. Table 14-5 shows, in terms of the state transitions, the four expected paths.

Table 14-5 State 5

	State 1	State 2	State 3	State 4	State 5
Path 1	TxRun/RxRun	TxDeact/RxRun	TxDeact/RxDeact	TxStop/RxDeact	TxStop/RxStop
Path 2	TxRun/RxRun	TxDeact/RxRun	TxDeact/RxDeact	TxDeact/RxStop	TxStop/RxStop
Path 3	TxRun/RxRun	TxRun/RxDeact	TxDeact/RxDeact	TxStop/RxDeact	TxStop/RxStop
Path 4	TxRun/RxRun	TxRun/RxDeact	TxDeact/RxDeact	TxDeact/RxStop	TxStop/RxStop

Transitioning around a stable state

It is permitted, but not expected, to transition around a stable TxRun/RxRun or TxStop/RxStop state.

In the majority of cases, moving to the stable Run/Run or Stop/Stop state would be expected.

The most likely use case for wanting to move quickly out of one of the stable states is when an interface has started to enter a low power state, but there is still some activity required. It might be that the low power state was entered prematurely, or it might be that some new activity arose, by coincidence, while the low power state was being entered. In this use case, it is desirable to be able to move back to the Run/Run state as quickly as possible.

Asynchronous race condition

There are situations where two output signals, X and Y, have a defined relationship such that:

- Output X must change after or at the same time as output Y, but it is not permitted to change before output Y.

This relationship applies specifically as follows:

- The assertion of RXACK must not occur before the assertion of TXREQ.
- The deassertion of RXACK must not occur before the deassertion of TXREQ.
- The assertion of TXREQ must not occur before the deassertion of RXACK.
- The deassertion of TXREQ must not occur before the assertion of RXACK.

In Figure 14-5 on page 14-441, these transitions are labeled as *Banned Output Race* and the resultant state is shown in red.

It is possible to observe these states if monitoring the output signals at a point in the system where asynchronous race conditions can result in two signals, that are asserted within the same cycle, are observed in different clock cycles.

A component that is on the other side of the interface, and has the two signals as inputs, can see the state transition if an asynchronous input race occurs. These transitions are labeled on the diagram as *Aysnc Input Race* and the resultant state is shown in yellow.

For all input race conditions, a component that observes the input race is required to wait for both signals before changing any output signals. This is represented in Figure 14-5 on page 14-441 by the fact that the only permitted output transition from a race state is caused by the arrival of the other signal associated with the race condition.

Combined Tx and Rx state machines without race conditions

In Figure 14-7 all transitions and states that occur as a result of a race condition in the combined Tx and Rx state machines have been removed.

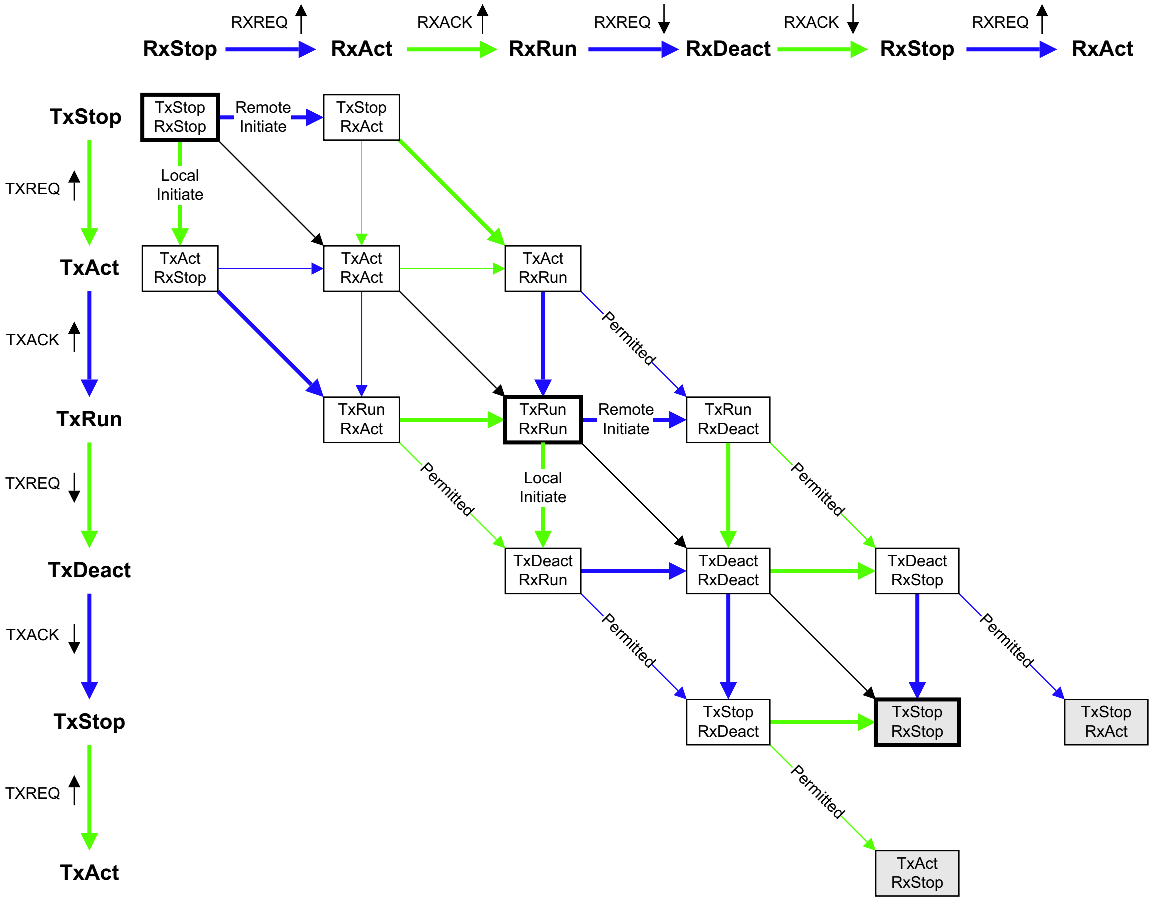


Figure 14-7 Combined Tx and Rx state machines without race conditions

14.7 Protocol layer activity indication

This section describes the signals that indicate Protocol layer activity. It contains the following subsections:

- *Introduction*.
- *TXSACTIVE signal*.
- *RXSACTIVE signal* on page 14-449.
- *Relationship between SACTIVE and LINKACTIVE* on page 14-449.

14.7.1 Introduction

SACTIVE signaling indicates that there are transactions in progress.

TXSACTIVE is an output signal that is asserted by an interface where there is a transaction either in progress or about to start:

- **TXSACTIVE** must be asserted before or in the same cycle in which the first flit relating to a transaction is sent.
- **TXSACTIVE** must remain asserted until after the last flit relating to all transactions is sent or received.

This means that the deassertion of **TXSACTIVE** on an interface implies that the component has completed all transactions in progress and does not need to send or receive any further flits.

A transaction that is given a RetryAck response is considered to be in progress, so **TXSACTIVE** must remain asserted until the associated credit has been supplied and used or returned.

RXSACTIVE is an input signal which indicates that the other side of the interface has ongoing Protocol layer activity. When **RXSACTIVE** is asserted a component must respond to Protocol layer activity in a timely manner.

SACTIVE signals must be synchronous to **CLK** and therefore are not required to be synchronized. If they cross a clock domain, the clock domain crossing bridge is required to synchronize the signals.

14.7.2 TXSACTIVE signal

The following rules apply to the **TXSACTIVE** signal:

- **TXSACTIVE** must be asserted when the transmitter has flits to send.
- A component that asserts **TXSACTIVE** must also, if required, initiate the link activation sequence. It is not permitted for a component to assert the **TXSACTIVE** signal and then wait for the other side of the interface to initiate the link activation sequence.
- **TXSACTIVE** must remain asserted until after the last flit relating to all transactions is sent or received.
- It is permitted for **TXSACTIVE** to be deasserted while transmitting link flits as part of the link deactivation sequence.

Note

To ensure an efficient power-down sequence, ARM recommends not to assert a deasserted TXSACTIVE signal during a link deactivation sequence.

It is permitted for the interface on an interconnect component to use the **RXSACTIVE** input signal to directly generate the **TXSACTIVE** output signal. This behavior is only permitted on the interconnect interface and it is not permitted on any attached component.

Except for an interconnect interface of a link, no other interface of a link is permitted to loop-back the incoming **RXSACTIVE** onto the outgoing **TXSACTIVE**.

Figure 14-8 shows the requirements for **TXSACTIVE** assertion during the life of a transaction.

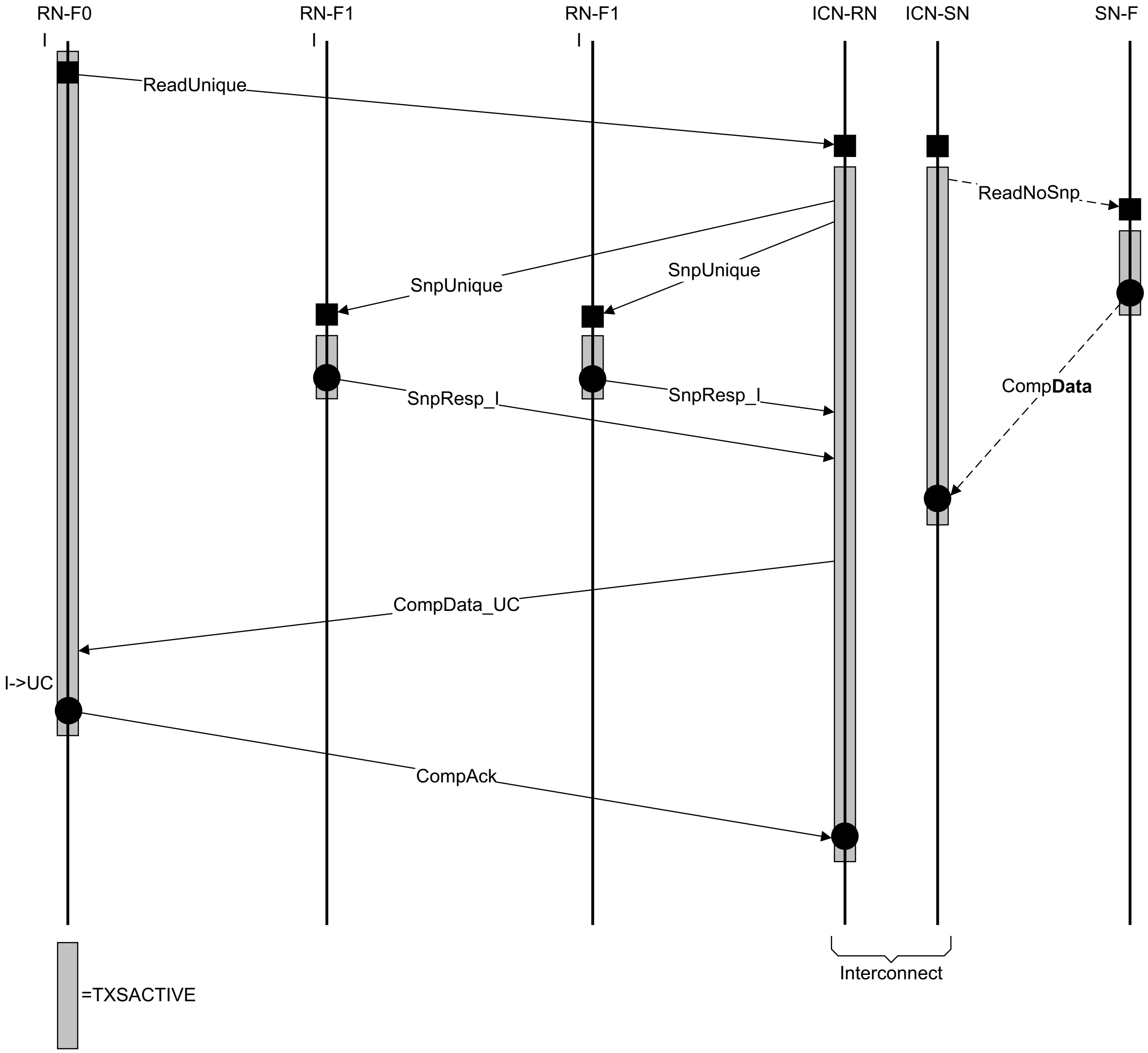


Figure 14-8 TXSACTIVE assertion during the life of a transaction

TXSACTIVE signaling from an RN

When initiating new transactions, an RN must assert **TXSACTIVE** in the same cycle or before **TXREQFLITV** is asserted and must keep it asserted until after the final completing flit of a transaction is sent or received.

The type of flit that completes a transaction initiated by an RN will depend on both the transaction type and the manner in which the transaction progresses. For example, a ReadNoSnp transaction might typically complete with the receipt of the last CompData flit, but could equally complete with a ReadReceipt, if this is later than the last CompData flit.

Table 14-6 shows the flit types that can complete a transaction. The PrefetchTgt transaction does not include an explicit completion message, the transaction is considered completed the cycle after it is sent. In Table 14-6, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

Table 14-6 RN completing flits for RN initiated transactions

Completing flit	Channel	Transaction type
CompAck	TXRSP	Read, Dataless, WriteUnique
Comp	RXRSP	Dataless, WriteEvictOrEvict, WriteUnique, WriteNoSnp, AtomicStore, DVM
Persist	RXRSP	CleanSharedPersistSep, Write*CleanShPerSep
CompPersist	RXRSP	CleanSharedPersistSep, Write*CleanShPerSep
CompCMO	RXRSP	Combined Write
StashDone	RXRSP	StashOnceSep
CompStashDone	RXRSP	StashOnceSep
TagMatch	RXRSP	WriteNoSnp[a], WriteUnique[a], Atomic
CompData	RXDAT	Read, Atomic
RespSepData	RXRSP	Read
DataSepResp	RXDAT	Read
ReadReceipt	RXRSP	ReadNoSnp, ReadOnce
PCrdReturn	TXREQ	All transaction types
NonCopyBackWrData	TXDAT	WriteUnique, WriteNoSnp, Atomic
NCBWrDataCompAck	TXDAT	WriteUnique,WriteNoSnp
CopyBackWrData	TXDAT	CopyBack

a. Separate transaction only, not in a Combined Write.

An RN-F or RN-D component must also assert **TXSACTIVE** while a Snoop transaction is in progress. **TXSACTIVE** must be asserted after receiving an initiating Snoop or SnpDVMOp flit, and no later than when its first Response flit is sent. It must keep **TXSACTIVE** asserted until after the final completing flit is sent for all Snoop transactions.

For an RN-F or RN-D the **TXSACTIVE** output is the logical OR of the requirements for the Request interface and the Snoop interface.

TXSACTIVE signaling from an SN

An SN cannot initiate new transactions and is only required to assert **TXSACTIVE** while it is processing a transaction that is in progress.

It must assert **TXSACTIVE** after receiving a transaction initiating flit and it must be asserted before or in the same cycle in which its first Response flit is sent. It must keep **TXSACTIVE** asserted until after the final completing flit is sent or received.

TXSACTIVE signaling from an ICN interface to an RN

The interconnect interface to an RN must assert **TXSACTIVE** in both the following conditions:

- On receiving a transaction initiating flit, it must be asserted before or in the same cycle in which its first Response flit is sent. It must keep **TXSACTIVE** asserted until after the final completing flit is sent or received.
- Before or in the same cycle in which its initiating Snoop or SnpDVMOp flit is sent. It must keep **TXSACTIVE** asserted until after the final completing flit is sent, which will be either SnpResp or SnpRespData.

TXSACTIVE signaling from an ICN interface to an SN

The interconnect interface to an SN must assert **TXSACTIVE** before, or in the same cycle in which its initiating Request flit is sent. It must keep **TXSACTIVE** asserted until after the final completing flit is sent or received.

14.7.3 RXSACTIVE signal

When **RXSACTIVE** is asserted, the receiver must respond to a link activation request in a timely manner. It is permitted for a receiver to delay responding to a link activation request when **RXSACTIVE** is deasserted.

Note

The deassertion of **RXSACTIVE** does not indicate that all Protocol layer activity has completed. It is possible for a receiver to receive a Protocol flit, which corresponds to a transaction that was in progress while **RXSACTIVE** was asserted, after **RXSACTIVE** is deasserted.

RXSACTIVE can be used in combination with a knowledge of the ongoing transactions, which will be indicated by the components **TXSACTIVE** output, to indicate that no further transactions are required. This can be used to control entry to a low power state.

14.7.4 Relationship between SACTIVE and LINKACTIVE

SACTIVE signaling is an indication of Protocol layer activity. A node can be considered inactive when both **TXSACTIVE** and **RXSACTIVE** are deasserted.

LINKACTIVE state is an indication of the Link layer activity. The Link layer at a node, or interconnect, can be considered inactive when its receiver is in TxStop state and its receiver is in RxStop state.

SACTIVE signaling is orthogonal to the LINKACTIVE states with one constraint as specified in *RXSACTIVE signal*.

A node, or interconnect, should only enable higher level clock gating and low power optimizations when both its Protocol and Link layers are inactive.

Chapter 15
System Coherency Interface

This chapter describes the interface signals that support connecting and disconnecting an RN-F from both the Coherency and DVM domains and an RN-D from the DVM domain. It contains the following sections:

15.1 Overview

The system coherency interface signals are:

SYSCOREQ	Master coherency request.
SYSCOACK	Interconnect coherency acknowledge.

Both **SYSCOREQ** and **SYSCOACK** signals must be synchronous to **CLK** and therefore are not required to be synchronized. If they cross a clock domain, the clock domain crossing bridge is required to synchronize the signals.

Figure 15-1 shows the system coherency interface signals connections.

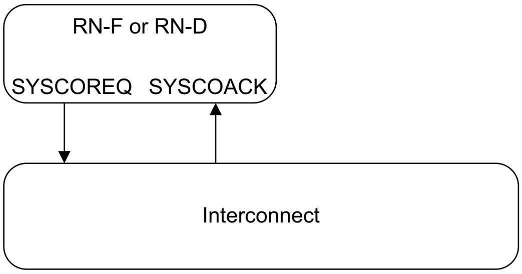


Figure 15-1 System coherency interface signals

Note

In this chapter:

- Coherency when stated, includes the DVM domain, unless explicitly stated otherwise.
- Snoop when stated, includes SnpDVMOp, unless explicitly stated otherwise.

15.2 Handshake

A Request Node, an RN-F or an RN-D, requests connection to system coherency by setting **SYSCOREQ** HIGH. The interconnect indicates that coherency is enabled by setting **SYSCOACK** HIGH.

The Request Node requests disconnection from system coherency by setting **SYSCOREQ** LOW. The interconnect indicates that coherency is disabled by setting **SYSCOACK** LOW.

Requests to enter and exit coherency are always initiated by the Request Node.

Figure 15-2 shows the system coherency interface handshake timing.

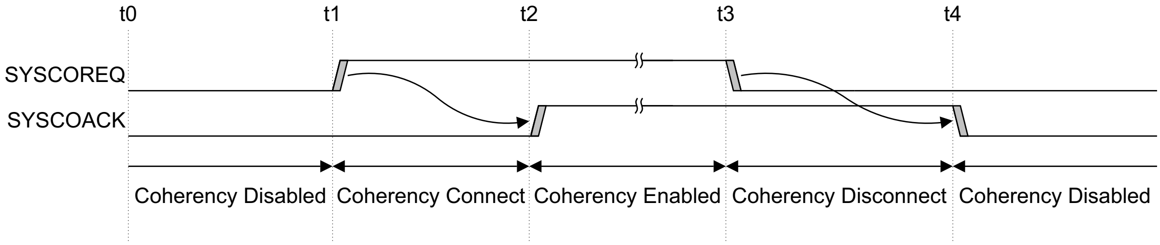


Figure 15-2 System coherency interface handshake timing

As Figure 15-2 shows, the interface signaling obeys four-phase handshake rules:

- **SYSCOREQ** can only change when **SYSCOACK** is at the same logic state.
- **SYSCOACK** can only change when **SYSCOREQ** is at the opposite logic state.

15.2.1 RN rules

Referring to Figure 15-2, an RN must:

- Be able to service Snoop requests when it sets **SYSCOREQ** HIGH at **t1**.
- Not issue a transaction that permits it to cache a coherent location until **SYSCOACK** goes HIGH at **t2**.
- Ensure all transactions that permit it to cache a coherent location are complete before it sets **SYSCOREQ** LOW at **t3**.

  SYSCOREQ can only be deasserted on the cycle after all of the following:
  - All data packets are received for Reads.
  - All data packets are sent for CopyBack.
  - All data packets are sent for snoops and Forwarding snoops.
- Keep servicing Snoop requests until **SYSCOACK** is sampled LOW at **t4**.

SACTIVE must be asserted during coherency connect transition periods to guarantee the **SYSCOACK** transition will occur. See *Protocol layer activity indication* on page 14-446.

Note

The transactions that permit a coherent location to be cached are:

- ReadUnique.
- ReadPreferUnique.
- MakeReadUnique.
- ReadClean.
- ReadNotSharedDirty.
- ReadShared.
- CleanUnique.
- MakeUnique.

15.2.2 Interconnect rules

Referring to Figure 15-2 on page 15-453:

When the interconnect samples **SYSCOREQ** HIGH it must:

- Set **SYSCOACK** HIGH without waiting for responses to any Snoop requests that it has sent after **SYSCOREQ** goes HIGH.
- Be able to service coherent data accesses from the interface when it sets **SYSCOACK** HIGH at **t2**.

When the interconnect samples **SYSCOREQ** LOW it must:

- Complete all Snoop accesses to the interface before it sets **SYSCOACK** LOW at **t4**.

15.2.3 Protocol states

Table 15-1 shows the interface states and the rules that the master must follow in relation to the interface state.

Table 15-1 System coherency interface states

State name	SYSCOREQ	SYSCOACK	Description
Coherency Disabled	0	0	• RN caches must not contain coherent data. • RN must not issue transactions that permit it to cache a coherent location or send DVM transactions. • RN not required to respond to Snoop requests. • Interconnect must not send Snoop requests.
Coherency Connect	1	0	• RN caches must not contain coherent data. • RN must not issue transactions that permit it to cache a coherent location or send DVM transactions. • RN must respond to Snoop requests. • Interconnect can send Snoop requests.
Coherency Enabled	1	1	• RN caches can contain coherent data. • RN can issue transactions that cache a coherent location and send DVM transactions. • RN must respond to Snoop requests.
Coherency Disconnect	0	1	• RN caches must not contain coherent data. • RN must not issue transactions that permit it to cache a coherent location or send DVM transactions. • RN must respond to Snoop requests. • Interconnect must complete outstanding Snoop requests but must not generate new Snoop requests.

Chapter 16
Properties, Parameters, and Broadcast Signals

This chapter describes the properties, parameters, and optional broadcast signals that specify the behavior supported by an interface. It contains the following sections:

16.1 Interface properties and parameters

A property is used to declare a capability. If a property is not declared, it is considered False.

The properties and parameters that specify the interface behavior are:

Atomic_Transactions

An Atomic_Transactions property is used to indicate if a component supports Atomic transactions:

- When not specified, or set to False, Atomic transactions are not supported.
- When set to True, Atomic transactions are supported.

A component that supports Atomic transactions must support all Atomic transactions. However, it is not required that a component that supports Atomic transactions supports the targeting of all memory types.

Cache_Stash_Transactions

A Cache_Stash_Transactions property is used to indicate if a component supports Cache Stashing transactions:

- When not specified, or set to False, Cache Stashing transactions are not supported.
- When set to True, Cache Stashing Transactions are supported.

Direct_Memory_Transfer

A Direct_Memory_Transfer property is used to indicate if a component supports Direct Memory Transfer transactions:

- When not specified, or set to False, Direct Memory Transfer transactions are not supported.
- When set to True, Direct Memory Transfer transactions are supported.
- The Direct_Memory_Transfer property is defined at each HN for each SN.

Direct_Cache_Transfer

A Direct_Cache_Transfer property is used to indicate if a component supports Direct Cache Transfer transactions:

- When not specified, or set to False, Direct Cache Transfer transactions are not supported.
- When set to True, Direct Cache Transfer transactions are supported.
- It is the responsibility of the HN-F to determine the correct snoop type to use.

Data_Poison A Data_Poison property is used to indicate if a component supports Poison:

- When not specified, or set to False, Poison is not supported and the Poison field is not present in the DAT packet.
- When set to True, Poison is supported and the Poison field is present in the DAT packet.

See *Poison* on page 9-337.

Data_Check The Data Check property is used to indicate if Data Check is supported:

- When not specified, or set to False, Data Check is not supported and the DataCheck field is not present in the DAT packet.
- When set to Odd_Parity, Data Check is supported and the DataCheck field is present in the DAT packet.

See *Data Check* on page 9-338.

Check_Type The Check Type property is used to indicate the protection scheme employed on an interface:

- When not specified or set to False, no checking signals are present on the interface.
- When set to Odd_Parity_Byte_Data, Odd parity checking is included for data signals only. Each parity signal covers up to 8 bits.

  Note

  Only the DataCheck field is present.

- When set to Odd_Parity_Byte_All, Odd parity checking is included for all functionally important signals. Each parity signal covers up to 8 bits.

  Note

  A ParityCheck field covering the complete packet and control signals replaces the DataCheck field.

See *Use of interface parity* on page 9-339.

CleanSharedPersistSep_Request

The CleanSharedPersistSep_Request property is used to indicate if a component supports CleanSharedPersistSep:

- When not specified, or set to False, the component does not support CleanSharedPersistSep and the component must not be sent a CleanSharedPersistSep request.
- When set to True, the component supports CleanSharedPersistSep.
- A Home that receives a CleanSharedPersistSep request must support such a request.
- A Home can track if a connected Slave supports CleanSharedPersistSep.
- If a Slave does not support CleanSharedPersistSep:
  - The Home must send a CleanSharedPersist instead of a CleanSharedPersistSep request to that Slave.
  - Home must take the responsibility of sending the Persist response to the Requester. This Persist response must only be sent after receiving the Comp response from the Slave.
- A Requester that does not support CleanSharedPersistSep will generate CleanSharedPersist instead.

MPAM_Support

The MPAM_Support property is used to indicate whether an interface supports MPAM.

- When not specified, or set to False, MPAM is not supported:
  - The interface is not MPAM enabled.
  - No MPAM signals are present on the interface.
- When set to 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 bits and the width of PerfMonGroup is 1bit.

How the MPAM field values are used by a receiver is IMPLEMENTATION DEFINED.

CCF_Wrap_Order

See *Critical chunk first wrap order* on page 2-141.

Req_Addr_Width

This parameter specifies the maximum physical address supported by a component:

- Legal values for this parameter are 44 to 52.
- When Req_Addr_Width is not specified, the default value is 44.

NodeID_Width

This parameter specifies the width of NodeID fields supported by a component, which determines the maximum permitted NodeID value in the system:

- The width specified is uniformly applied to all NodeID related fields.
- Legal values of NodeID_Width are 7 to 11.
- When NodeID_Width is not specified, the default value is 7.

Data_Width This parameter specifies the data width in the DAT channel packet supported by a component:

- Legal values for Data_Width are 128, 256, and 512.
- When Data_Width is not specified, the default value is 128.

Enhanced_Features

The Enhanced_Features property describes the combined support for some of the miscellaneous features in the CHI specification that do not have an explicit property or parameter defined.

When the Enhanced_Features property is True, the component supports all the following enhanced features:

- Data return from SC state.
- I/O Deallocation transactions.
- ReadNotSharedDirty transaction.
- CleanSharedPersist transaction.
- Receiving of Forwarding snoops.

When not specified, or set to False, the component does not support the enhanced features that do not have an explicitly defined property or parameter.

16.1.1 DVM Property

A DVM_Support property is used to indicate Arm ARM specification supported by a given component. When set to:

False	The component does not support any DVM transactions.
DVM_v8	The component supports DVM transactions required to support Armv8.
DVM_v8.1	The component supports DVM transactions required to support Armv8.1.
DVM_v8.4	The component supports DVM transactions required to support Armv8.4.

In a system with heterogeneous components, it is the responsibility of the system configuration to determine the lowest common DVM specification 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 and denial of service, the interconnect must detect unsupported DVM operations. The interconnect must suppress unsupported DVM operation propagation and respond in a protocol-compliant manner. Error indication in such a response is optional.

Note

The following expression can be used by the interconnect to determine if a DVM operation is a new one added in Armv8.4 and not supported by older specifications.

If the following expression is true, then the transaction is a DVM operation supported by Armv8.4 only.

```
[TLBI and (
        (Hypervisor and Secure) or
        (Guest OS and Secure and VMID) or
        (non-Secure only) or
        (Range))] OR
[VICI and (
        (Guest OS and Secure and VMID))]
```

16.2 Optional interface broadcast signals

This specification includes six sets of optional pins to determine broadcasting of certain groups of transactions in the interconnect. These pins are optional at the RN to ICN and ICN to SN interfaces. The six sets of optional broadcast pins are:

- **BROADCASTINNER** and **BROADCASTOUTER**.
- **BROADCASTCACHEMAINTENANCE** and **BROADCASTPERSIST**.
- **BROADCASTATOMIC**.
- **BROADCASTICINVAL**.
- **BROADCASTMTE**.
- **BROADCASTTLBIINNER** and **BROADCASTTLBIOUTER**.

An implementation that includes these signals at the interface must ensure that the signal values are stable when Reset is deasserted.

BROADCASTINNER and **BROADCASTOUTER** determine if inner and outer domain transactions respectively must be broadcast. This specification requires that these two pins must be set to the same value. When set to zero none of the inner and outer transactions are broadcast except for *Cache Maintenance Operations* (CMO).

The **BROADCASTCACHEMAINTENANCE** and **BROADCASTPERSIST** interface signals provide efficient maintenance of downstream caches in the interconnect space. The broadcast signals are used as follows:

BROADCASTCACHEMAINTENANCE

- When asserted:
  - CMO transactions must be broadcast beyond the interface for maintenance of downstream caches.
  - CleanSharedPersist* must be converted to CleanShared before broadcasting to downstream caches if **BROADCASTPERSIST** is deasserted.
- When deasserted:
  - Non-persistent CMO transactions are not required to be broadcast beyond the interface.
  - Broadcasting of Persistent CMO, that is CleanSharedPersist*, beyond the interface is determined by the assertion of the **BROADCASTPERSIST** signal.

BROADCASTPERSIST

- When asserted, CleanSharedPersist* must be broadcast beyond the interface for maintenance of downstream caches. This requirement is independent of the **BROADCASTCACHEMAINTENANCE** signal value.
- When deasserted, broadcasting of the Persistent CMO beyond the interface as a Non-persistent CMO is determined by the **BROADCASTCACHEMAINTENANCE** signal value.

The direction of the signal at the RN to ICN interface is input to RN and at the ICN to SN interface it is input to ICN.

Table 16-1 on page 16-461 shows the broadcast signal encodings using the following keys:

BI	**BROADCASTINNER**.
BO	**BROADCASTOUTER**.
BCM	**BROADCASTCACHEMAINTENANCE**.
BP	**BROADCASTPERSIST**.

Table 16-1 CMO broadcast at the interface with unspecified BI and BO

Broadcast signal			Transaction to be broadcast
BI = BO	BCM	BP	
0	0	0	None.
0	0	1	Persistent CMO only.
0	1	0	Both Non-persistent and Persistent CMO. Persistent CMO is converted to Non-persistent CMO.
0	1	1	Both Non-persistent and Persistent CMO.
1	0	0	All inner and outer transactions. Persistent CMO is converted to Non-persistent CMO
1	0	1	All inner and outer transactions. All Persistent CMO.
1	1	0	All inner and outer transactions including all Non-persistent and Persistent CMO. Persistent CMO is converted to Non-persistent CMO.
1	1	1	All inner and outer transactions including all Non-persistent and Persistent CMO.

BROADCASTATOMIC

- When asserted, the interface is permitted to generate Atomic transactions.
- When deasserted, the interface must not generate Atomic transactions.

An RN is not required to make use of Atomic transactions. An RN that does not make use of Atomic transactions itself, needs no added functionality to be compatible with an interconnect that supports Atomic transactions.

An RN that supports atomic operations but does not include support for the execution of atomic operations must be able to send Atomic transactions.

See *Atomic transaction support* on page 16-463.

BROADCASTICINVAL

- When asserted, DVMOp for ICache Invalidations must be sent to the interconnect.
- When deasserted, DVMOp for ICache invalidations are not required to be sent to the interconnect.

The **BROADCASTICINVAL** signal at each RN is used to inform the RN that broadcasting of Instruction Cache (*ICache*) invalidations using the DVM mechanism is required.

In a system where all Instruction caches are fully coherent the hardware coherency mechanism automatically invalidates all ICache copies on a cache line update, In such systems, it is not necessary to broadcast ICache invalidation operations.

If a system contains one or more Instruction caches that are not updated by the hardware coherency mechanism, then it is necessary for ICache invalidation operations to be broadcast using DVM transactions.

BROADCASTMTE

- When asserted, requests with MTE can be sent beyond the interface.
- When deasserted, requests with MTE must not be sent beyond the interface.

The **BROADCASTMTE** signal is typically deasserted when the interface does not support MTE functionality.

When the **BROADCASTMTE** signal is deasserted, all other MTE related interface pins must be tied to zero. The interconnect is permitted to remove the related MTE transport wires.

The interface fields that can be fixed to a value of zero are:

- On DAT channels:
  - TagOp, Tag, and TU.
- On REQ and RSP channels:
  - TagOp.

BROADCASTTLBIINNER (BTI), BROADCASTTLBIOUTER (BTO)

- **BTI** and **BTO** determine the broadcasting of TLBI operations in the interconnect.
- Table 16-2 shows the permitted **BTI** and **BTO** signal encodings.

Table 16-2 BTI and BTO signal encodings

BTI	BTO	Permitted
0	0	Yes
0	1	Yes
1	0	Reserved
1	1	Yes

Note

The decision to broadcast a DVM(Sync) does not only depend on the values of BTI and BTO but must also consider if any DVM operations other than TLBI need to be pushed to their completion by the DVM(Sync).

16.3 Atomic transaction support

The CHI component support requirements for Atomic transactions are described in the following sections:

- *Request Node support*.
- *Interconnect support* on page 16-464.
- *Slave Node support* on page 16-464.

16.3.1 Request Node support

A Requester component is required to support a mechanism to suppress the generation of Atomic transactions to ensure compatibility in systems where Atomic transactions are not supported. A Requester can use the optional interface pin **BROADCASTATOMIC** to determine whether Atomic transactions are transmitted.

An RN is not required to make use of Atomic transactions. An RN that does not make use of Atomic transactions itself, needs no added functionality to be compatible with an interconnect that supports Atomic transactions.

An RN that supports atomic operations but does not include support for the execution of atomic operations must be able to send Atomic transactions.

For an RN that supports both the execution of atomic operations as well as the sending of Atomic transactions the following applies:

- For cacheable locations, both Snoopable and Non-snoopable, an RN is able to perform an atomic operation locally without generating an Atomic transaction at its interface. To achieve this, the Requester obtains a copy of the location in its local cache, in the same manner that it would for a store operation, and then performs the atomic operation within its local cache. For cacheable locations that are Snoopable, if the contents of the cache line are updated and the cache line was not previously Dirty, then the cache line must be marked as Dirty.

16.3.2 Interconnect support

Interconnect support for Atomic transactions is optional.

The Atomic_Transactions property is used to indicate that an interconnect supports Atomic transactions.

If Atomic transactions are not supported by the interconnect, all attached RNs must be configured to not generate Atomic transactions. The **BROADCASTATOMIC** pin can be used for this purpose, when implemented. See *Request Node support* on page 16-463.

For interconnects that support Atomic transactions, atomic operation execution can be supported at any point within an interconnect, including passing an Atomic transaction downstream to a Slave Node.

Atomic transactions are not required to be supported for every address location.

If Atomic transactions are supported for a given Snoopable address location, then they must be supported for the complete Snoopable address range.

If Atomic transactions are not supported for a given address location, then an appropriate error response can be given for the Atomic transaction. See *Atomic transactions* on page 9-332.

For transactions to a Device the Atomic transaction must be passed to the appropriate endpoint slave. If the slave is configured to indicate that it does not support Atomic transactions, then the interconnect must return an Error response for the transaction.

For Non-snoopable transactions, the Atomic transaction must be performed either:

- At a point, or past a point, where the transaction is visible to all other agents.
- At the endpoint.

For Snoopable transactions, the interconnect can either:

- Perform the atomic operation required by an Atomic transaction within the interconnect.
  - This requires that the interconnect performs the appropriate Read, Write and Snoop transactions to complete the Atomic transaction.
- If the appropriate endpoint slave is configured to indicate that it supports Atomic transactions, then the interconnect can pass the Atomic transaction to the slave.
  - The interconnect is still required to perform the appropriate Snoop and Write transactions before issuing the Atomic transaction to the Slave.

16.3.3 Slave Node support

Slave Node support for Atomic transactions is optional.

The Atomic_Transaction property is used to indicate that an SN supports Atomic transactions.

If an SN supports Atomic transactions for particular memory types, or for particular address regions, then on receiving an Atomic transaction that it does not support, the SN must give an appropriate Error response.

Appendix A
Message Field Mappings

This appendix shows the field mappings for the request, response, data, and snoop request messages. It contains the following sections:

- *Request message field mappings* on page A-467.
- *Response message field mappings* on page A-469.
- *Data message field mappings* on page A-470.
- *Snoop Request message field mappings* on page A-471.

Table A-1 shows the conventions used in the field mapping tables.

Table A-1 Key to field mapping table conventions

Symbol	Description
CF	Common Field. Two or more protocol message fields share the same set of bits in this packet field.
`X`	Inapplicable. Field value can be any value.
`1`	Applicable. Field value is used, must be set to one.
`0`	Applicable. Field value is used, must be set to zero.
`0`[a]	Inapplicable. Field value must be set to zero.
`Y`	Applicable. Field value is used. See specification for permitted values and usage.
D	Inapplicable. Field value must be default setting for MPAM fields.
`8B`	Size field must be set to 8-byte encoding.
`64B`	Size field must be set to 64-byte encoding.
`-`	Assigned to another protocol message field that shares the same set of bits in this packet field.

A.1 Request message field mappings

Table A-2 and Table A-3 on page A-468 show the Request message field mappings. See Table A-1 on page A-466 for the conventions used in the field mappings. For further information on field use see *Protocol flit fields* on page 13-404.

Table A-2 Request message field mappings part 1

Request message	Qos	TgtID	SrcID	TxnID	Opcode	AllowRetry	PCrdType	RSVDC	TagOp	TraceTag	MPAM
ReqLCrdReturn	X	X	X	0	Y	X	X	X	X	X	X
PrefetchTgt	Y	Y	Y	X	Y	0	X	Y	Y	Y	Y
PCrdReturn	Y	Y	Y	0[a]	Y	0[a]	Y	Y	0[a]	Y	0[a]
DVMOp	Y	Y	Y	Y	Y	Y	Y	Y	0[a]	Y	0[a]
ReadNoSnp	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadNoSnpSep	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadShared	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadClean	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadOnce	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadPreferUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MakeReadUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadNotSharedDirty	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CleanShared	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CleanSharedPersist	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CleanSharedPersistSep	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CleanInvalid	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MakeInvalid	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadOnceCleanInvalid	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ReadOnceMakeInvalid	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CleanUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MakeUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Evict	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteNoSnpPtl(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteNoSnpFull(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteNoSnpZero	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteEvictFull	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteEvictOrEvict	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteCleanFull(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteBackPtl	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteBackFull(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteUniquePtlStash	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteUniqueFullStash	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteUniquePtl(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteUniqueFull(CMO)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
WriteUniqueZero	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
StashOnceUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
StashOnceSepUnique	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
StashOnceShared	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
StashOnceSepShared	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
AtomicLoad	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
AtomicStore	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
AtomicCompare	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
AtomicSwap	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table A-3 Request message field mappings part 2

Request message				MemAttr				CF				CF			CF					CF			CF			CF		
	Addr	NS	Size	Allocate	Cacheable	Device	EWA	SnpAttr	DoDWT	Order	LikelyShared	Excl	SnoopMe	ExpCompAck	LPID	TagGroupID[4:0]	StashGroupID[4:0]	PGroupID[4:0]	GroupIDExt	ReturnNID	StashNID	SLCRepHint	StashNIDValid	Endian	Deep	ReturnTxnID	StashLPIDValid	StashLPID
ReqLCrdReturn	X	X	X	X	X	X	X	X	X	X	X	X		X	X				X	X			X			X		
PrefetchTgt	Y	Y	X	X	X	X	X	X	X	X	X	X	-	0[a]	Y	-	-	-	-	0[a]			X			0[a]		
PCrdReturn	0[a]	0[a]	0[a]	0[a]	0[a]	0[a]	0[a]	0[a]	-	0[a]	0[a]	0[a]		0	0[a]				0[a]	0[a]			0[a]			0[a]		
DVMOp	Y	0[a]	8B	0[a]	0[a]	0[a]	0[a]	Y	-	0[a]	0[a]	0[a]		0	Y	-	-	-	Y	0[a]			0[a]			0[a]		
ReadNoSnp	Y	Y	Y	Y	Y	Y	Y	0	-	Y	0	Y	-	Y	Y	-	-	-	Y	Y	-	Y	0[a]			Y	-	-
ReadNoSnpSep	Y	Y	Y	Y	Y	Y	Y	0	-	Y	0	0	-	0	Y	-	-	-	Y	Y	-	Y	0[a]			Y	-	-
ReadShared	Y	Y	64B	Y	1	0	1	1	-	0	Y	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadClean	Y	Y	64B	Y	1	0	1	1	-	0	Y	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadOnce	Y	Y	64B	Y	1	0	1	1	-	Y	0	0	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadUnique	Y	Y	64B	Y	1	0	1	1	-	0	0	0	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadPreferUnique	Y	Y	64B	Y	1	0	1	1	-	0	0	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
MakeReadUnique	Y	Y	64B	Y	1	0	1	1	-	0	0	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadNotSharedDirty	Y	Y	64B	Y	1	0	1	1	-	0	Y	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
CleanShared	Y	Y	64B	Y	Y	Y	Y	Y	-	0[a]	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
CleanSharedPersist	Y	Y	64B	Y	Y	Y	Y	Y	-	0[a]	0	0	-	0	0[a]				0[a]	-	-	Y	-	-	Y	0[a]		
CleanSharedPersistSep	Y	Y	64B	Y	Y	Y	Y	Y	-	0[a]	0	0	-	0	-	-	-	Y	Y	Y	-	Y	-	-	Y	0[a]		
CleanInvalid	Y	Y	64B	Y	Y	Y	Y	Y	-	0[a]	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
MakeInvalid	Y	Y	64B	Y	Y	Y	Y	Y	-	0[a]	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadOnceCleanInvalid	Y	Y	64B	Y	1	0	1	1	-	Y	0	0	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
ReadOnceMakeInvalid	Y	Y	64B	0	1	0	1	1	-	Y	0	0	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
CleanUnique	Y	Y	64B	Y	1	0	1	1	-	0	0	Y	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
MakeUnique	Y	Y	64B	Y	1	0	1	1	-	0	0	0	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
Evict	Y	Y	64B	0	1	0	1	1	-	0	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteNoSnpPtl(CMO)	Y	Y	Y	Y	Y	Y	Y	-	Y	Y	0	Y	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteNoSnpFull(CMO)	Y	Y	64B	Y	Y	Y	Y	-	Y	Y	0	Y	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteNoSnpZero	Y	Y	64B	Y	Y	Y	Y	-	-	Y	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteEvictFull	Y	Y	64B	1	1	0	1	1	-	0	Y	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteEvictOrEvict	Y	Y	64B	1	1	0	1	1	-	0	Y	0	-	1	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteCleanFull(CMO)	Y	Y	64B	Y	1	0	1	1	Y	0	Y	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteBackPtl	Y	Y	64B	Y	1	0	1	1	-	0	0	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteBackFull(CMO)	Y	Y	64B	Y	1	0	1	1	Y	0	Y	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteUniquePtlStash	Y	Y	Y	Y	1	0	1	1	-	Y	Y	0	-	Y	Y	-	-	-	Y	-	Y	Y	Y	-	-	-	Y	Y
WriteUniqueFullStash	Y	Y	64B	Y	1	0	1	1	-	Y	Y	0	-	Y	Y	-	-	-	Y	-	Y	Y	Y	-	-	-	Y	Y
WriteUniquePtl(CMO)	Y	Y	Y	Y	1	0	1	1	Y	Y	Y	0	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteUniqueFull(CMO)	Y	Y	64B	Y	1	0	1	1	Y	Y	Y	0	-	Y	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
WriteUniqueZero	Y	Y	64B	Y	1	0	1	1	-	Y	Y	0	-	0	Y	-	-	-	Y	-	-	Y	0[a]			0[a]		
StashOnceUnique	Y	Y	64B	Y	1	0	1	1	-	0	Y	0	-	0	Y	-	-	-	0[a]	-	Y	-	Y	-	-	-	Y	Y
StashOnceSepUnique	Y	Y	64B	Y	1	0	1	1	-	0	Y	0	-	0	-	-	Y	-	Y	-	Y	-	Y	-	-	-	Y	Y
StashOnceShared	Y	Y	64B	Y	1	0	1	1	-	0	Y	0	-	0	Y	-	-	-	Y	-	Y	-	Y	-	-	-	Y	Y
StashOnceSepShared	Y	Y	64B	Y	1	0	1	1	-	0	Y	0	-	0	-	-	Y	-	Y	-	Y	-	Y	-		-	Y	Y
AtomicLoad	Y	Y	Y	Y	Y	Y	Y	Y	-	Y	0	-	Y	0	Y	Y	-	-	Y	Y	-	Y	-	Y	-	Y	-	-
AtomicStore	Y	Y	Y	Y	Y	Y	Y	Y	-	Y	0	-	Y	0	Y	Y	-	-	Y	Y	-	Y	-	Y	-	0[a]		
AtomicCompare	Y	Y	Y	Y	Y	Y	Y	Y	-	Y	0	-	Y	0	Y	Y	-	-	Y	Y	-	Y	-	Y	-	Y	-	-
AtomicSwap	Y	Y	Y	Y	Y	Y	Y	Y	-	Y	0	-	Y	0	Y	Y	-	-	Y	Y	-	Y	-	Y	-	Y	-	-

A.2 Response message field mappings

Table A-4 shows the Response message field mappings. See Table A-1 on page A-466 for the conventions used in the field mappings. For further information on field use see *Protocol flit fields* on page 13-404.

Table A-4 Response message field mappings

Response message	QoS	TgtID	SrcID	TxnID	Opcode	RespErr	Resp	CBusy	CF				PCrdType	CF		TagOp	TraceTag
									DBID	TagGroupID	StashGroupID	PGroupID		FwdState	DataPull		
RspLCrdReturn	X	X	X	0	Y	X	X	X	X				X	X		X	X
SnpResp	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	0[a]	-	Y	0[a]	Y
SnpRespFwded	Y	Y	Y	Y	Y	Y	Y	Y	X	-	-	-	0[a]	Y	-	0[a]	Y
CompAck	Y	Y	Y	Y	Y	0	0[a]	0[a]	X	-	-	-	0[a]	0[a]		0[a]	Y
RetryAck	Y	Y	Y	Y	Y	0	0[a]	Y	X	-	-	-	Y	0[a]		0[a]	Y
Comp	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	0[a]	0[a]		Y	Y
CompCMO	Y	Y	Y	Y	Y	Y	Y	Y	X	-	-	-	0[a]	0[a]		0[a]	Y
Persist	Y	Y	Y	0[a]	Y	Y	0[a]	Y	-	-	-	Y	0[a]	0[a]		0[a]	X
CompPersist	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	0[a]	0[a]		0[a]	Y
StashDone	Y	Y	Y	0[a]	Y	Y	0[a]	Y	-	-	Y	-	0[a]	0[a]		0[a]	X
CompStashDone	Y	Y	Y	Y	Y	Y	Y	Y	-	-	Y	-	0[a]	0[a]		0[a]	Y
RespSepData	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	0[a]	0[a]		0[a]	Y
CompDBIDResp	Y	Y	Y	Y	Y	Y	0	Y	Y	-	-	-	0[a]	0[a]		0[a]	Y
DBIDResp	Y	Y	Y	Y	Y	0	0[a]	Y	Y	-	-	-	0[a]	0[a]		0[a]	Y
DBIDRespOrd	Y	Y	Y	Y	Y	0	0[a]	Y	Y	-	-	-	0[a]	0[a]		0[a]	Y
TagMatch	Y	Y	Y	0[a]	Y	Y	Y	Y	-	Y	-	-	0[a]	0[a]		0[a]	X
PCrdGrant	Y	Y	Y	0[a]	Y	0	0[a]	Y	0[a]				Y	0[a]		0[a]	Y
ReadReceipt	Y	Y	Y	Y	Y	0	0[a]	Y	X	-	-	-	0[a]	0[a]		0[a]	Y

A.3 Data message field mappings

Table A-5 shows the Data message field mappings. See Table A-1 on page A-466 for the conventions used in the field mappings. For further information on field use see *Protocol flit fields* on page 13-404.

Table A-5 Data message field mappings

																CF								
Data message	**Qos**	**TgtID**	**SrcID**	**TxnID**	**Opcode**	**RespErr**	**Resp**	**CBusy**	**DBID**	**CCID**	**DataID**	**RSVDC**	**BE**	**Data**	**HomeNID**	**FwdState**	**DataPull**	**DataSource**	**TraceTag**	**DataCheck**	**Poison**	**TagOp**	**Tag**	**Tu**
DatLCrdReturn	X	X	X	0	Y	X	X	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X
SnpRespData	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	Y	Y	Y	Y	Y	Y
SnpRespDataFwded	Y	Y	Y	Y	Y	Y	Y	Y	X	Y	Y	Y	Y	Y	0[a]	Y	-	-	Y	Y	Y	Y	Y	Y
CopyBackWrData	Y	Y	Y	Y	Y	Y	Y	0[a]	X	Y	Y	Y	Y	Y	0[a]	0[a]			Y	Y	Y	Y	Y	Y
NonCopyBackWrData	Y	Y	Y	Y	Y	Y	Y	0[a]	X	Y	Y	Y	Y	Y	0[a]	0[a]			Y	Y	Y	Y	Y	Y
NCBWrDataCompAck	Y	Y	Y	Y	Y	Y	Y	0[a]	X	Y	Y	Y	Y	Y	0[a]	0[a]			Y	Y	Y	Y	Y	Y
CompData	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	Y	Y	-	-	Y	Y	Y	Y	Y	Y	Y
DataSepResp	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	Y	Y	-	-	Y	Y	Y	Y	Y	Y	Y
SnpRespDataPtl	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	Y	Y	Y	Y	Y	Y
WriteDataCancel	Y	Y	Y	Y	Y	Y	Y	0[a]	X	Y	Y	Y	0	Y	0[a]	0[a]			Y	Y	Y	Y	Y	Y

A.4 Snoop Request message field mappings

Table A-6 shows the snoop request message field mappings. See Table A-1 on page A-466 for the conventions used in the field mappings. For further information on field use see *Protocol flit fields* on page 13-404.

Table A-6 Snoop Request message field mappings

Snoop Request message	QoS	SrcID	TxnID	Opcode	Addr[(43-51):3]	NS	FwdNID	Common Field				DoNotGoToSD	RetToSrc	TraceTag	MPAM
								FwdTxnID	StashLPIDValid	StashLPID	VMIDExt				
SnpLCrdReturn	X	X	0	Y	X	X	X	X	X	X	X	X	X	X	X
SnpShared	Y	Y	Y	Y	Y	Y	0[a]	0[a]				Y	Y	Y	D
SnpClean	Y	Y	Y	Y	Y	Y	0[a]	0[a]				Y	Y	Y	D
SnpOnce	Y	Y	Y	Y	Y	Y	0[a]	0[a]				Y	Y	Y	D
SnpNotSharedDirty	Y	Y	Y	Y	Y	Y	0[a]	0[a]				Y	Y	Y	D
SnpUnique	Y	Y	Y	Y	Y	Y	0[a]	0[a]				1	Y	Y	D
SnpPreferUnique	Y	Y	Y	Y	Y	Y	0[a]	0[a]				Y	Y	Y	D
SnpCleanShared	Y	Y	Y	Y	Y	Y	0[a]	0[a]				1	0	Y	D
SnpCleanInvalid	Y	Y	Y	Y	Y	Y	0[a]	0[a]				1	0	Y	D
SnpMakeInvalid	Y	Y	Y	Y	Y	Y	0[a]	0[a]				1	0	Y	D
SnpSharedFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	D
SnpCleanFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	D
SnpOnceFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	0	Y	D
SnpNotSharedDirtyFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	D
SnpUniqueFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	1	0	Y	D
SnpPreferUniqueFwd	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	Y	Y	Y	D
SnpUniqueStash	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	-	-	0	Y	Y
SnpMakeInvalidStash	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	-	-	0	Y	Y
SnpStashUnique	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	-	-	0	Y	Y
SnpStashShared	Y	Y	Y	Y	Y	Y	0[a]	-	Y	Y	-	-	0	Y	Y
SnpQuery	Y	Y	Y	Y	Y	Y	0[a]	0[a]				1	0[a]	Y	D
SnpDVMOp	Y	Y	Y	Y	Y	0[a]	0[a]	-	-	-	Y	0[a]	0[a]	Y	0[a]

Appendix B
Communicating Nodes

This appendix specifies, for each packet type, the nodes that communicate using that packet type. It contains the following sections:

- *Request communicating nodes* on page B-474.
- *Snoop communicating nodes* on page B-476.
- *Response communicating nodes* on page B-477.
- *Data communicating nodes* on page B-478.

B.1 Request communicating nodes

Table B-1 shows the Request communicating nodes. In Table B-1, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

For some Requests, both an expected target and a permitted target are given. The use of the permitted target can occur in the case of a software based error. The permitted target must complete the transaction in a protocol compliant manner, this might require the use of an error response.

Table B-1 Request communicating nodes

Request	From	To	
		Expected	Permitted
ReadNoSnp WriteNoSnpFull WriteNoSnpPtl WriteNoSnpZero	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
	ICN(HN-F)	SN-F	-
	ICN(HN-I)	SN-I	-
ReadNoSnpSep	ICN(HN-F)	SN-F	-
	ICN(HN-I)	SN-I	-
ReadClean ReadShared ReadNotSharedDirty ReadUnique ReadPreferUnique MakeReadUnique CleanUnique MakeUnique Evict WriteBackFull WriteBackPtl WriteEvictFull WriteEvictOrEvict WriteCleanFull	RN-F	ICN(HN-F)	ICN(HN-I)
ReadOnce ReadOnceCleanInvalid ReadOnceMakeInvalid StashOnceUnique StashOnceShared StashOnceSepUnique StashOnceSepShared WriteUniqueFull WriteUniqueFullStash WriteUniquePtl WriteUniquePtlStash WriteUniqueZero	RN-F, RN-D, RN-I	ICN(HN-F)	ICN(HN-I)

Table B-1 Request communicating nodes (continued)

Request	From	To	
		Expected	Permitted
CleanShared CleanSharedPersist CleanSharedPersistSep CleanInvalid MakeInvalid	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
	ICN(HN-F)	SN-F	-
	ICN(HN-I)	SN-I	-
DVMOp	RN-F, RN-D	ICN(MN)	-
PCrdReturn	RN-F, RN-D, RN-I	ICN(HN-F, HN-I, MN)	-
	ICN(HN-F)	SN-F	-
	ICN(HN-I)	SN-I	-
AtomicStore AtomicLoad AtomicSwap AtomicCompare	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
	ICN(HN-F)	SN-F	-
	ICN(HN-I)	SN-I	-
PrefetchTgt	RN-F, RN-D, RN-I	SN-F	-

B.2 Snoop communicating nodes

Table B-2 shows the Snoop communicating nodes.

Table B-2 Snoop communicating nodes

Snoop	From	To
SnpShared SnpClean SnpOnce SnpNotSharedDirty SnpUnique SnpPreferUnique SnpCleanShared SnpCleanInvalid SnpMakeInvalid SnpSharedFwd SnpCleanFwd SnpOnceFwd SnpNotSharedDirtyFwd SnpUniqueFwd SnpPreferUniqueFwd SnpUniqueStash SnpMakeInvalidStash SnpStashUnique SnpStashShared SnpQuery	ICN(HN-F)	RN-F
SnpDVMOp	ICN(MN)	RN-F, RN-D

B.3 Response communicating nodes

Table B-3 shows the Response communicating nodes.

Table B-3 Response communicating nodes

Response		From	To
Upstream	RetryAck PCrdGrant Comp CompDBIDResp	ICN(HN-F, HN-I, MN)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F)
		SN-I	ICN(HN-I)
	CompCMO ReadReceipt	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F)
		SN-I	ICN(HN-I)
	RespSepData	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I
	DBIDResp	ICN(HN-F, HN-I, MN)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F), RN-F, RN-D, RN-I
		SN-I	ICN(HN-I), RN-F, RN-D, RN-I
	DBIDRespOrd	ICN(HN-F, HN-I, MN)	RN-F, RN-D, RN-I
	StashDone CompStashDone	ICN(HN-F)	RN-F, RN-D, RN-I
	TagMatch	ICN(HN-F)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F), RN-F, RN-D, RN-I
	Persist	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F), RN-F, RN-D, RN-I
		SN-I	ICN(HN-I), RN-F, RN-D, RN-I
	CompPersist	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I
		SN-F	ICN(HN-F)
		SN-I	ICN(HN-I)
Downstream	CompAck	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)
	SnpResp	RN-F	ICN(HN-F)
		RN-F, RN-D	ICN(MN)
	SnpRespFwded	RN-F	ICN(HN-F)

B.4 Data communicating nodes

Table B-4 shows the Data communicating nodes.

For some Data, both an expected target and a permitted target are given. The use of the permitted target can occur in the case of an incorrect address decode. The permitted target must complete the transaction in a protocol compliant manner. In Table B-4, unless explicitly stated otherwise, a reference to a Write transaction includes both the individual Write transaction and the corresponding Combined Write transaction.

Table B-4 Data communicating nodes

Data		**From**	**To**	
			Expected	**Permitted**
Upstream	CompData	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I	-
		SN-F	RN-F, RN-D, RN-I, ICN(HN-F)	-
		SN-I	RN-F, RN-D, RN-I, ICN(HN-I)	-
	DataSepResp	ICN(HN-F, HN-I)	RN-F, RN-D, RN-I	-
		SN-F	RN-F, RN-D, RN-I	ICN(HN-F)
		SN-I	RN-F, RN-D, RN-I	ICN(HN-I)
Downstream	CopyBackWrData	RN-F	ICN(HN-F)	ICN(HN-I)
	WriteDataCancel	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
		ICN(HN-F)	SN-F	-
		ICN(HN-I)	SN-I	-
	NonCopyBackWrData	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
		RN-F, RN-D	ICN(MN)	-
		ICN(HN-F)	SN-F	-
		ICN(HN-I)	SN-I	-
	NCBWrDataCompAck	RN-F, RN-D, RN-I	ICN(HN-F, HN-I)	-
	SnpRespData SnpRespDataFwded SnpRespDataPtl	RN-F	ICN(HN-F)	-
Peer-to-Peer	CompData	RN-F	RN-F, RN-D, RN-I	-

Appendix C
Revisions

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

Table C-1 Issue B

Change	Location
No changes, first public release	–

Table C-2 Issue C

Change	Location
New feature: Response after receiving first Data packet.	*Snoopable Reads excluding ReadOnce** on page 2-42. *ReadNoSnp, ReadOnce, ReadOnceCleanInvalid, ReadOnceMakeInvalid* on page 2-47.
New feature: Separate Non-data and Data-only response.	*Reads with separate Non-data and Data-only responses* on page 2-53 and multiple related locations.
New feature: Combined CompAck with WriteData.	*WriteUnique transaction* on page 2-109 and multiple related locations.
Clarification: Regarding byte enables for Write transactions.	*Byte Enables* on page 2-137.
Update: Concerning the list of fields that can change value from the original request in the retried request.	*Request Retry* on page 2-147.
Clarification: Concerning the transaction responses permitted to be sent to the same address when a Snoop transaction response is pending.	*At the RN-F node* on page 4-234.

Table C-2 Issue C (continued)

Change	Location
Additional information: Concerning Legal RespErr field values for WriteDataCancel.	Table 9-7 on page 9-330.
Clarification: Regarding TraceTag field value propagation.	*TraceTag usage and rules* on page 11-358.
Corrections and additions: Concerning message field mappings.	Table A-2 on page A-467, Table A-4 on page A-469, and Table A-5 on page A-470.

Table C-3 Issue D

Change	Location
New feature: Persistent CMO with two part response.	*Dataless* on page 2-58. *Dataless transactions* on page 2-103. *Cache Maintenance transactions* on page 4-171. *Details of transaction identifier fields* on page 2-89. *Persistent CMO with snoop and separate Comp and Persist* on page 5-251. *Interface properties and parameters* on page 16-456. *Dataless transactions* on page 9-329.
New feature: Deep Persistent cache maintenance.	*Cache Maintenance transactions* on page 4-171. *Deep* on page 13-414.
New feature: Interface Parity.	*Use of interface parity* on page 9-339. *Interface properties and parameters* on page 16-456.
New feature: *Memory System Performance Resource Partitioning and Monitoring* (MPAM).	*MPAM* on page 11-355. *Interface properties and parameters* on page 16-456.
New feature: Completer Busy indication.	*Completer Busy* on page 11-357.
New feature: ICache Invalidation broadcast signal.	*Optional interface broadcast signals* on page 16-460.
Additional requirement: Concerning **SACTIVE** synchronization to **CLK**.	*Protocol layer activity indication* on page 14-446.
Additional requirement: Concerning **SYSCOREQ** and **SYSCOACK** synchronization to **CLK**.	Chapter 15 *System Coherency Interface*.
Correction: Concerning the use of **RXSACTIVE** to directly generate the **TXSACTIVE** signal.	*TXSACTIVE signal* on page 14-446.
Update: Concerning Ordered Write Observation flow enhancements.	*Streaming Ordered Write transactions* on page 2-124.
Update: Concerning the relaxation of the order requirement between Cache Maintenance transactions and any other transaction to the same address.	*Cache Maintenance transactions* on page 4-171.
Update: Concerning UD_PD state is permitted on a DataSepResp response.	*Read and Atomic transaction completion* on page 4-190.
Update: Concerning DVM early Comp.	*DVM early Comp for Non-sync DVMOps* on page 8-295.

Table C-3 Issue D (continued)

Change	Location
Update: Concerning increased TXID width.	*Flit packet definitions* on page 13-398.
Clarification: Regarding when a RespSepData response includes a Non-data Error.	*Read Transactions* on page 9-327.
Clarification: Regarding when the DataPull bit is set in a SnpRespData message.	*DataPull* on page 13-419.

Table C-4 Issue E.a

Change	Location
New feature: Writes with optional Data: • WriteEvictOrEvict.	*CopyBack* on page 2-68. *CopyBack transactions* on page 4-175. Chapter 9 *Error Handling*. Chapter 12 *Memory Tagging*.
New feature: Write Zero with no Data: • WriteNoSnpZero. • WriteUniqueZero.	*Non-CopyBack* on page 2-64. *Transaction ordering* on page 2-120. *Write transactions* on page 4-173. Chapter 9 *Error Handling*. Chapter 12 *Memory Tagging*.
New feature: SnpQuery Snoop request.	*Snoop request types* on page 4-184. Chapter 9 *Error Handling*.
New feature: DBIDRespOrd response.	*Non-CopyBack* on page 2-64. *Miscellaneous response* on page 4-200.
New feature: New transactions to support Exclusive reads: • ReadPreferUnique. • SnpPreferUnique. • SnpPreferUniqueFwd.	*Snoopable Reads excluding ReadOnce** on page 2-42. *Snoop transactions* on page 2-83. *Read transactions* on page 4-165. *Snoop request types* on page 4-184. *Exclusive transactions* on page 6-278. Chapter 9 *Error Handling*. Chapter 12 *Memory Tagging*.
New feature: Direct Write-data Transfer.	*Direct Write-data Transfer* on page 2-70. *DoDWT* on page 13-417.
New feature: Combined Write transactions: • CompCMO.	*Transaction classification* on page 1-25. *Write and CMO combined request* on page 2-72. *Combined Write requests* on page 4-176.
New feature: Two-part StashOnce transaction including: • StashOnceSep requests. • StashDone response. • CompStashDone response.	*Non-CMO transactions* on page 2-58 *StashOnce or StashOnceSep transaction* on page 2-111. *Dataless transactions* on page 4-170. *Independent Stash request* on page 7-289. Chapter 9 *Error Handling*. Chapter 12 *Memory Tagging*.
New feature: Forward indication on Snoop forward treated as a hint.	*Request types and corresponding Snoop requests* on page 4-187.

Table C-4 Issue E.a (continued)

Change	Location
New feature: Increasing inter-port bandwidth: • Multiple interfaces. • Replicated channels.	*Increasing inter-port bandwidth* on page 13-390.
New feature: Memory tagging.	Chapter 12 *Memory Tagging*.
New feature: Extending DVM operations: • Range based TLBI. • Level hint in TLBI operations, • DVM Domain.	Chapter 8 *DVM Operations*.
New feature: SLC replacement hint.	*SLC replacement hint* on page 11-353.
Additional requirement: Concerning Transaction Ordering guarantees.	*Transaction ordering* on page 2-120.
Additional requirement: Concerning change in WriteNoSnpFull behavior.	*WriteDataCancel in Write transactions* on page 2-76.
Correction: Concerning Size field value in ReadNoSnpSep.	*Read transactions* on page 4-165.
Correction: Concerning the Ordering guarantees provided by the Comp and CompData response.	*Completion Response and Ordering* on page 2-116.
Update: Concerning removal of the DoNotDataPull attribute on snoops.	-
Update: Concerning extending the GroupID field width.	*GroupIDExt* on page 13-406.
Update: Concerning extending the TxnID field width.	*Flit packet definitions* on page 13-398.
Update: Concerning ICache invalidation operations.	*Virtual Instruction Cache Invalidate* on page 8-318.
Update: Concerning Secure EL2 TLBI operations.	*TLB Invalidate* on page 8-310.
Update: Concerning the Order requirements between transactions with different Order field values.[a]	*Transaction ordering* on page 2-120.
Clarification: Regarding Comp and cancelled Write.	*WriteDataCancel in Write transactions* on page 2-76.
Clarification: Regarding CopyBack Write transaction and RetryAck response.	*Request Retry* on page 2-147.
Clarification: Regarding the SnpAttr and Cacheable field value in a standalone CMO transaction.	*Cache Maintenance transactions* on page 4-171.
Clarification: Regarding the attributes of Exclusive accesses.	*Exclusive monitors* on page 6-275.
Clarification: Regarding the receiving of WriteData and the sending of the Persist response.	*Cache Maintenance transactions* on page 4-171.

[a] This Update applies retrospectively to CHI Issue D.

Glossary

This glossary describes some of the technical terms used in AMBA 5 CHI documentation.

Advanced Microcontroller Bus Architecture (AMBA)	The AMBA family of protocol specifications is the ARM open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.
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.
AMBA	*See* Advanced Microcontroller Bus Architecture (AMBA).
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.
Barrier	An operation that forces a defined ordering of other actions.
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.
Byte	An 8-bit data item.
Cache	Any cache, buffer, or other storage structure that can hold a copy of the data value for a particular address location.
Cache hierarchy	The organization of different size caches in a hierarchy, typically with the cache with faster access and smaller size closer to the core and larger and slower access ones farther away from the core. The last level of this hierarchy might be connected to the memory. In this specification, in relation to a referenced cache, above refers to caches closer to the core, and below refers to caches farther from the core.

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 state

State of a block of data in a cache, of 64-byte size in this specification. The state determines if the block is cached in any other caches in the system and also if it is different from the copy of the block in memory. See *Cache state model* on page 1-28 for a description of the cache states supported in this specification.

Channel

A set of signals grouped together to communicate a particular set of messages between a transmitter and receiver pair. For example Request channel is used to communicate request messages.

A channels consist of a set of information signals and a separate Valid and Credit signal to provide the channel handshake mechanism.

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.

See also Cache line.

Completer

See Completer on page 1-23.

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.

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.

Device

See Peripheral slave component.

Direct Data Transfer

Sending Read data directly from a Snoopee or Slave to the Requester bypassing the Home Node.

Don’t Care

See Don’t Care on page 1-24.

Downstream

A 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, Peer to Peer, Slave component, Upstream.

Downstream Cache

See Downstream cache on page 1-23.

Endpoint

See Endpoint on page 1-24.

Final Destination

Final destination for a Memory transaction is a peripheral or physical memory, also called an Endpoint.

Flit

See Flit on page 1-23.

HN

See HN on page 1-24.

ICN

See ICN on page 1-24.

In a timely manner

See In a timely manner on page 1-24.

IMPLEMENTATION DEFINED

Behavior that is not defined by the architecture, but is defined and documented by individual implementations.

When IMPLEMENTATION DEFINED appears in body text, it is always in small capitals.

IMPLEMENTATION SPECIFIC

Behavior that is not architecturally defined, and might not be documented by an individual implementation. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

When IMPLEMENTATION SPECIFIC appears in body text, it is always in small capitals.

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.

IO Coherent node

See IO Coherent node on page 1-24.

Line

See Cache line.

Link

A Link is the connection used for communicating between a transmitter and receiver pair.

Link layer Credit

See Link layer Credit on page 1-24.

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 Speculative read, Store.

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 neither in the RNs nor in the Interconnect.

Main memory can be referred to as memory when the context makes the intended meaning clear.

Master

See Master on page 1-23.

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 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.

Message *See* Message on page 1-23.

Observer A processor or other master component, such as a peripheral device, that can generate reads from or writes to memory.

Outstanding Request

A transaction is outstanding from the cycle that the Request is first issued until either:

- The transaction is fully completed, as determined by the return of all ReadReceipt, CompData, DBIDResp, Comp, CompDBIDResp responses that are expected for the transaction.
- It receives RetryAck and PCrdGrant and is either:
  - Retried using a credit of the appropriate PCrdType, and then is fully completed as determined above.
  - Cancelled, and returns the received credit using the PCrdReturn message.

Peer node A protocol node of the same type with reference to itself. For example, the peer node for a Request Node is another Request Node.

Peer to Peer Communication between the same type of nodes. For example, from one RN to another RN.

See also Downstream, Upstream.

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-compliant manner to prevent system deadlock. The protocol does not require continued correct operation of the peripheral.

See also Memory slave component, Slave component.

Permission to store

A component has permission to store if it can perform a store to the associated cache line without informing any other components or the interconnect.

Packet *See* Packet on page 1-23.

Phit *See* Phit on page 1-23.

PoC *See* PoC on page 1-23.

PoS *See* PoS on page 1-23.

PoP *See* PoP on page 1-23.

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 specification, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Protocol Credit

See Protocol Credit on page 1-24.

Requester

See Requester on page 1-23.

RN

See RN on page 1-24.

Slave

See Slave on page 1-24.

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.

SN

See SN on page 1-24.

Snooped cache

A hardware-coherent cache that receives Snoop transactions.

Snoop filter

A snoop filter is able to track the cache lines that might be allocated within a master.

Speculative read

A transaction that a component 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 speculative read is performed 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 Load.

Stash

The action of placing data in a cache closer to the agent that is expected to be the next user of the data.

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 Load, 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 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.

See also Memory Management Unit (MMU).

TLB

See Translation Lookaside Buffer (TLB).

Transaction

See Transaction on page 1-23.

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.

UNPREDICTABLE In the AMBA Architecture means that the behavior cannot be relied upon.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

When UNPREDICTABLE appears in body text, it is always in small capitals.

Upstream A 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, Peer to Peer, Slave component.

Write-Back cache

A cache in which, when a store is permitted to store, 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 next level cache or 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-Invalidate protocol

See Write-Invalidate protocol on page 1-24.