



Learn the architecture - Introducing AMBA CHI

1.1

Non-Confidential

Copyright © 2021, 2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102407_0101_01_en



Learn the architecture - Introducing AMBA CHI

Copyright © 2021, 2024 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0101-01	15 May 2024	Non-Confidential	Moved the “CHI cache line states” section to “CHI protocol fundamentals”
0100-02	21 June 2024	Non-Confidential	Minor update
0100-01	15 June 2021	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the use 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 Limited ("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 the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, any patents, copyrights, trade secrets, trademarks, or other rights.

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.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted 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 this document shall prevail.

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

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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.

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introducing the AMBA Coherent Hub Interface.....	6
2. Introduction to CHI.....	7
3. CHI protocol fundamentals.....	10
4. Transaction flows.....	18
5. DVM operations.....	39
6. Cache stashing.....	42
7. I/O Deallocation.....	52
8. Data Memory Transfer, Direct Cache Transfer, and PrefetchTgt.....	57
9. Atomic operations.....	69
10. RAS features.....	70
11. Other protocol changes and extensions.....	73
12. Related information.....	80
13. Next steps.....	81

1. Introducing the AMBA Coherent Hub Interface

Coherent Hub Interface (CHI) is an evolution of the AXI Coherency Extensions (ACE) protocol. It is part of the Advanced Microcontroller Bus Architecture (AMBA) that Arm provides. AMBA is a freely available, globally adopted, open standard for the connection and management of functional blocks in a system-on-chip (SoC). It facilitates right-first-time development of multi-processor designs, with large numbers of controllers and peripherals.

CHI is appropriate for a wide range of applications that require coherency including mobile, networking, automotive, and data centers. AMBA CHI has been designed to maintain performance in systems with a growing number of components and traffic.

This guide introduces the first three issues of the CHI protocol, provides a general overview of CHI, and explores several features in-depth.



Note

Diversity and inclusion are important values to Arm. Because of this, we are reevaluating the terminology we use in our documentation. Older Arm documentation, including the AMBA AXI and ACE protocol specification, uses the terms master and slave. This guide uses replacement terminology, as follows:

- The new term Requester is synonymous with master in older documentation
 - The new term Subordinate is synonymous with slave in older documentation
-

2. Introduction to CHI

CHI is designed for scalability, allowing you to build small, medium, or large systems. These systems use multiple components ranging from processor clusters, graphic processors, and memory controllers to I/O bridges, PCI express (PCIe) subsystems, and the interconnect itself.

In this section, we introduce the basic concepts of the first three issues of CHI.

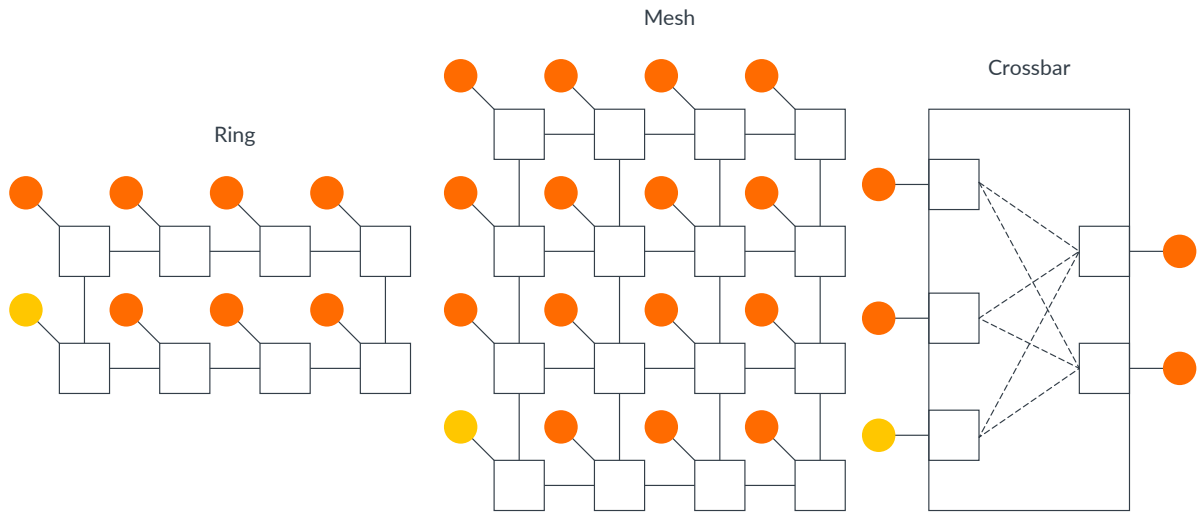
CHI network topologies

CHI defines the different components in a CHI network but does not define which topology is used to connect these components. This topology flexibility allows you to drive component connection by performance, power, and area requirements.

The topologies are:

- A ring topology. In the ring, each component connects directly to two other components, forming a ring where all the components can communicate with each other. The disadvantage of this topology is that latency increases linearly with the number of components in the ring. This is because a transaction must traverse the ring until it reaches its destination. For this reason, the ring topology is best suited for medium-sized systems.
- The mesh topology. Compared to the ring, the mesh contains more paths for a transaction to reach its destination and therefore reduces the travel time of a transaction. This provides higher bandwidth in the system, at the cost of more area. The mesh topology is best suited for large scale systems.
- A crossbar interconnect. This topology allows every node to connect to every possible node. This design offers the best performance, because every component has a direct connection to the component with which it needs to communicate. The drawback of this topology is the cost of connecting all the components. This is because the number of wires needed in the system can grow significantly with each additional component. For this reason, the crossbar topology is best suited for small-sized systems.

The following figure shows different types of topologies that can implement the CHI protocol:

Figure 2-1: Possible topologies for CHI implementation

In this diagram, the circles represent requester and subordinate components in the network. The squares represent intermediary components to route transactions between requester and subordinate.

CHI protocol iterations

There are currently five issues of the CHI protocol: A to E. This guide describes issues A to C and the main differences between these three issues.

The CHI-A issue is the first release of the CHI protocol. It provides a transport layer with features to reduce congestion.

The CHI-A specification describes the fundamental behaviors of CHI. This specification includes:

- Definitions for new channels, CHI terminology, and component naming
- Examples of requests, snoop filters, and cache state transitions
- Rules for transaction ordering, exclusive accesses, and Distributed Virtual Memory (DVM) operations

The CHI-B issue extends CHI-A but is not directly backwards compatible with CHI-A. It adds features to support Armv8.1 and Armv8.2 system extensions such as:

- Larger physical address width
- Atomic Transactions
- VMID extension for DVMs
- Descriptions of channel fields, transaction structure, and Reliability, Availability, and Serviceability (RAS) features
- The Direct Memory Transfer and Direct Cache Transfer features, which decrease memory and Snoop access latencies

The CHI-C issue is a minor extension of CHI-B. This extension primarily adds features that reduce request hazard lifetimes. CHI-C also adds two opcodes that decrease the time of completion acknowledge, with protocol changes to support these opcodes.

3. CHI protocol fundamentals

The CHI protocol classifies different components in a system by node type and provides a means for communication between nodes. This section of the guide reviews the node types and their communication means, then looks more closely at the protocol messages, which are sent as Flits.

Nodes

The three types of main node types are Request Nodes (RNs), Home Nodes (HNs), and subordinate Nodes (SNs). Additionally, there is the Miscellaneous Node (MN), which will also be covered in this section.

RNs generate transactions, like read and write requests, and these transactions are sent to a HNs. HNs are responsible for ordering Requests, generating transactions to SNs, and can issue snoops or handle DVM operations.

Each of these node types can be further categorized as the following:

- RNs can be Fully Coherent, I/O Coherent, or I/O Coherent with DVM support:
 - Fully Coherent Request Nodes (RN-Fs), contain coherent caches and will accept and respond to snoops
 - I/O Coherent Request Nodes (RN-Is), do not have coherent caches, and cannot accept snoops
 - I/O Coherent Request Node with DVM support (RN-Ds), have the same functionality as RN-Is and can also accept DVM messages
- Home Nodes can be Fully Coherent, Non-coherent, or Miscellaneous:
 - Fully Coherent Home Nodes (HN-Fs) order all requests to coherent memory and issue snoops to RN-Fs
 - Non-coherent Home Nodes (HN-Is) order requests that target the I/O subsystem
 - Miscellaneous Nodes (MNs) handle DVM transactions sent by request nodes. These are sometimes implemented as HN-D nodes
- Subordinate Nodes (SN-Fs) for normal memory or for peripheral and normal memory:
 - SN-Fs connect to memory devices that back the coherent memory space. For example, a memory controller would connect to an SN-F node.
 - SN-Fs for peripheral or normal memory connect to I/O peripherals or non-coherent memory

The following table summarizes the behavior of each node category:

-	RN-F, HN-F, SN-F	RN-I, HN-I, SN-I	I/O Coherent DVM support	MN
RN	Coherent caches	No coherent caches	Accepts DVM messages	-
RN	Accepts and responds to snoops	Does not accept or respond to snoops	Same as RN-I in all other respects	-

-	RN-F, HN-F, SN-F	RN-I, HN-I, SN-I	I/O Coherent DVM support	MN
HN	Orders requests to coherent memory	Orders requests that target the I/O subsystem	-	Handles DVM transactions sent by RNs
HN	Issues snoops to RN-Fs	-	-	-
SN	Connects to memory devices that back the coherent memory space	Connects to I/O peripherals or non-coherent memory	-	-

Some components in the system can also be classified as a requester or completer, as described in the following list:

- A requester is a component that starts a transaction by issuing a request message. The term requester can be used for a component that independently initiates transactions. 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.
- A completer is a component that responds to a transaction it receives from another component. A completer can either be an interconnect component, such as a HN or an MN, or a component, such as a subordinate, that is outside of the interconnect.

CHI cache line states

CHI uses a similar coherency model to ACE, adding support for snoop filters and directory-based systems for snoop scaling.

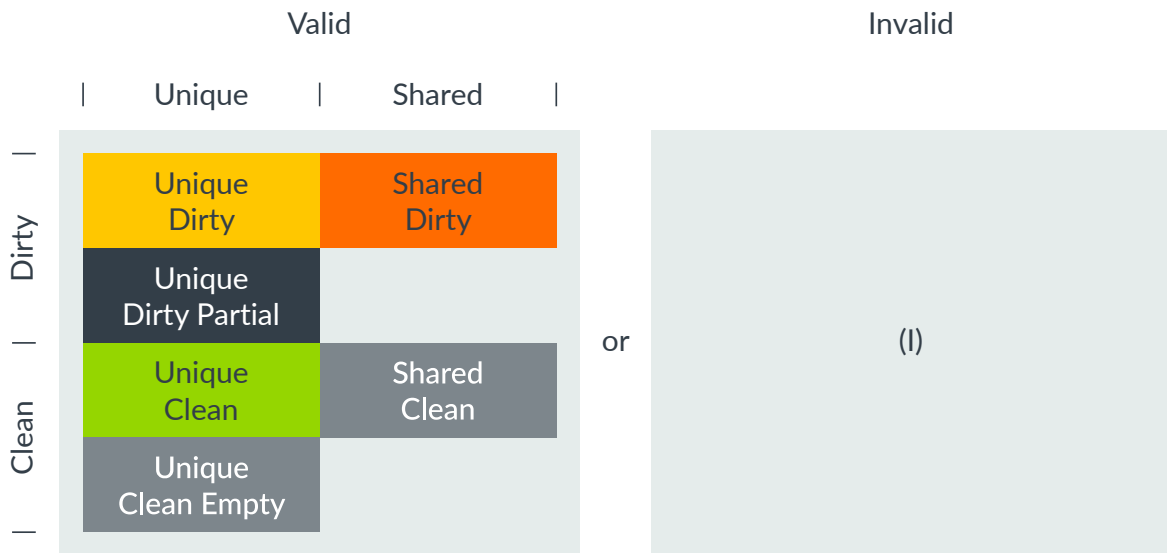
CHI also uses the same terms as ACE to define cache states and adds partial and empty cache line states. The cache line states terms are:

- Valid and Invalid to describe whether a cache line is present in a local cache or not.
- If a cache line is Valid, it must be either Unique or Shared:
 - Unique means that the cache line is present in this cache only and is not present in any other requester local cache. A store to a local cache line can only happen when it is held in a Unique state.
 - Shared means that the cache line is present in this cache and may or may not be present in other requester local caches.
- If a cache line is Valid, it must be either Clean or Dirty:
 - Clean means that the cache is not responsible for updating Main Memory. The cache line can still hold a different value to Main Memory as a result of a previous update in another cache.
 - Dirty means that the cache line has been modified with respect to Main Memory. When this line is evicted from this cache, the requester must ensure either that Main Memory is updated, or that the Dirty responsibility is passed to another component in the system.
- A line can be in a Partial and Empty state:
 - An Empty cache line has no Valid bytes of data, but the ownership of the line still belongs to a requester.
 - A Partial cache line can have some bytes valid, which includes none or all bytes. This is because the state is updated but valid bytes have not been written yet, or because all bytes

have been written but the state has not been updated. There are additional restrictions to the responses that can be given when a line in this state is snooped.

These terms are then combined to describe seven cache line states as shown in the following figure:

Figure 3-1: A diagram showing cache line states



The figure contains the following cache line states:

Invalid

The cache line is not present in the cache.

Unique Dirty

This cache line only exists in this cache only and is modified with respect to Main Memory. In this state the requester can perform a write to the cache line, because the line is already in a Unique state. If a snoop instructs it, the cache line must be forwarded to the Requester.

Unique Dirty Partial

This cache line exists in this cache only and is considered modified with respect to Main Memory. It can have some bytes valid, where some includes none or all bytes. In this state, the requester can perform a write to the cache line because the line is already in a Unique state. In response to a snoop, the cache line cannot be forwarded directly to the original requester, even when instructed to by the snoop.

Shared Dirty

This cache line has been modified with respect to main memory, and this particular cache has the responsibility to update main memory. Because the cache line is Shared, it might exist in one or more local caches, but this is not guaranteed. If the line is present in multiple caches, these caches will have this line in Shared Clean.

Unique Clean

The cache line has not been modified with respect to Main Memory, and only exists in a single local cache. It can be modified without notifying other caches.

Unique Clean Empty

The cache line is present only in this cache, but none of the bytes are valid. The cache line can be modified without notifying other caches. If a snoop requests the line, the line must not be returned to a Home Node or forwarded directly to the original requester.

Shared Clean

The cache line might be held in one or more local caches. The line might have been modified with respect to Main Memory, but this cache is not responsible for writing the line back to memory on eviction.

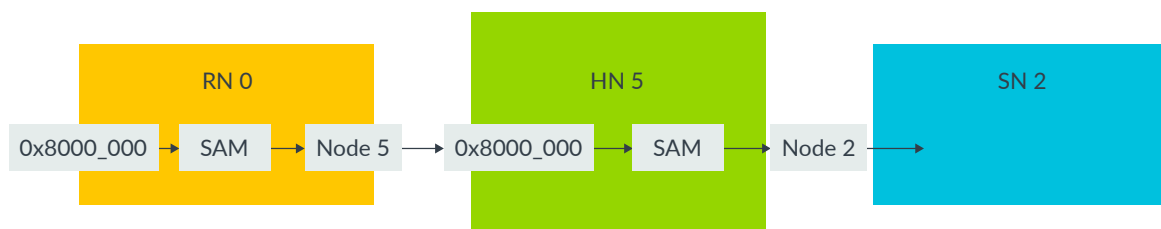
System Address Map

Every component in the system is assigned a Unique Node ID. CHI uses the System Address Map (SAM) to convert physical addresses to a target Node ID.

To be able to determine the target Node ID of outgoing requests, each RN and HN must have a SAM.

The following figure demonstrates the RN SAM mapping physical addresses to HN Node IDs, and the HN SAM mapping physical addresses to SN Node IDs:

Figure 3-2: The RN SAM maps physical addresses to HN Node IDs, and the HN SAM maps physical addresses to SN Node IDs



In this figure, the order of events is as follows:

1. The transaction with address 0x8000_0000 passes through the RN SAM in Node 0.
2. The RN SAM determines the destination as Node 5.

3. The transaction is routed to the HN with Node 5.
4. The HN receives the transaction.
5. The HN passes the address through its HN SAM and determines the destination as Node 2.
6. The transaction gets routed to the SN with Node 2.

The RN SAM must meet the following requirements:

- It must fully describe the entire system address space
- Any physical addresses that do not correspond to a physical component must map to a node that can provide an appropriate error response
- All RNs must have a consistent view of the RN SAM. For example, address `0xFF00_0000` must always go to the same HN, regardless of which RN issued it.



Note

The exact format and structure of a SAM is entirely implementation defined. The CHI specification does not provide guidance on how to map addresses to Node IDs.

Node channels

CHI uses different channels to ACE for:

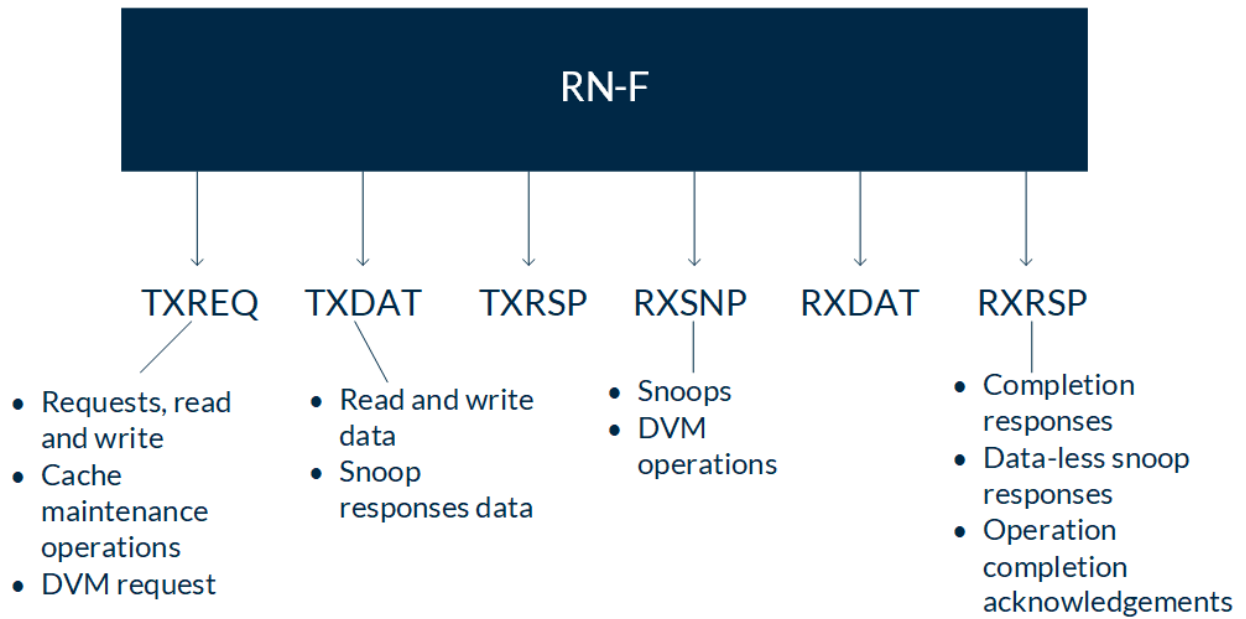
- Requests and Labeled REQ, and sends Read and Write requests, Cache Maintenance requests, and DVM Requests
- Responses and Labeled RSP, and sends completion responses for various types of messages, ranging from write and cache management responses to data less snoop responses and operation completion acknowledgments.
- Snoops and Labeled SNP, and issues snoops or sends DVM operations Data transfer messages and Labeled DAT, and sends write and read data, and snoop responses with data



Note

Channels prefixed with the letters TX are used to transmit messages, and channels prefixed with the letters RX are used to receive messages.

The following diagram shows the channels present on the CHI Requester Interface of an RN-F:

Figure 3-3: RN-F and the CHI Master Interface

When the RN-F issues a read request, it sends the request out on its `TXREQ` channel. When the read data returns, the RN-F receives the data on its `RXDAT` channel. The TX signals on each node connect to the RX signals on the Target nodes. The following constraints occur on the SNP channel:

- Only the HN-F and MN issue messages on the SNP channel
- The RN-F accepts only snoops on the SNP channel
- The MN accepts only DVM message snoops on the SNP channel

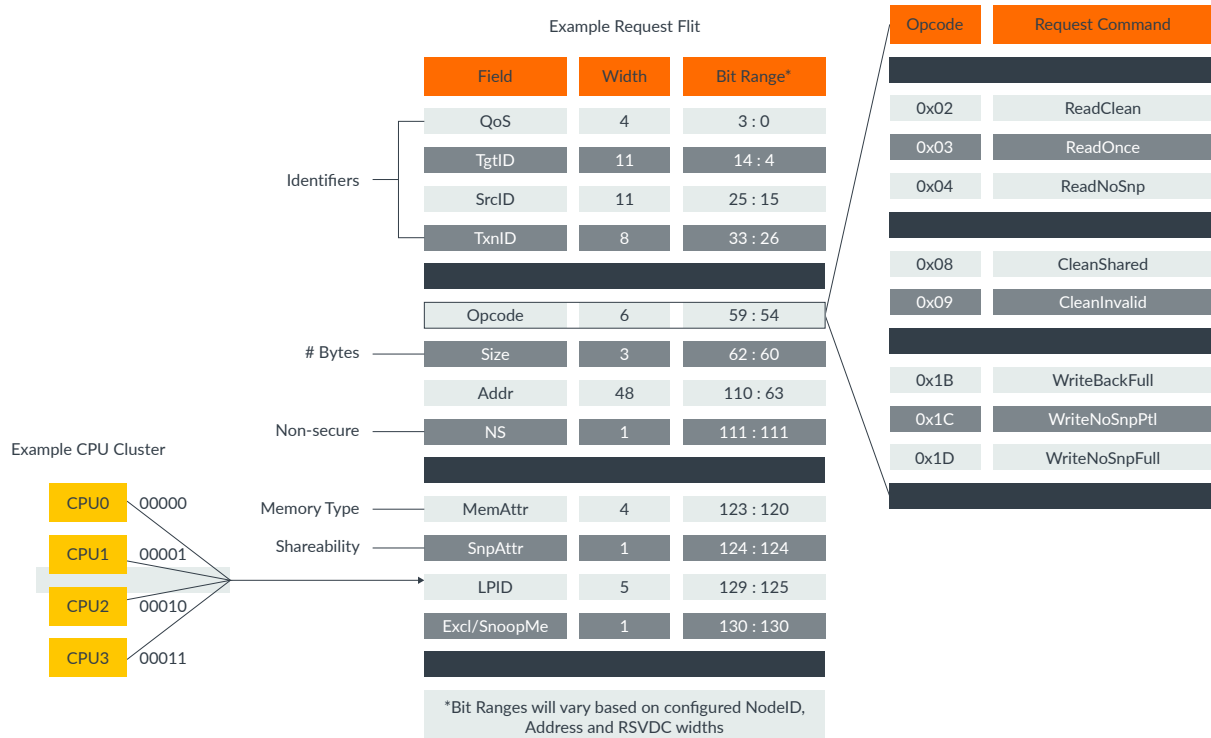
Flits

All protocol messages are sent in the form of a Flit. Flits are a packetized collection of control fields and identifiers that communicate a protocol message.

Some of the control fields sent in a Flit include opcodes, memory attributes, address, data, and error responses. Each channel needs different Flit control fields. For example, a Flit to read or write on the Request channel needs an Address field, and a Flit on the Data channel needs the Data and Byte Enable fields.

The fields in a Flit are not serialized over multiple packets like fields in the PCIe or Ethernet protocols. Instead, they are sent in parallel.

The following diagram shows a Request Flit, and the details of the Flit Opcode:

Figure 3-4: Rectangles containing, identifiers, bytes and a CPU cluster example

The handshake mechanism for communicating a Flit in CHI is different to that in ACE. Each channel is associated with a FLITV signal, and this signal is set to high by the transmitter to indicate that the Flit is valid. The transfer then takes place on the next rising CLK edge. Flits can only be sent by a transmitter when it has previously received a credit from the receiver. A credit signaled by LCRDV asserted and qualified with a rising CLK edge.

To provide additional information in a Flit, CHI defines multiple identifier fields. For example:

- The Source ID field (`srcID`) is used to route Flits across a CHI network. The field identifies the sender of the Flit, and every Flit has the `srcID` field. The value used for the Source ID is the Node ID of the component sending a message.
- The Target ID field (`tgtID`), is also used to route Flits in the network. The Target ID value is the Node ID of the node receiving the message. Every Flit, except the snoop Flit, contains the Target ID field. The reason that the snoop channel does not contain a Target ID is that CHI uses the implementation to identify which nodes receive a snoop. An HN-F can use any mechanism to route snoops, for example to broadcast snoops to all RN-Fs, or using a Snoop filter to target only a subset of RN-Fs. Whatever mechanism is used, when a snoop Flit leaves the interconnect, it is already targeting a specific node.
- The Transaction ID field (`txnID`), is present in every Flit. This field is an 8-bit field that identifies every transaction between a Source Node and a Target Node. Each outstanding request from an RN must have a unique `txnID`. An RN can have up to 256 outstanding transactions at any point in time.

- The Request opcode (`opcode`), is present in the REQ flit. This specifies the transaction type and is the primary field that determines the transaction structure. For example, different types of read request, write request or dataless request.
- The Data Buffer ID (`DBID`), is present only in the Response and Data Flits. The Target Node uses this identifier to signal availability to receive Write Data, and to deallocate transactions that require Completion Acknowledgement.
 - For Writes, a requester cannot send Write Data until it receives a DBID value in the Response from the completer
 - Some read transactions complete with a Completion Acknowledgment, which is where a requester indicates that it has received the read data. For more information, see [Transaction flows](#). When read data is sent back to the requester, the data flit includes a DBID value for the requester to use when sending the Completion Acknowledge message.

The following table summarizes which Flit type can use each Identifier:

Identifier	Request Flit (REQ)	Response Flit (RSP)	Data Flit (DAT)	Snoop Flit (SNP)
SrcID	Yes	Yes	Yes	Yes
TgtID	Yes	Yes	Yes	-
TxnID	Yes	Yes	Yes	Yes
DBID	-	Yes	Yes	-

For an example of how these identifier fields change for an entire transaction, see [Transaction flows](#).

4. Transaction flows

A transaction is the set of messages that the system needs to complete a Request by a node. This section includes examples for:

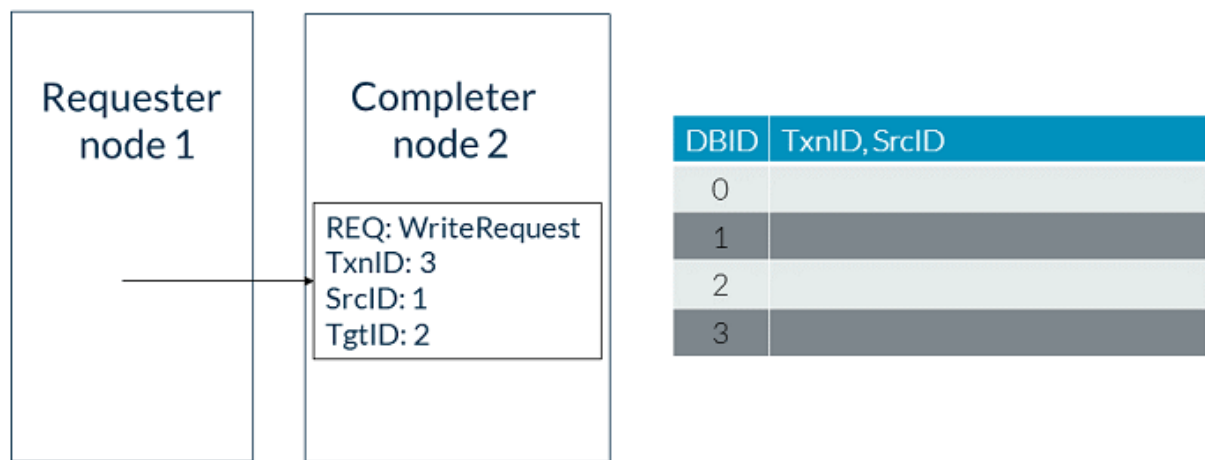
- Using Identifiers in a Write Request between a requester and a completer
- The sequence of messages needed to complete a ReadNoSnp Transaction
- The flow of a WriteNoSnp Transaction from a Request Node through to a completer Node

Write Requests and identifiers

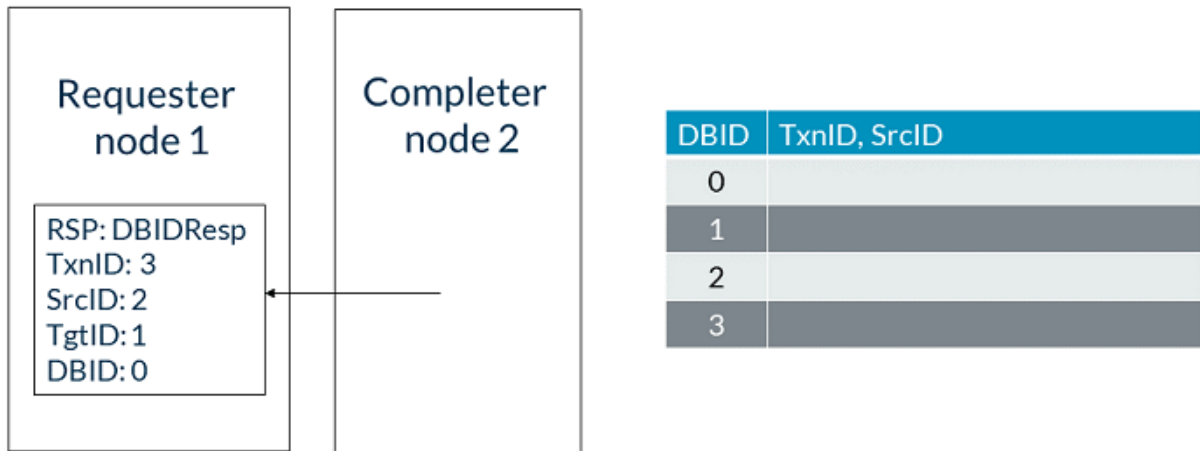
In the following example, we show the use of Identifiers in a Write Request between a requester and a completer. The requester is assigned Node ID 1, and the completer is assigned Node ID 2. The following describes the sequence of events:

1. The requester sends a Write Request to the completer with transaction ID (T_{xnID}) 3. The Source ID (S_{rcID}) field in the requester is populated with the Node ID of the requester. The Target ID (T_{gtID}) field is populated with the Node ID of the completer. This step is shown in the following diagram:

Figure 4-1: A diagram showing a requester sending a write request



2. The completer assigns the transaction ID and Source ID of the Request to an available Data Buffer slot. In this example, the Request is assigned the Data Buffer ID (DBID) 0.
3. The completer sends a $DBID_{resp}$ message back to the requester with T_{xnID} 3 and the DBID value 0, as shown in the following diagram:

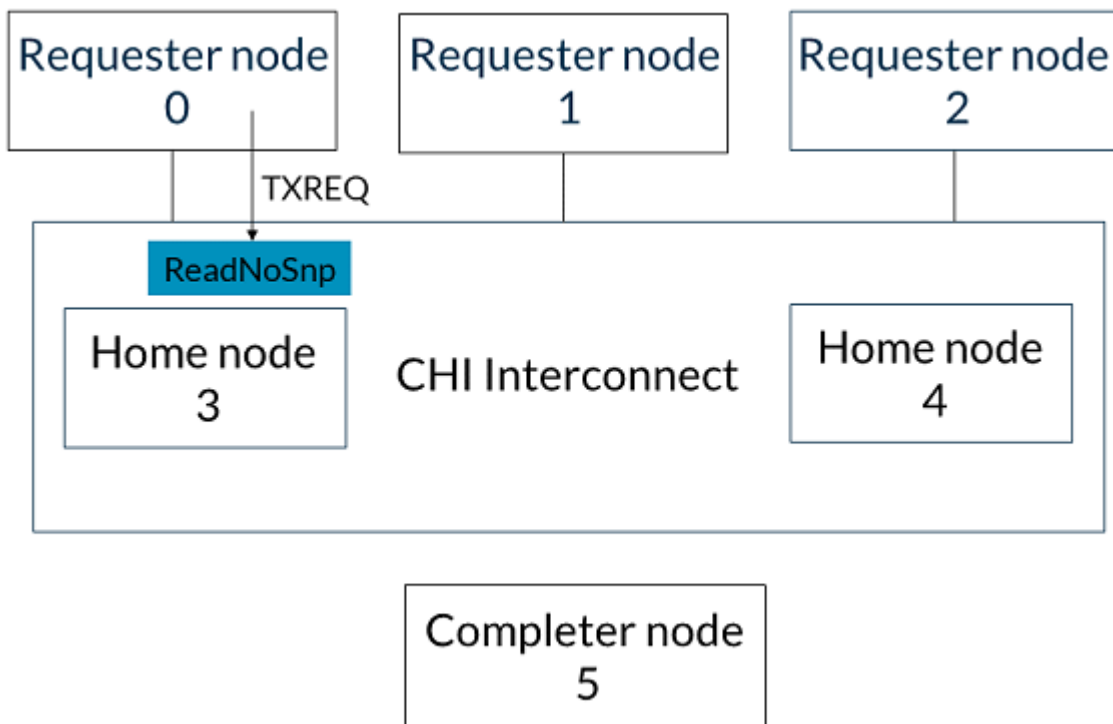
Figure 4-2: A diagram showing a completer sending a DBIDResp message

4. The Requester sends the write data to the Completer using the DBID received as the Transaction ID.
5. The buffer slot corresponding to DBID 0 gets deallocated when the transaction completes.

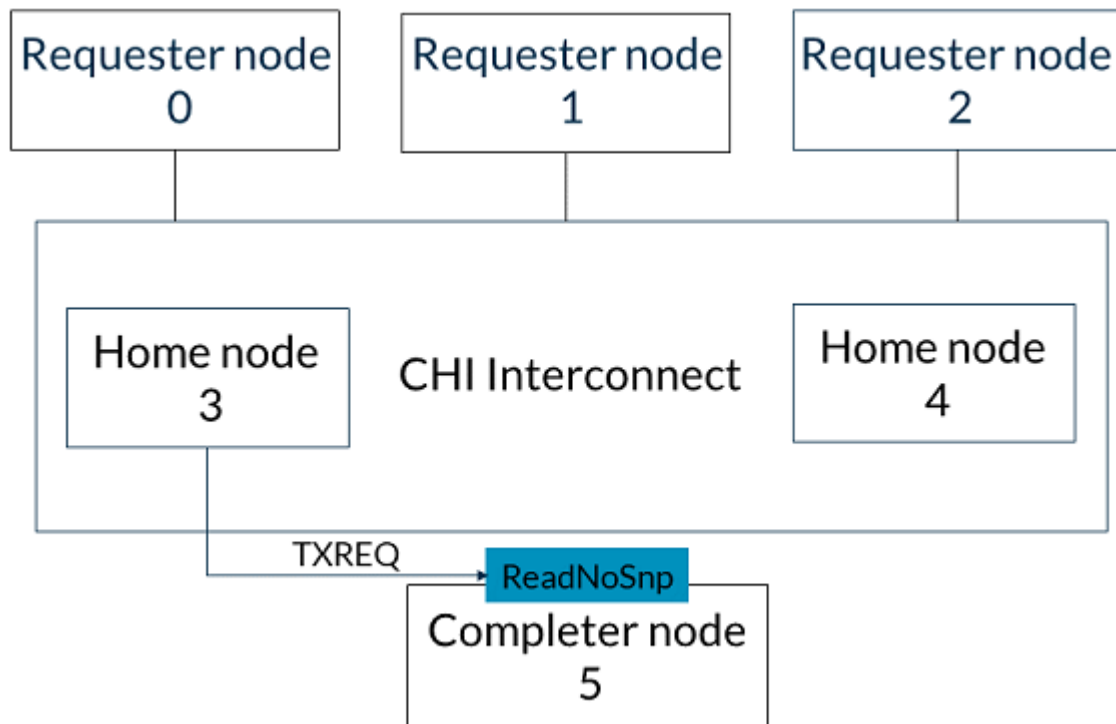
ReadNoSnp Transaction flow

The following example shows the sequence of messages needed to complete a `ReadNoSnp` transaction. Requester node 0 issues the `ReadNoSnp` request and completer node 5 provides the read data. The following describes the sequence of events:

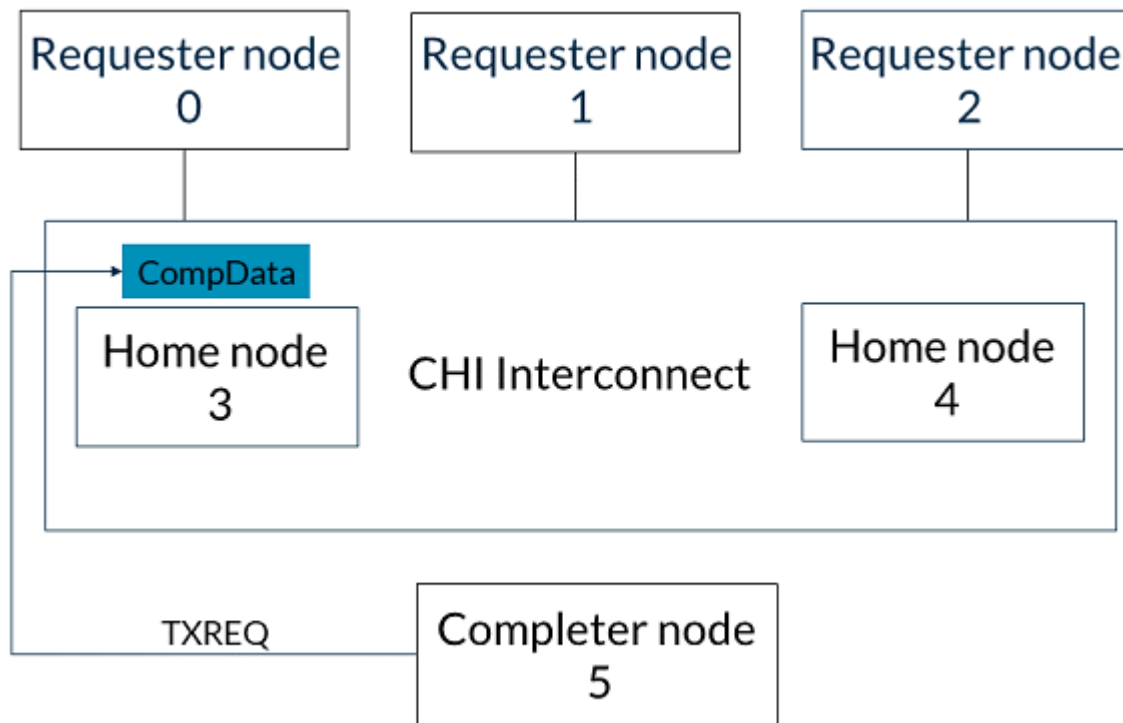
1. Requester node 0 issues a `ReadNoSnp` to the CHI Interconnect, targeting Home node 3. The transaction is sent on the TXREQ channel of the requester node. This step is shown in the following diagram:

Figure 4-3: A requester issuing a ReadNoSnp

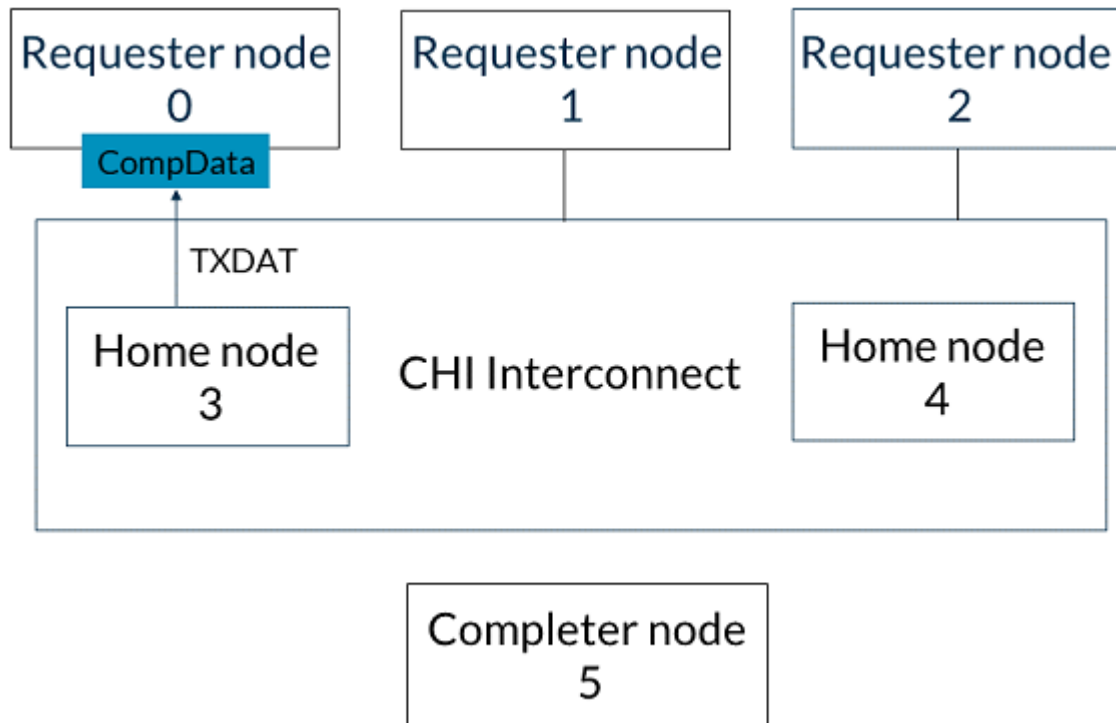
2. Home node 3 issues a `ReadNoSnp` to completer node 5 on its TXREQ channel to retrieve the data, as shown in the following diagram:

Figure 4-4: A home node issuing a ReadNoSnpn

- Completer node 5 returns data to Home node 3 by issuing a `compData` response on its TXDAT channel, as shown in the diagram:

Figure 4-5: A completer node returning data

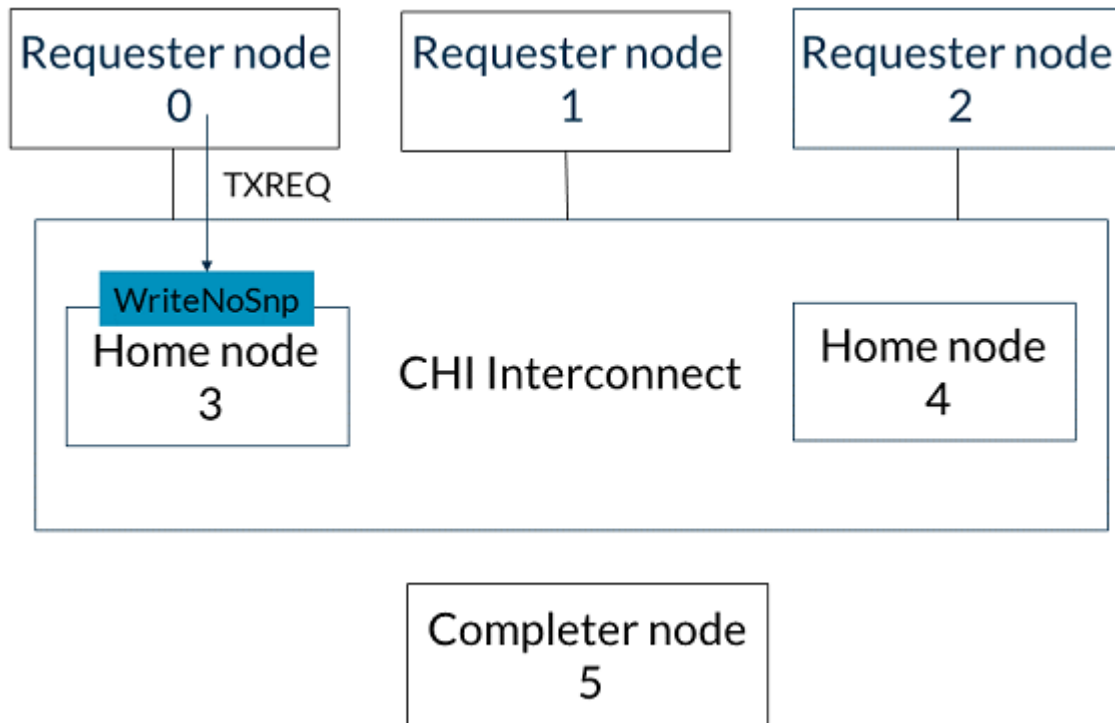
4. Home node 3 sends the `compData` response to requester node 0. Requester node 0 receives the data on its `RXDAT` channel, shown in the diagram:

Figure 4-6: A home node sending a CompData response**WriteNoSnp Transaction flow**

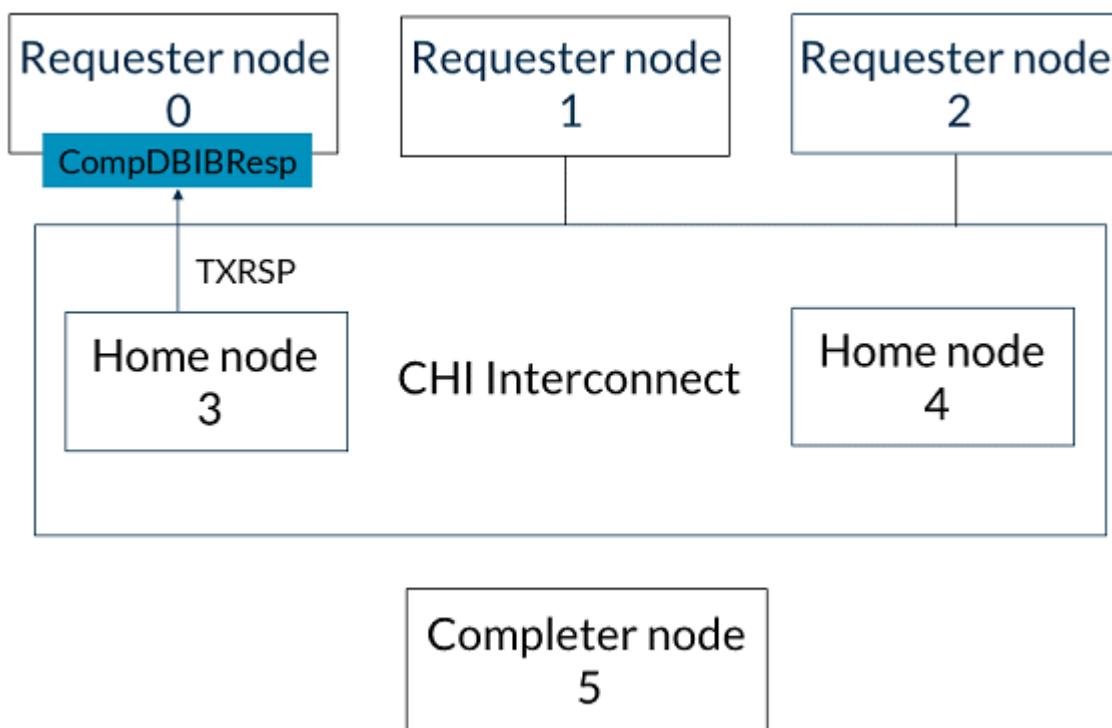
This section explains the flow of a `writeNoSnp` Transaction from Request Node 0 to completer Node 5.

The following describes the sequence of events:

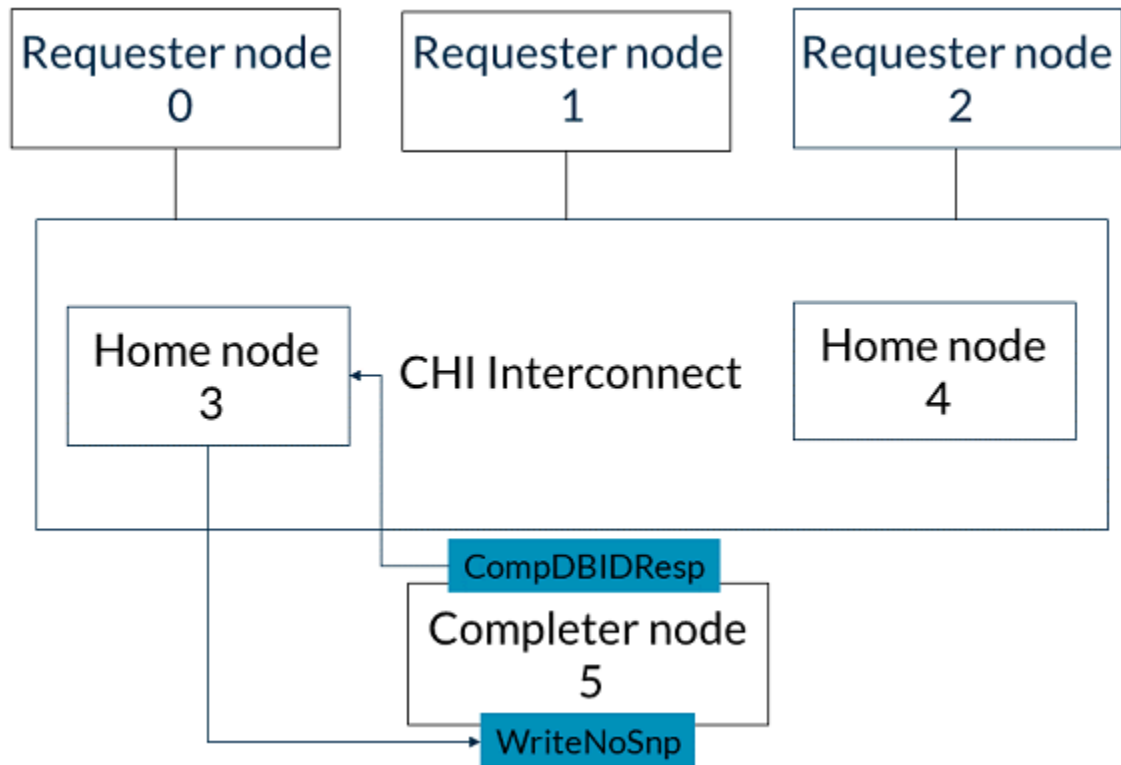
1. Request node 0 issues a `writeNoSnp` message to Home node 3 on the TXREQ channel, as shown in the diagram:

Figure 4-7: The request node issuing a WriteNoSnp

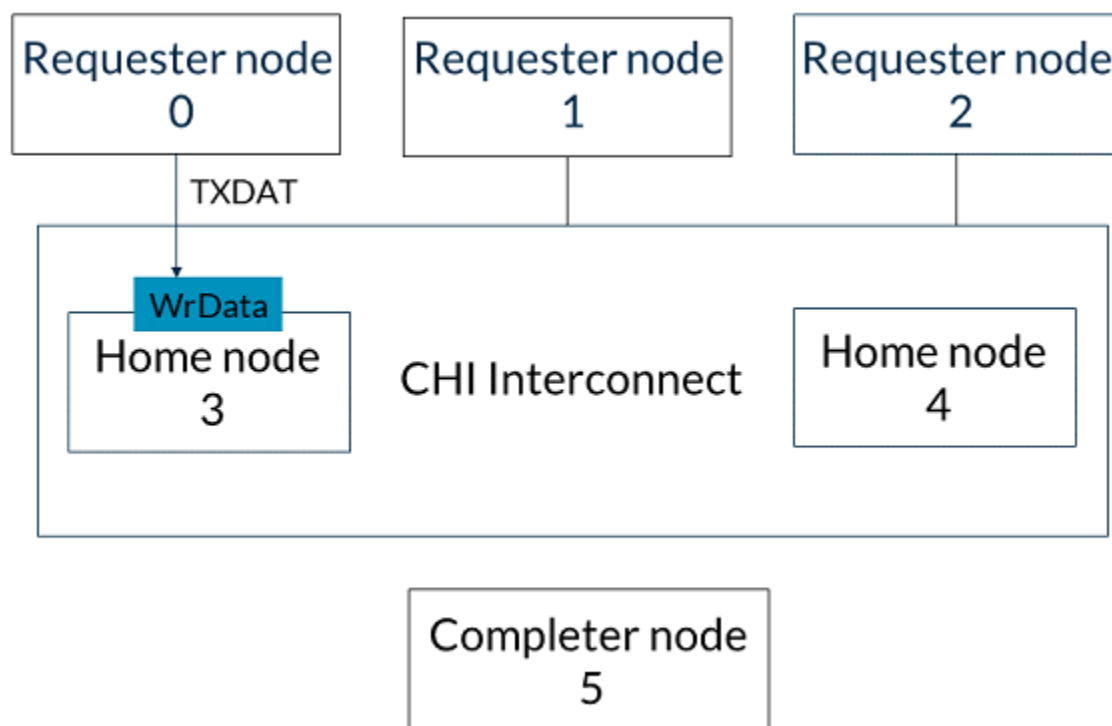
- Home node 3 responds with a `CompDBIDResp` message to request node 0. This response indicates that it can accept write data and the `WriteNoSnp` is observable by other requesters. This message gets sent over the Home node TXRSP channel. This step is shown in the following diagram:

Figure 4-8: Home node response

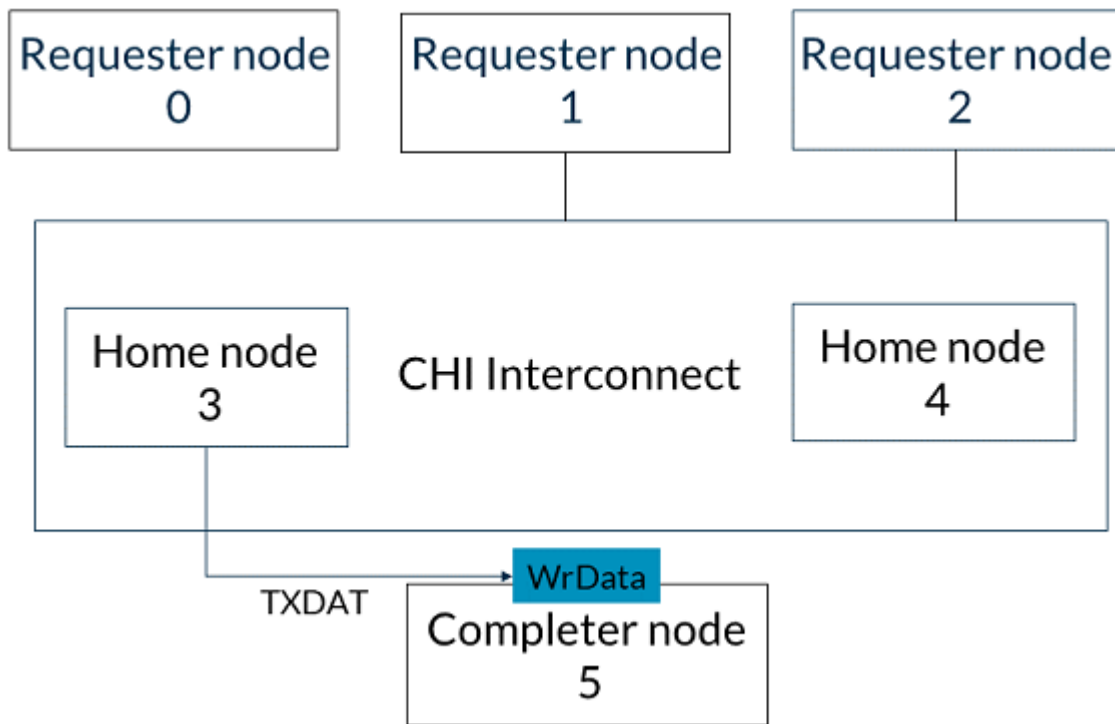
3. The following two steps can happen in any order:
 - a. Home node 3 issues the `writeNoSnp` message to completer node 5 and receives a `CompDBIDResp` response, as shown in the diagram:

Figure 4-9: Home node issuing a WriteNoSnp

- b. Alternatively, Request Node 0 could send the write data for the `writeNoSnp` over its TXDAT channel to Home node 3, as shown in the diagram:

Figure 4-10: Requester node sending write data

4. After receiving the `compDBIDResp` from the completer node and the write data from the requester node, Home node 3 sends the write data to completer node 5 on the TXDAT channel, as shown in the diagram:

Figure 4-11: Home node sending write data to the completer

Completion Acknowledgment

CHI uses a Completion Acknowledgement response to maintain the order of the following transactions:

- Transactions issued by a Fully Coherent Request Node (RN-F)
- Snoop transactions caused by these RN-F transactions

Completion Acknowledgment ensures that a snoop transaction that is ordered after a Coherent Transaction from the RN-F is received only after that Coherent Transaction completes.

HN-Fs can maintain transaction order by stalling transactions. For example, an RN-F may already have an outstanding transaction that is making progress for a particular cache line. If another requester in the system issues a transaction that results in a snoop to that same line, the HN-F can stall this later transaction. When the original RN-F completes the Coherent Transaction, the RN-F uses its TXRSP channel to send a Completion Acknowledgment (`compAck`) message to the HN-F. The HN-F then unblocks the snoops that were waiting for Completion Acknowledgement.

This mechanism has similar functionality to RACK/WACK in ACE.

Not every transaction in CHI requires Completion Acknowledgment. Request Flits contain an `ExpCompAck` field to signal when Completion Acknowledgment is needed. If Completion Acknowledgment is required, the RN-F sets `ExpCompAck` to 1 in the Request and issues a `compAck` Response when the Request completes.

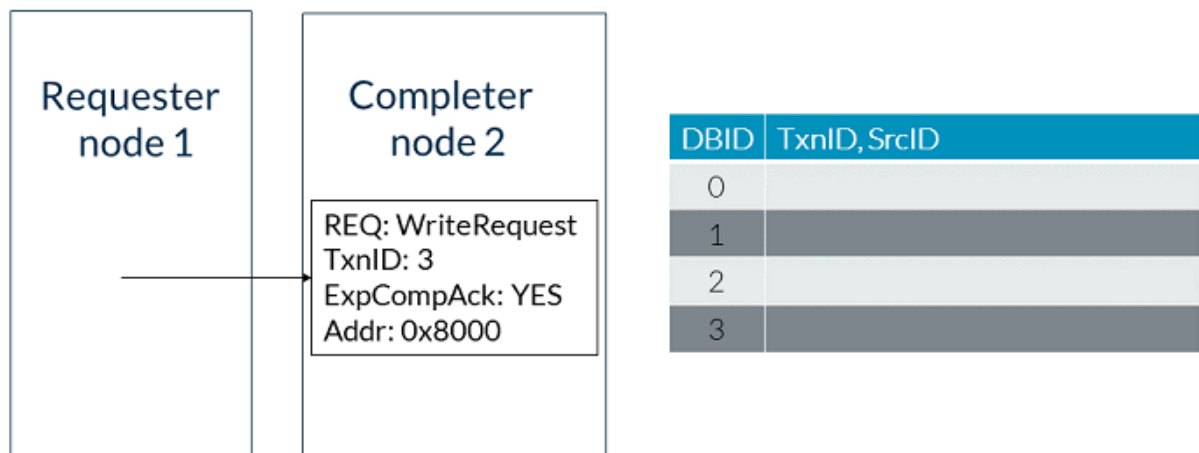
The procedure is as follows:

1. The Request Node (RN-F) issues a Request with `ExpCompAck = 1`.
2. The Home Node completes the Request.
3. The Home Node sends `comp` or `compData` to the RN-F.
4. The RN-F sends a `compAck` to the Home Node.
5. The Home Node can now send a waiting Snoop to the RN-F.

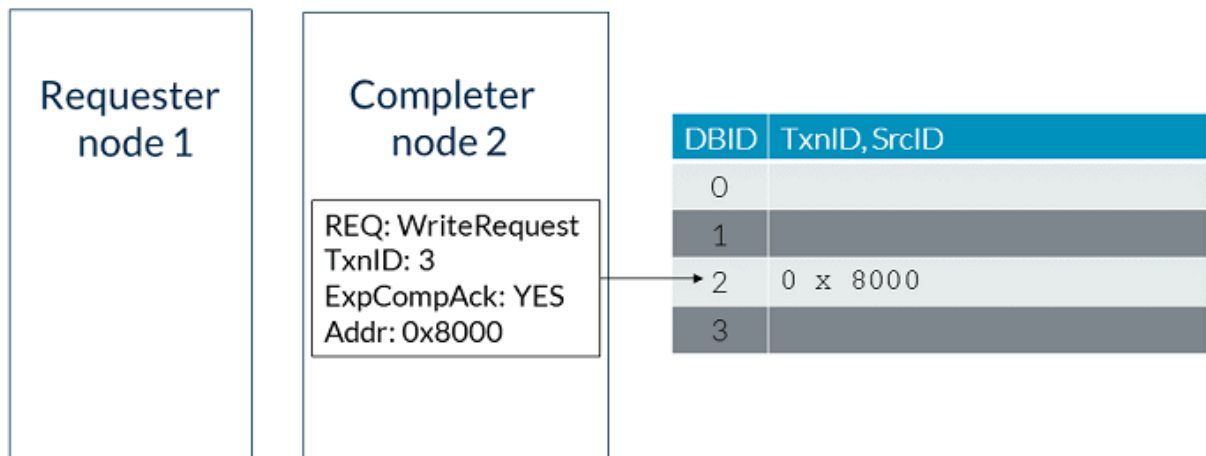
The following example shows the messages that are sent when you require Completion Acknowledgement in a Read Request:

1. The requester sends a Read Request for address `0x8000` to the completer with the `ExpCompAck` field set to 1, as shown in the diagram:

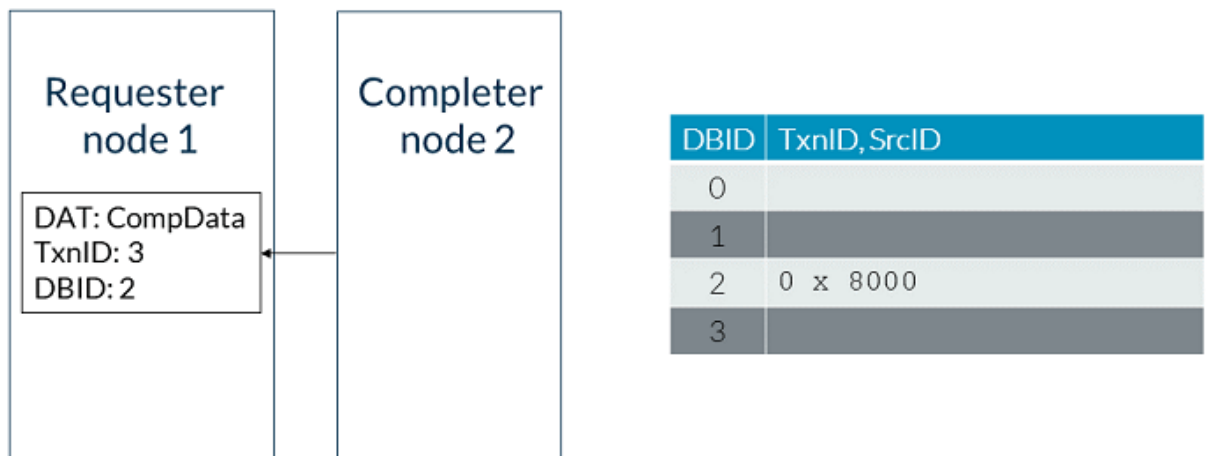
Figure 4-12: Requester sending a read request



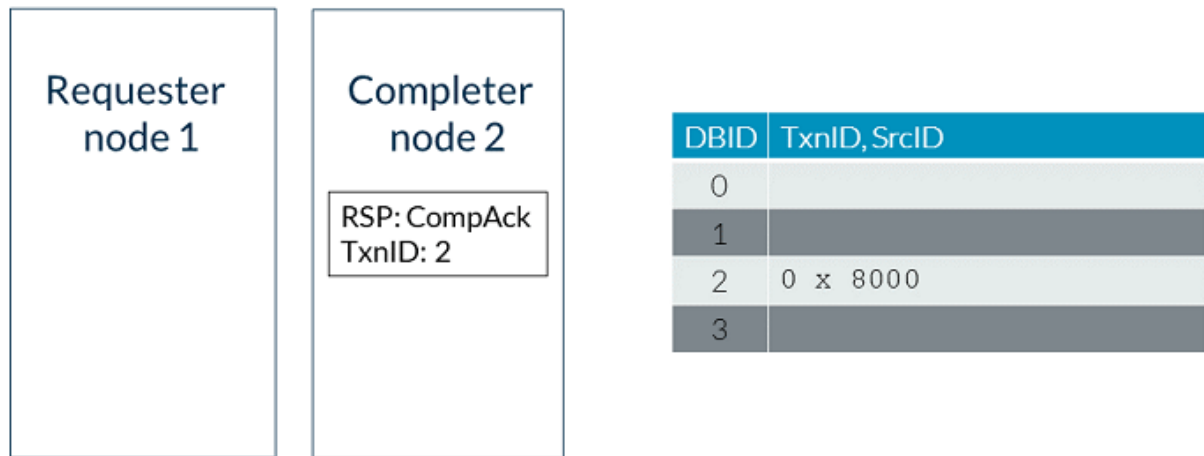
2. The completer assigns an arbitrary DBID location for the read address, blocking the interconnect from issuing snoops for future Coherent Requests to it. This location is shown in the diagram:

Figure 4-13: The DBID location

- The completer responds to the requester with a `CompData` response, signaling the completion of the transaction and sending the read data simultaneously. The `DBID` field in the response is populated with the DBID of the location that is used to store the read address. This step is shown in the following diagram:

Figure 4-14: The completer responding to the requester

- The requester sends a `CompAck` message. The `CompAck` uses the DBID value it received from the completer as the transaction ID, as shown in the diagram:

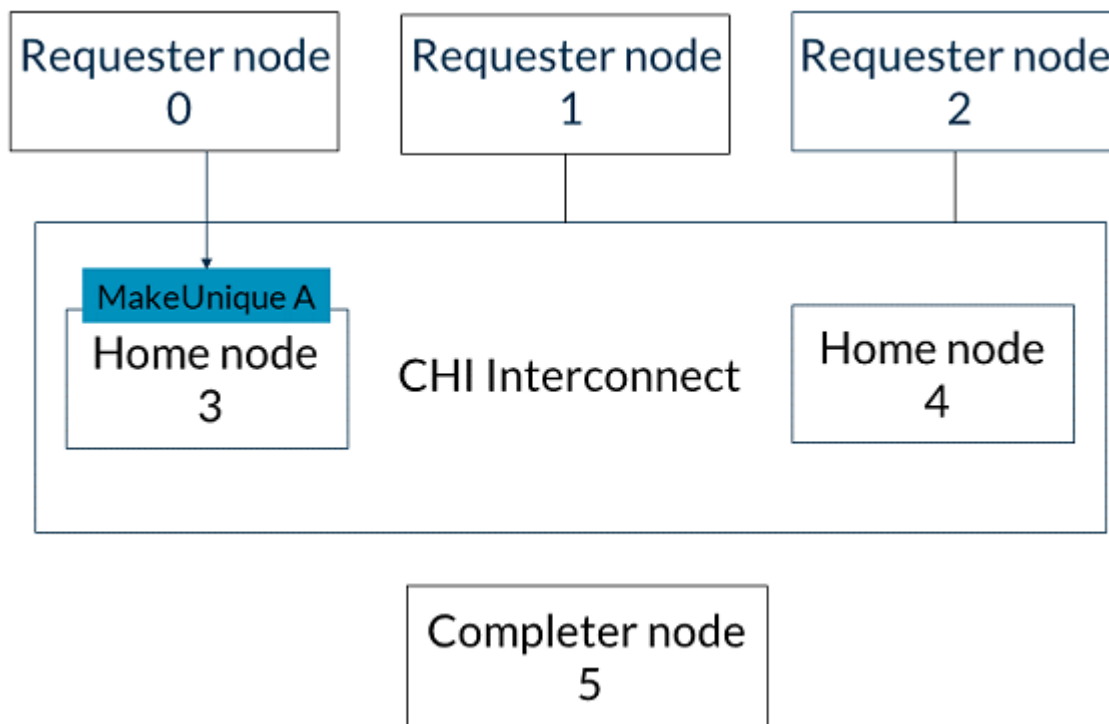
Figure 4-15: The requester sending a CompAck message

5. The completer clears the DBID location for address 0x8000, allowing the interconnect to issue future snoops to this location.

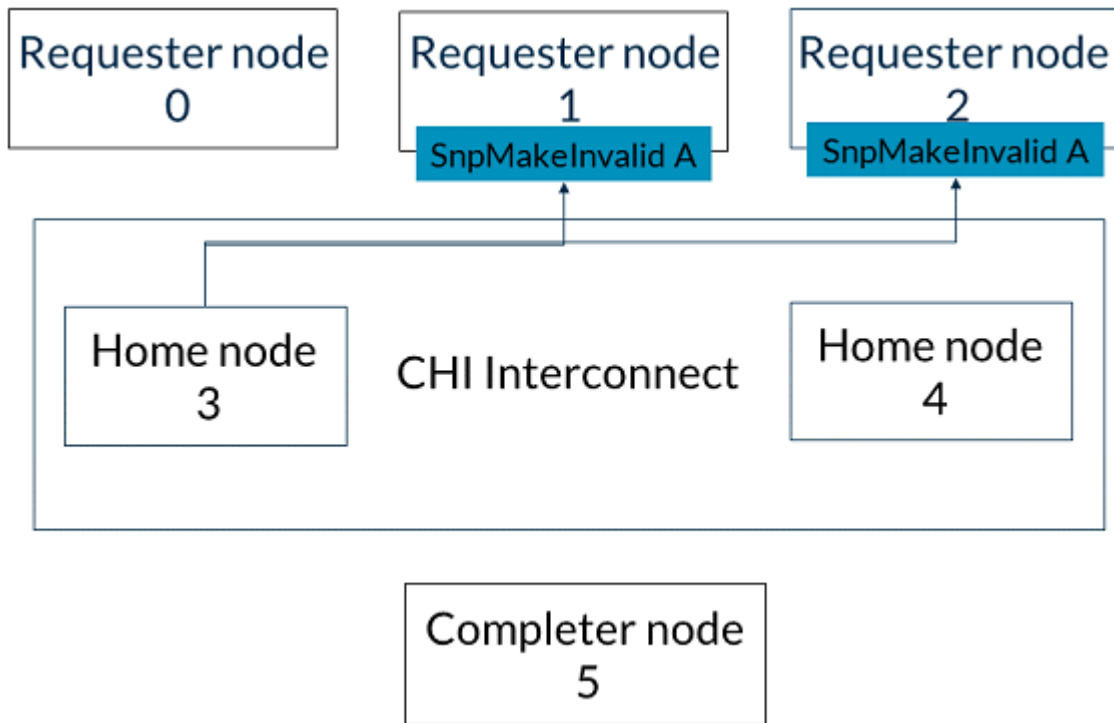
CompAck with Snoops

This example shows the messages sent when you require a Completion Acknowledgement with multiple request nodes accessing the same cacheable memory location. In this example, the CHI Interconnect broadcasts snoops to all caching requesters. Alternatively, it can utilize a snoop filter and target only requesters that had the line present locally. The following list describes the sequence of events:

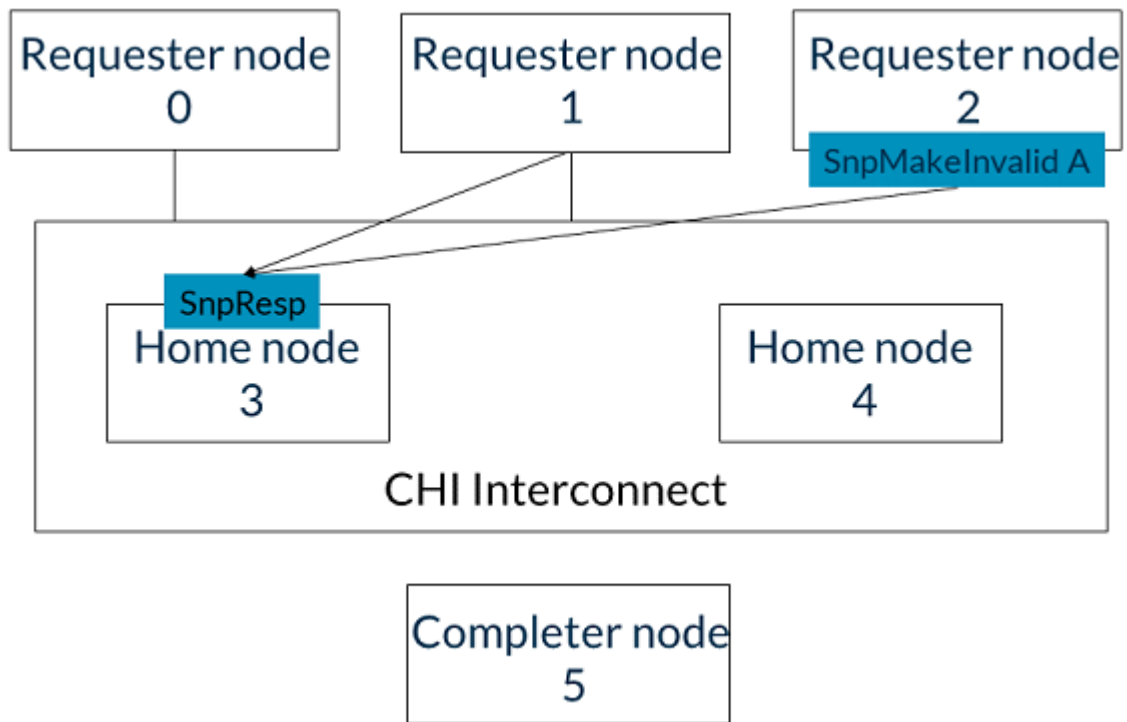
1. Requester node 0 issues a `MakeUnique` message for address A to Home node 3. This transaction completes when requester node 0 issues a Completion Acknowledge to Home node 3. This step is shown in the following diagram:

Figure 4-16: The requester sending a MakeUnique message

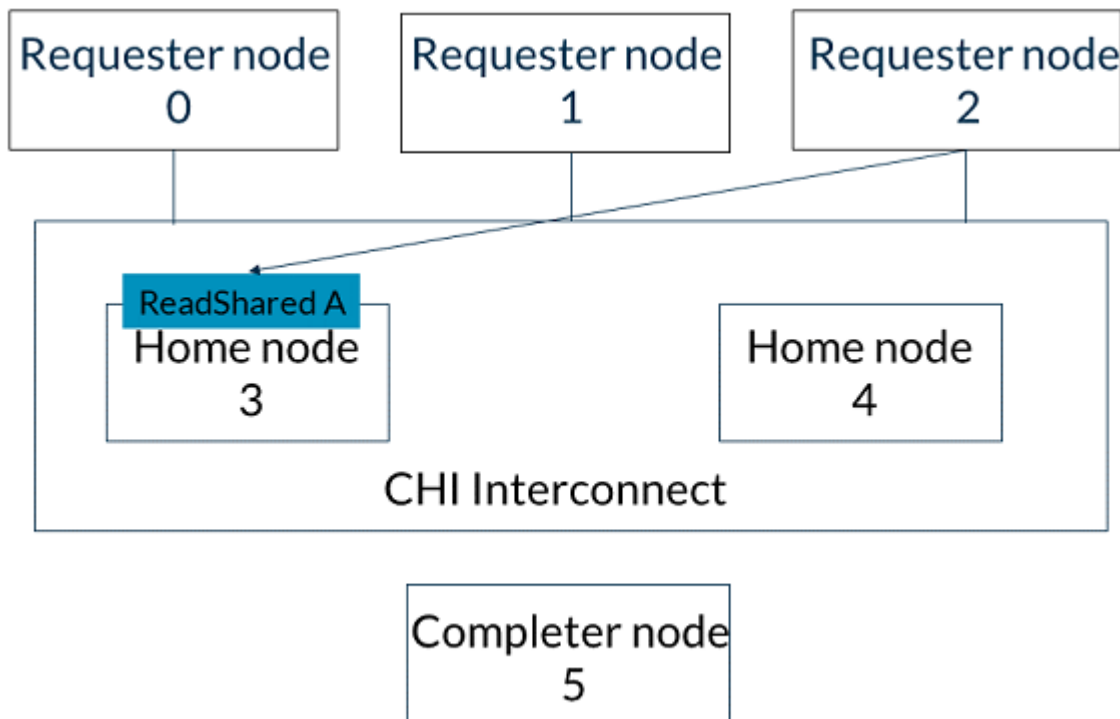
2. Home node 3 issues `snpMakeInvalid` snoops for address A to requester nodes 1 and 2, as shown in the diagram:

Figure 4-17: The home node issuing SnpMakeInvalid snoops

- Requester nodes 1 and 2 respond with `snpResp_1` responses. These responses mean that address A has been invalidated. Home node 3 can receive `snpResp_1` in any order. This step is shown in the diagram:

Figure 4-18: The requester response

4. Requester node 2 sends a `ReadShared` request for address A to Home node 3. Note that Home node 3 still has not responded to the `MakeUnique` message from requester node 0. Any Snoops generated from the `ReadShared` Request are now blocked until requester node 0 sends the Completion Acknowledge message for the `MakeUnique`. This step is shown in the diagram:

Figure 4-19: The requester sending a ReadShared request

5. Home Node 3 responds with a `comp_uc` message to Request Node 0 because of the Snoop responses it had received in step 3.
6. Request Node 0 sends the `compAck` message and unblocks Snoops to address A.
7. Home Node 3 generates `snpShared` Snoops to Request Node 0 and 1 for address A.
8. Request Node 1 responds with `snpResp`, indicating it does not have the data.
9. Request Node 0 responds with `snpRespData`, sending the most recent Data for address A. Home Node 3 can receive these two responses in any order.
10. After receiving both Snoop Responses, Home Node 3 returns the Snoop data to Request Node 2.
11. Request Node 2 sends Completion Acknowledge to Home Node 3. Home Node 3 can generate future Snoops to address A.

Endpoint Order and Request Order

Transactions in CHI can be ordered by Endpoint Order and Request Order, and are described as follows:

- Endpoint Order maintains the order of transactions from a single requester to a single subordinate address range. For example in Endpoint Order, multiple device accesses are issued to the programmable register bank of a subordinate.
- Request Order maintains the order of transactions from a single requester to the same address. For example, ordering is required when multiple requests are issued to an overlapping non-cacheable address such as Normal NC, Device-GRE and Device-nGRE. CHI does not require an

exact granularity for address matching when Request Order is set, and the granularity is defined by the implementation.



If you set Endpoint Order, Request Order is implied.

The type of ordering is controlled by the Order field in the Request Flit.

Only some Request types can use Request Order and Endpoint Order. These Request types are:

- `ReadNoSnp` and any of the `ReadOnce` type requests:
 - The requester issues a `ReadNoSnp` OR `ReadOnce` type request that requires ordering
 - The subordinate accepts the request and responds with a `ReadReceipt` message. The `ReadReceipt` signals that the next ordered request can be issued
 - By issuing the `ReadReceipt` response, the subordinate guarantees that it will maintain requests in the order in which they were received
- `WriteNoSnp` and `WriteUnique` type requests:
 - A requester issues a `WriteNoSnp` OR `WriteUnique` type request that requires ordering
 - The subordinate responds with a `DBIDResp` message to signal that it can accept the message. The `DBIDResp` response signals that a Data Buffer slot is available to accept the write data, and the requester can issue the next ordered request.
 - By issuing the `DBIDResp`, the subordinate guarantees that it maintains requests in the order they are received

For more information about request types, see [CHI specification](#).

The order of events is as follows:

1. The requester initiates Read Request 1 to the subordinate with `ReqOrder` set.
2. The requester issues Read Request 2 to the subordinate also with `ReqOrder` set, but the requester is blocked from sending the request because Request 1 is still outstanding.
3. The subordinate responds with a `ReadReceipt` message for Read Request 1, signaling that the request has been accepted.
4. In any order:
 - a. The requester sends Read Request 2 to the subordinate.
 - b. The subordinate returns the read data for Read Request 1 to the requester.

Request retry

Sometimes a target node might not have enough resources to accept a request.

To prevent blocking the Request channel when resources are not available, CHI provides a Request Retry mechanism. The Request Retry mechanism uses Protocol Credits to indicate resource

availability. It is the responsibility of the subordinate to determine and record the type of Protocol Credit (`PCrd`) that is required to process the request.

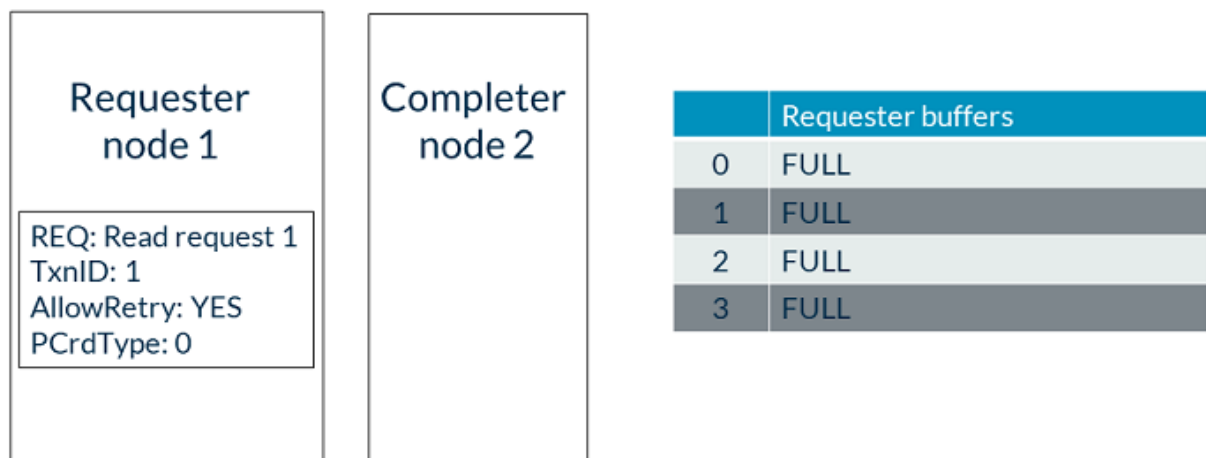
The mechanism can use different types of Protocol Credit to track different resources. For example, Read Requests and Write Requests can use separate Data Buffers, so each buffer can use a different type of Protocol Credit to indicate availability. The values for different types of Protocol Credit are defined by the implementation.

The following example describes the sequence of messages sent with a Request Retry. In this example, requester node 1 issues a request because the completer is unable to accept the request.

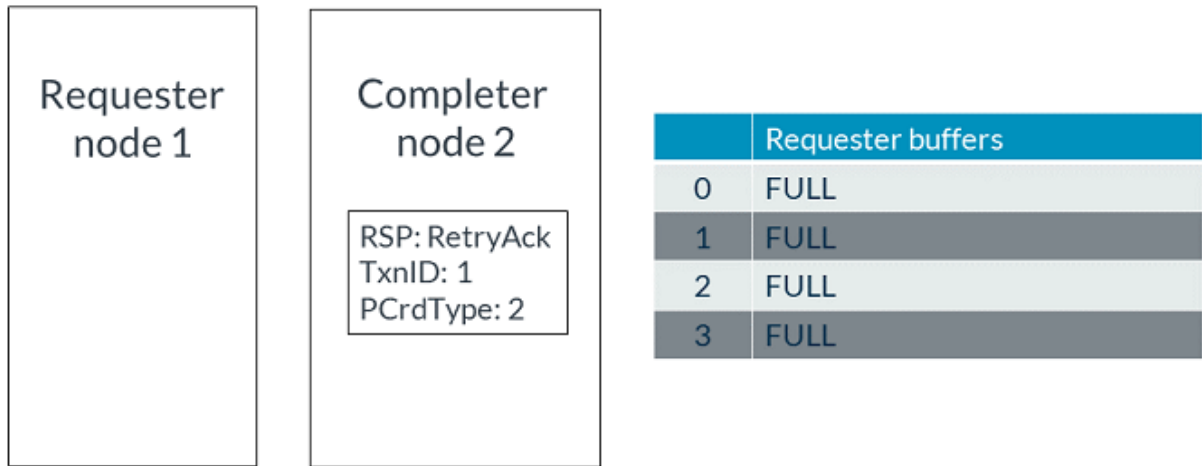
The following describes the sequence of events:

1. Every request is initially issued without a Protocol Credit. The Request Flit has a control field called `allowRetry`. Setting this field to YES the first time that the request is sent indicates that the request is not using a Protocol Credit. When `allowRetry` is YES, the `PCrdType` field in the request must be 0. The following diagram shows the request settings:

Figure 4-20: Request settings



2. The target in this example is unable to accept the Request because the requester buffer is full, so it responds with a `RetryAck` message.
3. The `RetryAck` Response Flit sets a `PCrdType` field with a value indicating the type of credit that is required to retry the Request. In this example the value of `PCrdType` is 2, as shown in the diagram:

Figure 4-21: The PCrdType value

4. When the target can accept the Request, it sends a PCrdGrant message on the RSP channel. The PCrdGrant Response Flit uses the PCrdType field to indicate the type of Protocol credit that has become available. The requester must only retry a Request if the Protocol credit types in the PCrdGrant messages and the RetryAck response match. In this example, both fields must be set to 2. If the protocol credit types match, the target node has guaranteed it can now accept the Request.
5. The requester reissues the Request with the AllowRetry field set to 0. Setting the AllowRetry field to 0 indicates to the target node that the Request is using a granted Protocol Credit.

5. DVM operations

Like ACE, CHI supports Distributed Virtual Memory (DVM) operations. DVM Requests pass operations to support the maintenance of a virtual memory system.

DVM operation transactions

CHI uses DVM operations to manage virtual memory.

The DVM operations conduct the following transactions:

- Transaction Lookaside Buffer (TLB) Invalidation
- Instruction Cache Invalidation
- Branch Predictor Invalidation
- DVM Synchronization

In CHI, all DVM operations are sent to the MN in two parts. This is different from ACE, where some DVM operations require two parts, and others require just one.

The following list describes the parts order in CHI:

- The first part of the DVM operation is sent as a Request to the MN, with the Opcode field set to `DVMOp`. The Request Flit uses the Address field to encode the attributes of the operation.
- The second part of the DVM gets sent as a Data Flit, only after the Request Node has received a DBID response from the MN. This second part carries the address that is targeted by the DVM operation.

When the MN receives both parts of the DVM operation, the MN generates DVM Snoops to the Request Nodes that are participating in the coherency domain. The MN sends the DVM Snoops in two parts on the Node Snoop channel.

Both parts of the DVM Snoop must use the same `TXNID` and the Opcode `snpdvmop` and use the following parameters:

- The first part uses the Address field to encode the operation attributes and the upper address bits of the target address
- The second part uses the Address field to send the remaining bits of the address

To differentiate the two parts, CHI requires that `bit[3]` of the `Address` field is set to 0 to indicate the first part and 1 for the second part.

The second part of the DVM snoop can arrive at the RN before the first part.

DVM operation types

CHI defines two types of DVM operations: Non-Synchronization DVMs (DVM Non-Sync) and Synchronization DVMs (DVM Sync). The attributes of a DVM operation dictate whether an RN must wait for an operation to complete before responding to a DVM Snoop.

Synchronization DVMs perform only synchronization, with no other operation.

Non-Synchronization DVMs are the Invalidation operations for TLB, Instruction Cache, and Branch Predictor. Non-Synchronization DVMs do not require the DVM operation to finish executing before more DVM operations are issued. This allows multiple Non-Synchronization DVMs to be outstanding.

In the following example, an RN-F can issue multiple Branch Predictor or Instruction Cache Invalidations, and the receiving RN-F or RN-D does not have to execute the operation immediately:

1. An RN-F or RN-D receives a DVM Snoop indicating a DVM Non-Sync.
2. The RN-F or RN-D issues a Snoop Response to the MN. The Snoop Response acknowledges receipt of the DVM message but does not indicate that the Request Node has executed the DVM operation.
3. The MN sends a Completion message to the initiating RN-F to indicate that the DVM operation has been accepted.

To ensure that all outstanding DVM Requests have executed, the following steps occur:

1. An RN-F issues a Synchronization DVM operation, also known as a DVM Sync, to the MN. Any DVM requests that need to be completed by the DVM Sync must receive their Completion response before issuing the DVM Sync.
2. The MN issues the DVM Sync to all the RN-Fs and RN-Ds on the snoop channel.
3. Each target RN ensures all its outstanding DVM operations have been executed.
4. Each RN issues one snoop response to the MN, indicating that all operations have been executed.
5. The MN sends a Completion response for the DVM Sync to the RN-F that originally issued the Synchronization DVM operation.

CHI DVM Syncs are similar to those in ACE. Both check that previously issued DVM operations have been completed. The difference is that CHI does not require a DVM Complete message.



Note

Arm cores generate DVM Syncs as a result of a DSB instruction. However, implementations may choose to issue DVM Syncs as a result of a DSB only if there are DVM Ops that have not already been synced.

DVM operation flow

This section describes a TLB Invalidation DVM Request followed by a Synchronization DVM operation and shows the following events:

- The different parts of a DVM Request
- The Snoops generated by the MN
- How a DVM sync ensures that prior DVM operations have been executed

The order of events is as follows:

1. Request Node 0 issues a TLB Invalidation DVM Request to the MN.
2. The MN responds with a DBIDResp message indicating that it can accept the second part of the DVM Request.
3. Request Node 0 issues a Write Data message to the MN. This is the second part of the DVM message.
4. The MN issues both DVM Request parts to Request Node 1.
5. Request Node 1 acknowledges the DVM Request by sending a Snoop Response to the MN.
6. The MN receives the Snoop Response.
7. The MN issues a Completion message to Request Node 0.
8. Request Node 0 issues a DVM Sync operation to the MN.
9. The MN responds with a DBIDResp message to Request Node 0.
10. Request Node 0 sends the Write Data message to the MN. This is the second part of the DVM Sync message.
11. The MN issues the DVM Sync Snoops to Request Node 1.
12. Request Node 1 completes all outstanding DVM operations.
13. Request Node 1 sends a Snoop Response to the MN, indicating it has completed all operations.
14. The MN issues a Completion message to Request Node 0. This is the response to the DVM Sync Request.

6. Cache stashing

Cache stashing is a mechanism to install data within specific caches in a system. CHI-B introduced this feature to improve system performance. The Cache Stashing mechanism improves system performance by allocating a cache line near its future point of use. This results in lower memory access latency when the data is used.

Typically, Cache Stash Requests are initiated by RN-I and RN-D Nodes. A Cache Stash Request is a suggestion, not a mandatory action, that a particular cache line should be installed within a particular cache in the system. The Device receiving the Cache Stash Request can ignore the Request.

CHI supports two main forms of Cache Stashing: Stash Transactions that contain write data, and Stash Transactions that are dataless. Both forms of cache stashing can target different cache levels as the stash target.

Cache Stashing support was added to the ACE5-Lite protocol. The CHI protocol is very flexible regarding Cache Stashing and allows a Stash Request to take multiple forms.

This section of the guide describes the different forms of Stash Requests and the flow of messages for these Requests.

Transaction flow

The basic transaction flow of cache stashing is as follows:

1. An RN issues a Cache Stash Request on the Request Channel.
2. The Cache Stash Request is forwarded to an HN-F.
3. The HN-F can either:
 - Ignore the Cache Stash Request. The RN-F treats the Stashing Snoop as the non-stash version and responds accordingly. Or,
 - Accept the request and generate snoops to RN-Fs. The RN-F responds fetches the cache line into its cache.
4. The RN-F targeted for the stash receives a particular type of noop Request called a Stashing Snoop.

The RN-F can:

- Provide a snoop response that acts as a read request for the associated cache line using the `DataPull` mechanism
- Provide a snoop response without `DataPull` then issue a separate read request for that cache line
- Provide a snoop response without fetching the line, ignoring the cache stash hint

Stashing Snoop Requests

All cache stash requests get sent to HN-F nodes. When an HN-F processes a cache stash request, it generates stashing snoops to the target RN-F.

CHI defines four different stashing snoop requests, each corresponding to the initial cache stash transaction. These requests are detailed in the following table:

Cache Stash Transaction	Snoop Request by HN-F	Action by RN-F
WriteUniquePtlStash	SnpUniqueStash	Invalidate cache line and return the data if it is Dirty (with write data)
WriteUniqueFullStash	SnpMakeInvalidStash	Invalidate cache line if present (with write data)
StashOnceShared	SnpStashShared	Make a Shared Request for the cache line (dataless)
StashOnceUnique	SnpStashUnique	Make a Unique Request for the cache line in preparation for a future write (dataless)

Cache stashing control fields

CHI adds control fields to the request, snoop, response, and data flits for cache stashing. These fields indicate:

- The NodeID of the stash target
- A specific logical processor cache inside an RN-F, like the L2 cache
- Whether the `DataPull` mechanism will be used

The Request Flit uses the following fields for a Cache Stash Request:

- `stashNID` holds the Node ID of the stash target. If an RN-F is chosen as the target for a stash, the `stashNID` field is populated with the Node ID of the RN-F.
- `stashNIDValid`. If the `stashNID` field should be used when stashing, `stashNIDValid` will be 1.
- `stashLPID` specifies the Logical Processor ID within an RN-F. This field allows specifying a lower-level cache, like an L2 cache, as a stash target.
- `stashLPIDValid`. If the `stashLPID` field should be used when stashing, `stashLPIDValid` will be 1.

The Snoop Flit also contains the following fields:

- `stashLPID` and `stashLPIDValid`. If the Cache Stash Request indicated that `stashLPID` was Valid (`stashLPIDValid` = 1), the Snoop will use the `stashLPID` value from the Cache State Request. If no `stashLPID` is specified (`stashLPIDValid` = 0), then the Stashed data may be placed in a shared cache within the RN-F.
- `DoNotDataPull`: If this field is set to 1, the stash target cannot request `DataPull`, and therefore cannot use the `DataPull` mechanism.

Transactions with write data

The requester issues a `WriteUniqueStash` Transaction if it is writing new data and requires a target to stash that data. The data written can be a full or partial cache line.

CHI uses one of the following opcodes to indicate a write with a stash hint:

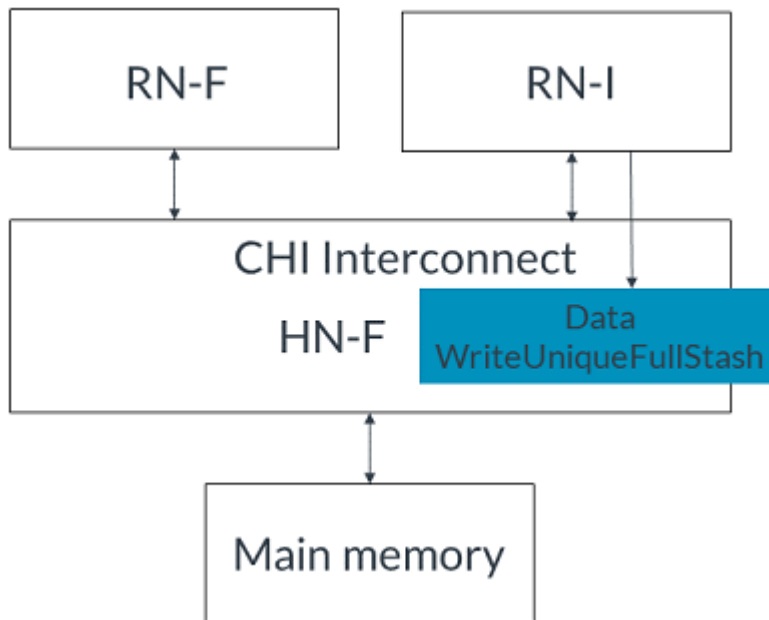
- `WriteUniquePtlStash` for partial cache line write
- `WriteUniqueFullStash` for a full cache line write

This section describes how an I/O requester issues a stash hint with write data. The target of the Stash Transaction is the RN-F in the system.

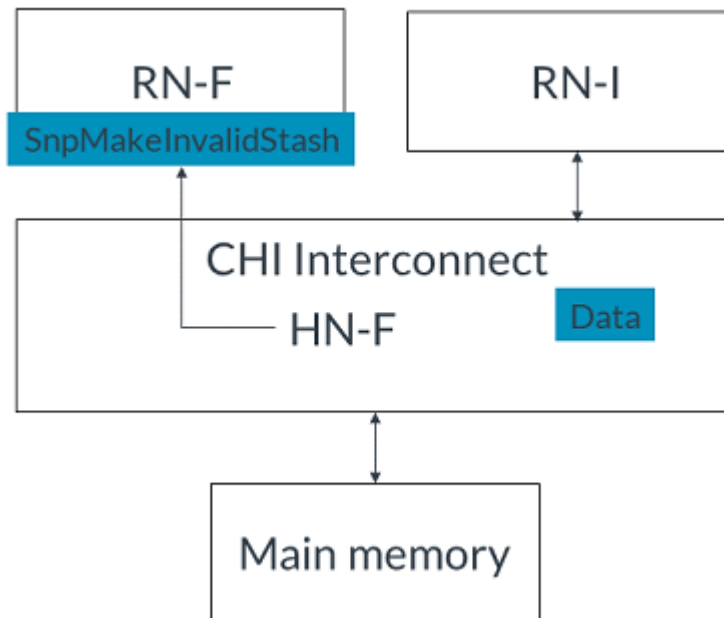
The order of events is as follows:

1. The RN-I issues a `WriteUniqueFullStash` request with write data. For simplicity, this example does not describe the `DBIDResp` from the HN-F. This request is shown in the following diagram:

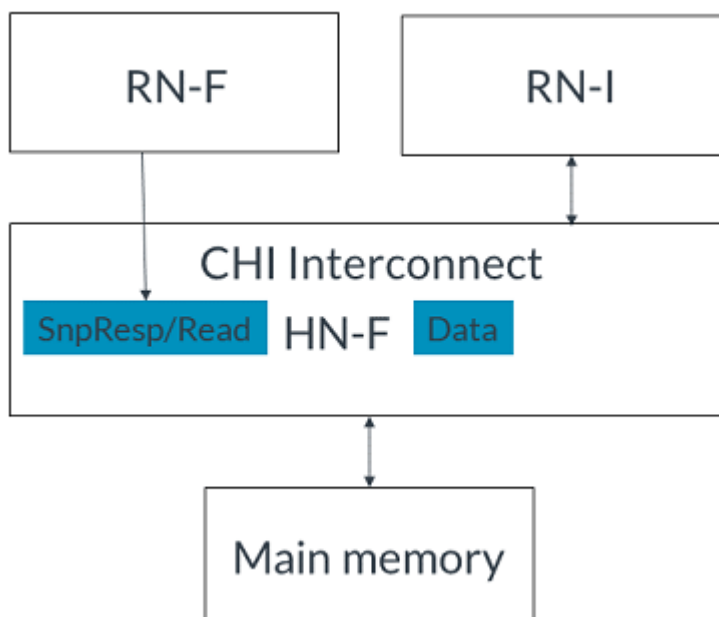
Figure 6-1: The RN-I request



2. The HN-F accepts the stash request then issues an `snpMakeInvalidStash` request to the RN-F, as shown in the following diagram:

Figure 6-2: The HN-F request

3. The RN-F receives the Snoop from the NH-F.
4. The RN-F accepts the stash hint and issues a Snoop Response, as shown in the diagram:

Figure 6-3: The RN-F issuing a snoop response

5. If the `DataPull` mechanism is used, the RN-F issues either a Snoop Response with an implicit Read Request, or a Snoop Response and a separate Read Request. For simplicity, an implicit Snoop Response and Read Request are used in this example, but not the full `DataPull` flow.

6. The HN-F sends the write data that it received from the RN-I to the RN-F.

Transactions without write data

Requesters use dataless Stash Transactions when they use a cache as a stash target without writing data. CHI uses the following opcodes for dataless stash requests:

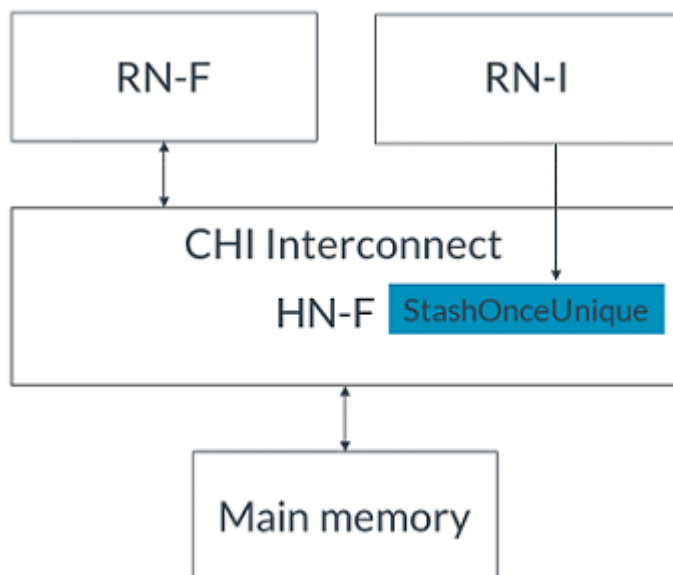
- `StashOnceShared` is issued if the cache line is expected to be read by the stash target. This opcode indicates that the cache line should be held in the Shared state after allocation.
- `StashOnceUnique` is issued if the cache line is expected to be written to by the stash target. This opcode indicates that the cache line should be held in the Unique state, allowing the stash target to immediately write to the cache line when it needs to in the future.

The following example describes a stash hint without write data. The RN-I sends the Stash Request with the RN-F as the stash target, and both the HN-F and RN-F accept the stash hint.

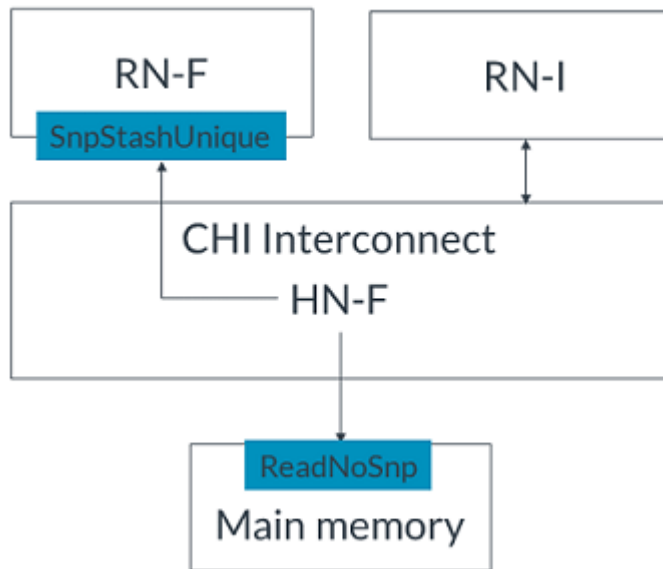
The sequence of events is as follows:

1. The RN-I issues a `StashOnceUnique` Request to the HN-F, indicating that the RN-F is the target and no write data is present. The following diagram shows this step:

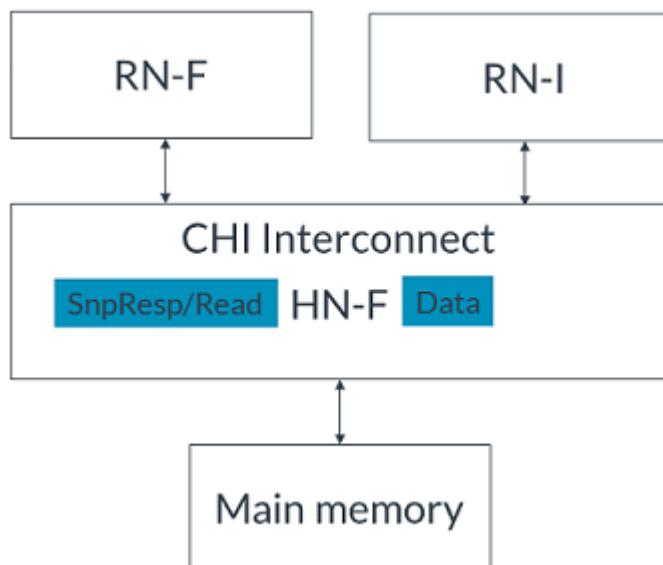
Figure 6-4: The RN-I request



2. The HN-F accepts the Stash Request.
3. The HN-F issues a `ReadNoSnp` Request to Main Memory to fetch the cache line and an `snpStashUnique` Snoop to the RN-F, as shown in the diagram:

Figure 6-5: The HN-F request

4. Main Memory returns the cache line to the HN-F, as shown in the diagram:

Figure 6-6: The main memory returning the cache line

5. The RN-F responds to the Snoop, requesting the cache line.
6. The HN-F forwards the cache line to the RN-F.

A Stash Request does not require a valid stash target. If the stash target is not specified, the HN-F targeted in the Request becomes the stash target. The HN-F then chooses whether to allocate the cache line into its cache or not.

The following steps describe a stash hint without write data targeting the system cache. The order of events is as follows:

1. The RN-F issues a `stashOnceShared` Request to the HN-F. The `stashNIDvalid` field is set to 0 to target the HN-F.
2. The HN-F issues a `readNoSnp` Request to Main Memory to get the specified cache line.
3. Main Memory returns the cache line to the HN-F.
4. The HN-F allocates the cache line into its cache.

The DataPull mechanism

The `DataPull` mechanism is a way to imply a Read Request through a Snoop Response so that a separate Read Request is not needed to fetch the cache line being stashed. `DataPull` is only applicable to Stashing Snoop Requests, not to any other snoop.

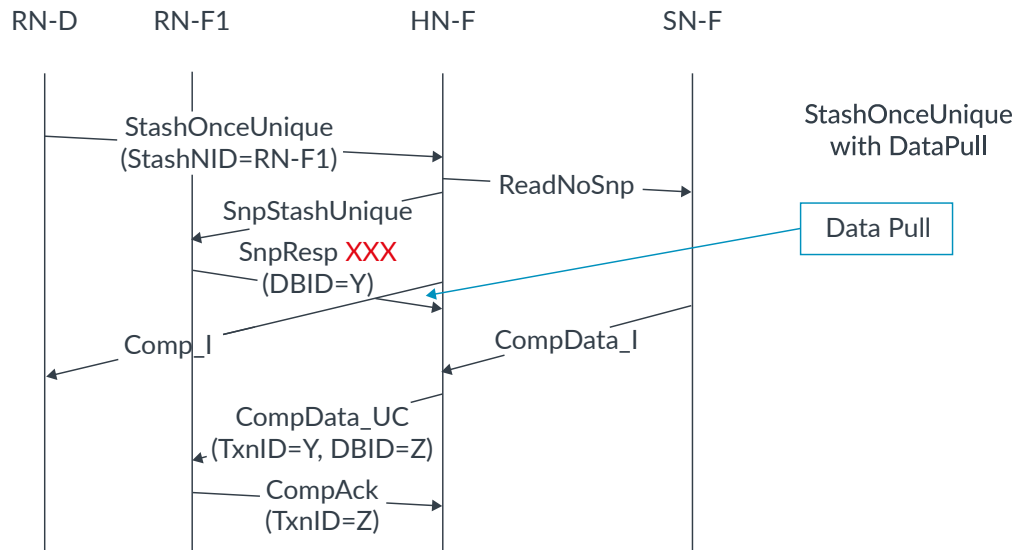
An RN-F receiving a Request asking for `DataPull` can choose whether to use `DataPull` or send a separate Read Request. If the RN-F chooses not to Request `DataPull`, it responds to the snoop and can send the Read Request later to obtain the cache line.

For information on all the permitted Snoop Responses with a `DataPull` refer to the [AMBA 5 CHI Architecture Specification](#).

In this section, we describe the process of a RF-F utilizing the `DataPull` mechanism to receive the data as part of the Stashing transaction. The full transaction flow for the `DataPull` is as follows:

1. The HN-F issues a Stashing Snoop and sets the `DoNotDataPull` field in the Snoop Flit to 0. This indicates that the RN-F stash target can request `DataPull`.
2. The RN-F that received `DoNotDataPull` = 0 can choose to request `DataPull` in its Snoop Response. In this example, the RN-F chooses to request `DataPull`.
3. The RN-F requests `DataPull` by setting two fields in the Response Flit:
 - The `DataPull` field is set to 1
 - The `DBID` field is populated with the `TxnID` that will be used to return the read data
4. The RN-F receives the read data.
5. The RN-F issues a `CompAck` message to the HN-F.

The following diagram shows the timing of the `DataPull` mechanism for a `StashOnceUnique` Transaction:

Figure 6-7: DataPull mechanism

In this diagram, the example system contains:

- One RN-D initiating the Request
- One RN-F as the stash target
- One HN-F
- One SN-F

The full transaction flow in the example is as follows:

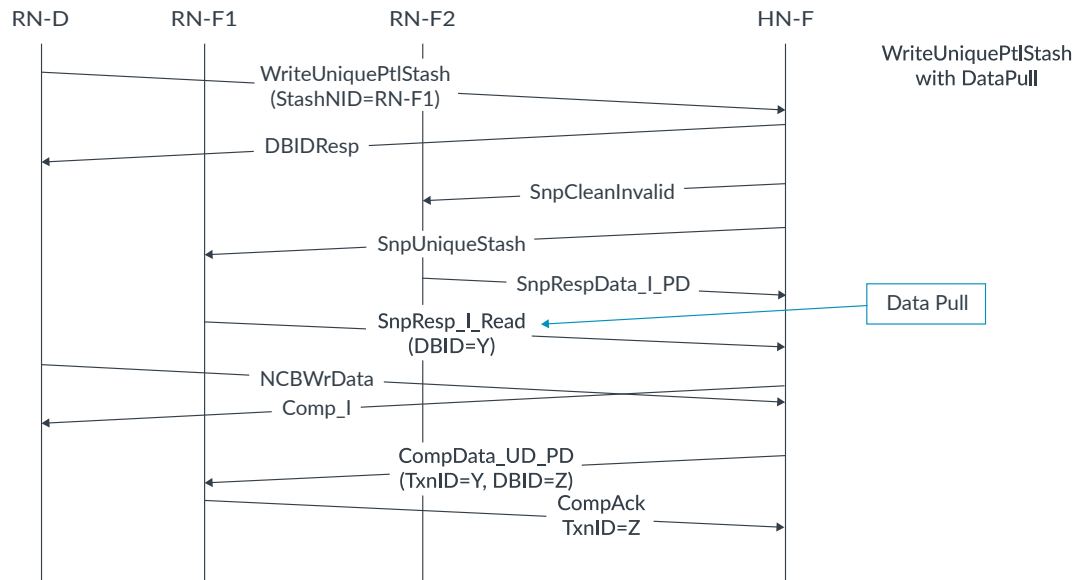
1. The RN-D issues a `stashOnceUnique` Request to the HN-F. The value of the `stashNID` field indicates that RN-F1 is the stash target.
2. The HN-F accepts the Stash Request.
3. The HN-F issues:
 - A `ReadNoSnp` Request to the SN-F
 - A `snpStashUnique` Snoop to RN-F1
4. The HN-F sends a Completion response to the RN-D.
5. RN-F1 accepts the stash hint.
6. RN-F1 responds to the `snpStashUnique` with a `SnpResp_I_Read DataPull` Request. The `snpResp_I_Read` response signals the implicit Read Request. The `DBID` field sets the Transaction ID as `Y`.
7. The SN-F returns the cache line to the HN-F.
8. The HN-F forwards the cache line to RN-F1 with:

- TxnID = Y
- DBID = Z

9. RN-F1 issues a Completion Acknowledge response to the HN-F with TxnID = Z.

The next example shows the timing of the DataPull mechanism for a WriteUniquePtlStash Transaction:

Figure 6-8: DataPull timing



In this example, the system has:

- One RN-D node
- Two RN-F nodes: RN-F1 and RN-F2. RN-F2 holds the cache line when the Stash Request is sent.
- One HN-F node

The transaction flow in the example is as follows:

1. The RN-D issues a WriteUniquePtlStash Request to the HN-F. The stash target is RN-F1.
2. The HN-F accepts the Stash Request.
3. The HN-F returns a DBIDResp to the RN-D.
4. The HN-F generates an snpCleanInvalid Snoop to RN-F2. This is because RN-F2 holds the cache line and a snpUniqueStash to the stash target RN-F1.
5. RN-F2 invalidates the cache line.
6. RN-F2 returns a Snoop Response with Dirty data to the HN-F.

7. RN-F1 issues a Snoop Response with an implicit Read Request. The DBID field sets the `TxnID` of the stash data as `y`.
8. The HN-F receives the Snoops response.
9. The HN-F issues a completion response to the RN-D.
10. The RN-D sends the write data for the Stash Request to the HN-F. The HN-F now has both the write data and the data from the Snoop response. The HN-F creates the new data for the cache line.
11. The HN-F sends ownership of the cache line to RN-F1. The response fields are:
 - `TxnID` = `y`
 - `DBID` = `z`
1. The RN-F1 sends a `compAck` response with `TxnID` = `z`.

7. I/O Deallocation

CHI-B provides the ability for an I/O Requester to deallocate cache lines in a Fully Coherent Node.

I/O Deallocation transactions provide hints that a cache line should be invalidated, and that Dirty data should be either written back to memory or discarded.

Because these requests are only hints, a Fully Coherent Node can choose not to invalidate a cache line and simply return data to the I/O requester. In other words, if the invalidation hint is ignored, the requests are treated as normal `ReadOnce` transactions. Because they can be ignored, I/O deallocation requests are not a replacement for Cache Maintenance Operations.

CHI defines two types of requests for I/O deallocation: `ReadOnceCleanInvalid` and `ReadOnceMakeInvalid`. Both Requests are useful for avoiding cache pollution with data that will not be used again in the near future. The difference between the two request types is that `ReadOnceMakeInvalid` is not required to write Dirty data to the next level of memory, which can lead to Dirty data being discarded in the system. This means that care must be taken when using this type of request.

I/O Deallocation transaction examples

This section describes two examples of transaction flow. The first example uses `ReadOnceCleanInvalid` and writes the Dirty data back to Main Memory. The second example uses `ReadOnceMakeInvalid` and discards the Dirty data.

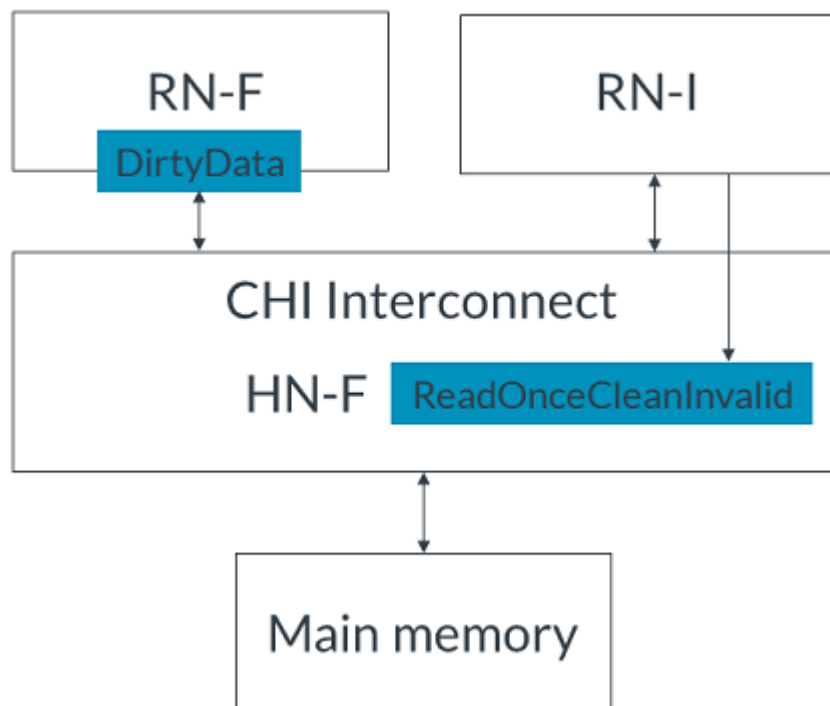
The system in both examples has:

- A Fully Coherent Request Node (RN-F) Device. The RN-F holds the requested cache line in the Dirty state
- An I/O Coherent Request Node (RN-I) Device
- A CHI interconnect
- Main Memory

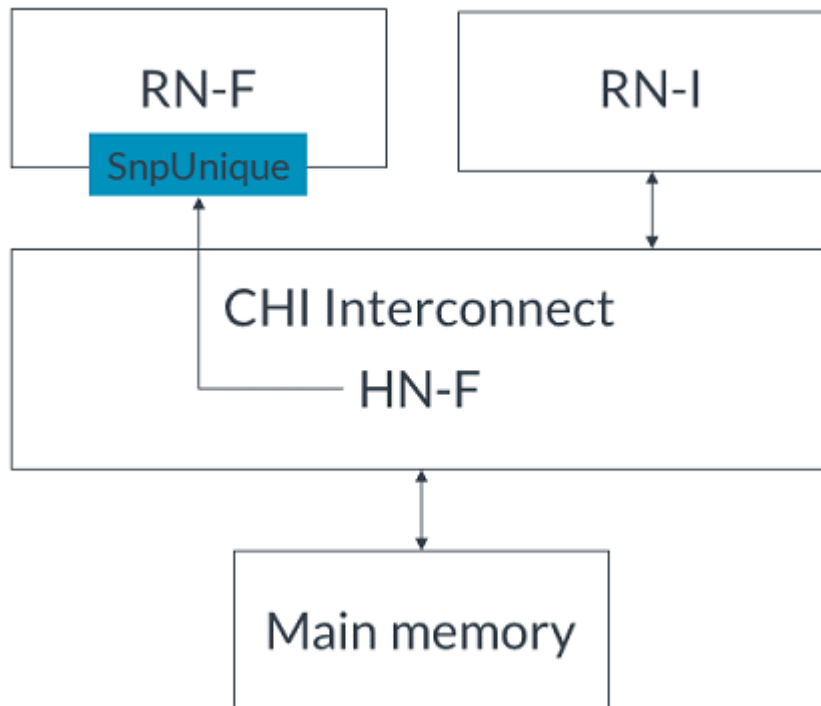
In the first example, a `ReadOnceCleanInvalid` Transaction flow reads and invalidates the data and writes it back to Main Memory.

The full transaction of this example is as follows:

1. The RN-I issues a `ReadOnceCleanInvalid` Transaction to the RN-F, as shown in the diagram:

Figure 7-1: The RN-I transaction

2. The HN-F sends a SnpUnique Request to the RN-F, requesting the cache line, shown in the following diagram:

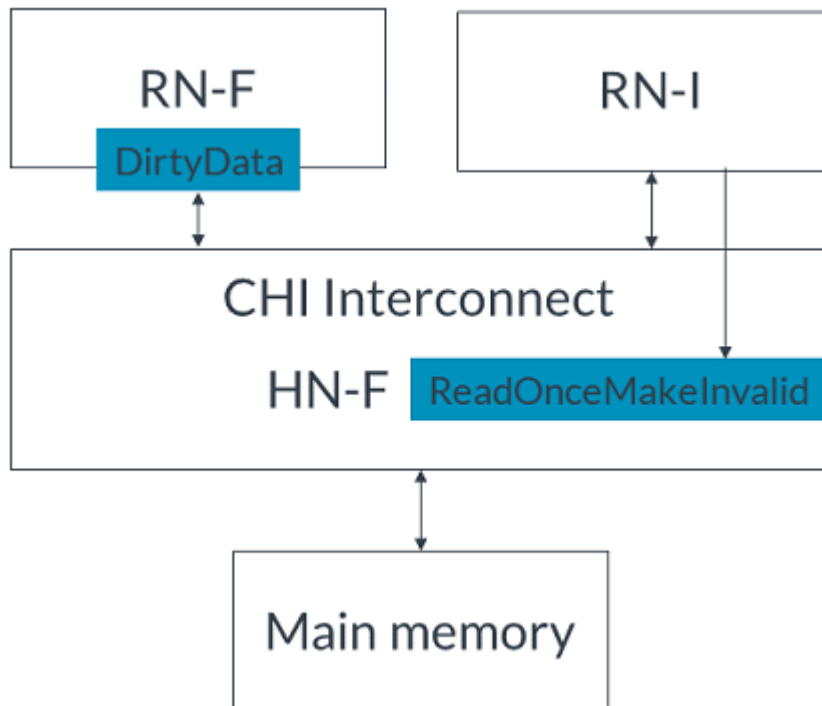
Figure 7-2: The HN-F request

3. The RN-F invalidates the cache line and sends the Dirty data to the HN-F.
4. The HN-F returns the data to the RN-I and writes the data to Main Memory, leaving it in the Clean state.

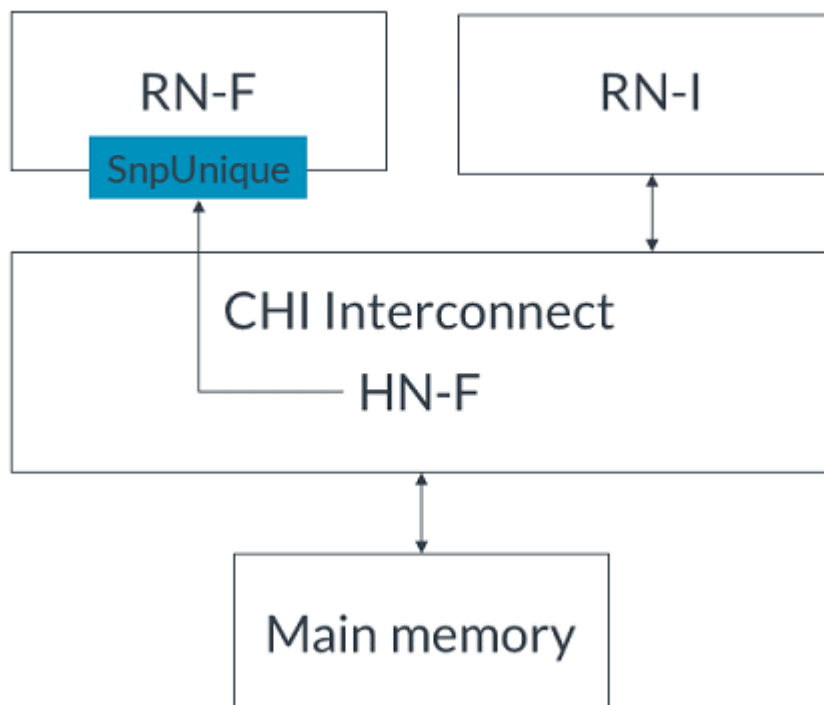
In the second example, a `ReadOnceMakeInvalid` transaction reads and invalidates the cache line in the RN-F, but rather than write the Dirty data to Main Memory, it discards the data.

The full transaction flow of this example is as follows:

1. The RN-I issues a `ReadOnceMakeInvalid` Transaction to the HN-F, as shown in the diagram:

Figure 7-3: The RN-I transaction

2. The HN-F sends a `snpUnique` Request to the RN-F, requesting the cache line, shown in the following diagram:

Figure 7-4: The HN-F request

3. The RN-F invalidates the cache line and sends the Dirty data to the HN-F.
4. The HN-F returns the data to the RN-I then discards the Dirty data.

**Note**

If you invalidate a Dirty line that an agent reads before it gets overwritten, the `ReadOnceMakeInvalid` Request can lead to loss of data. Only use this transaction when you know that you will not use this data again in the future.

8. Data Memory Transfer, Direct Cache Transfer, and PrefetchTgt

In CHI-A, read data and Snoop data both travelled through the Home Node before the initiating Request Node received it. Travelling through the Home Node increased access latency in these Requests. To reduce this latency, CHI-B adds the Direct Memory Transfer (DMT) and Direct Cache Transfer (DCT) mechanisms.

The following table summarizes the differences between CHI-A and CHI-B for data going from an SN or RN to an RN:

-	SN to RN	RN to RN
CHI-A	Read data from the SN must pass through the HN on the way back to the RN	Snoop data from an RN must pass through the HN on the way back to the RN
CHI-B	Direct Memory Transfer (DMT): SN data bypasses the HN and goes directly to the RN	Direct Cache Transfer (DCT): RN data bypasses the HN and goes directly to the RN

To support DMT and DCT operations, additional identifiers are added to the Request, Snoop, and Data flits. These additional fields specify the following information needed to correctly route the data and any required responses to the correct endpoint:

- The end target for the read data
- `TxnID` of the original Request
- The HN that issued the request to SN-F, or the snoop to RN-F

The HN still requires a `compAck` notification that the DMT or DCT completed.

The CHI-B issue also adds the Prefetch Target (`PrefetchTgt`) Transaction to reduce the access latency of memory accesses. `PrefetchTgt` Transactions are sent directly from an RN-F to an SN-F and do not require any data to be returned. The Memory controller can use this as a hint and buffer the data for the `PrefetchTgt` Request. The buffering provides quicker access times if a normal Request for that data is received while the data is in the buffer.

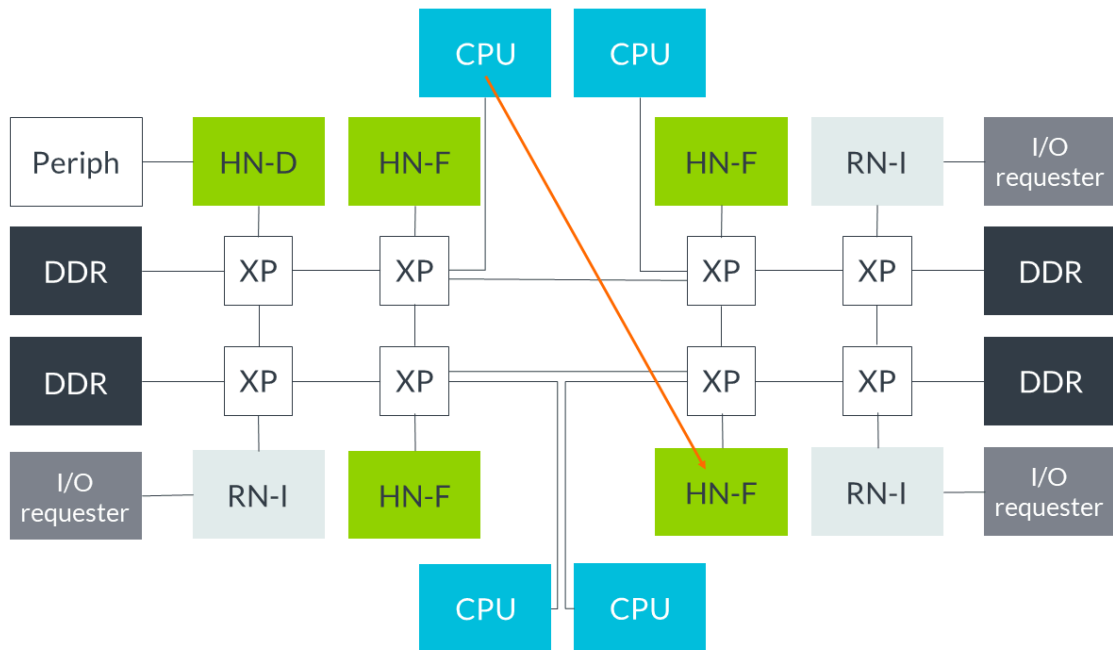
This section describes the transaction flows of DMT, DCT, and `PrefetchTgt`. This section also describes the additional identifier fields in each flit and uses examples that illustrate how the identifier fields are assigned in each message.

Direct Memory Transfer

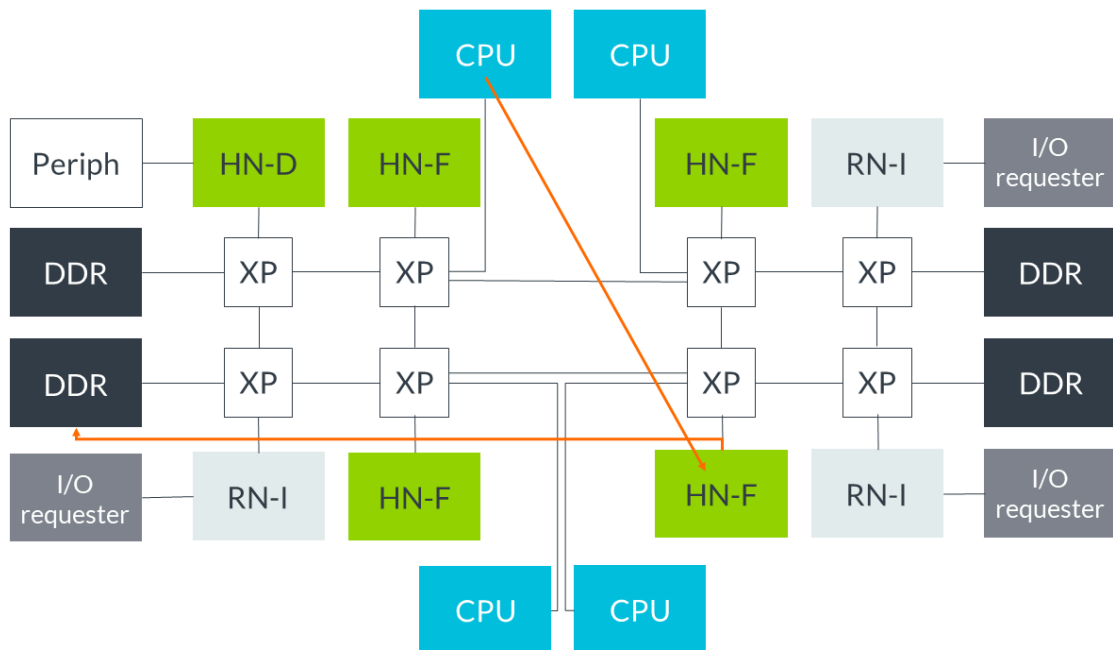
In the following examples, you can compare the path that read data takes for a Read Request with and without DMT.

For a read request without DMT, the transaction flow is as follows:

1. The CPU issues a Read request to the HN-F, as shown in the diagram:

Figure 8-1: The CPU issuing a read request

2. The **HN-F** has a cache-miss for the address and issues a Read Request to the Memory controller, as shown in the following diagram:

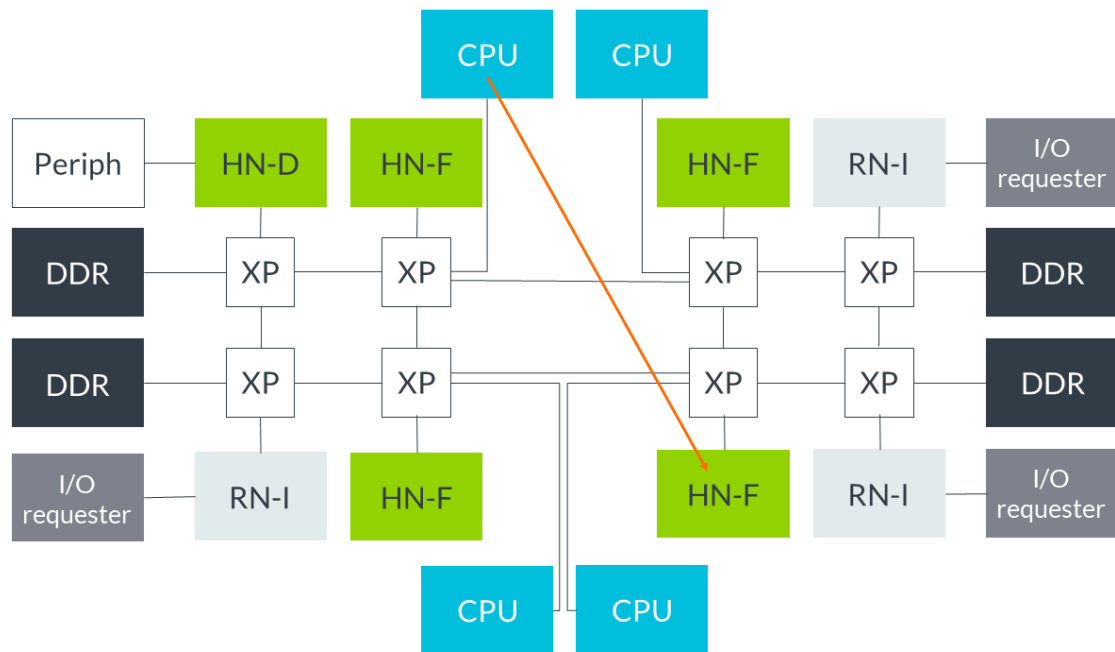
Figure 8-2: HN-F cache-miss

3. The Memory controller gets the data for the Read Request then sends the data back to the HN-F.
4. The HN-F returns the read data to the CPU that requested the cache line. The read data needed to return to the HN-F before reaching its destination.

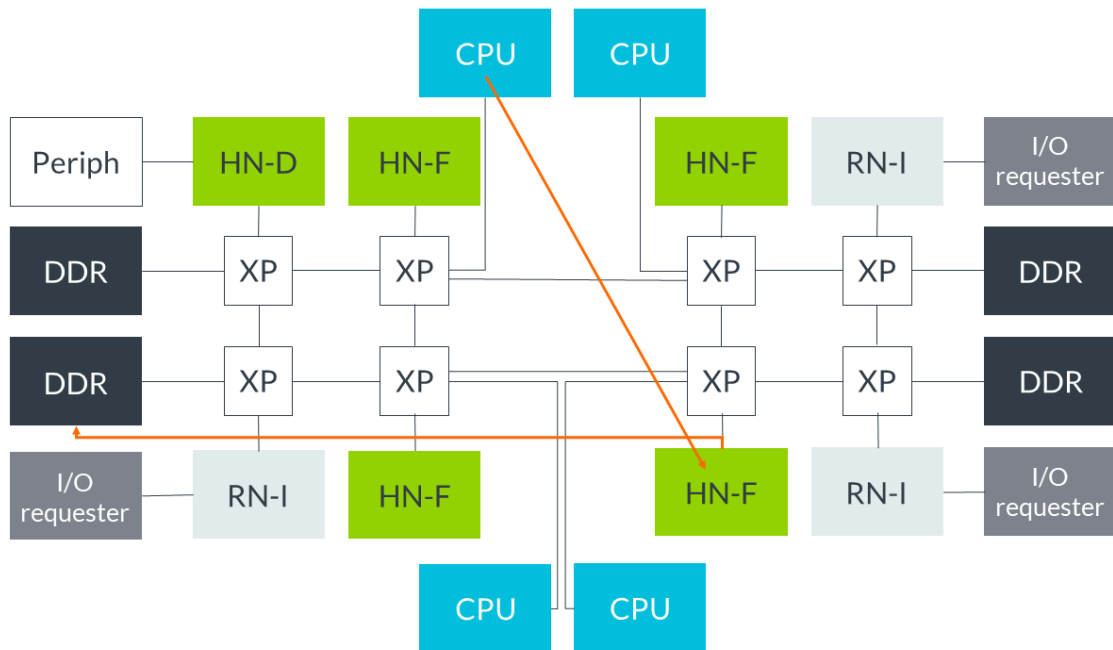
In the second example, DMT was utilized, and the transaction flow was modified as follows:

1. The CPU issues a Read Request to the HN-F, as shown in the diagram:

Figure 8-3: The CPU issuing a read request



2. The HN-F has a cache-miss for the address and issues a Read Request to the Memory Controller, as shown in the following diagram:

Figure 8-4: HN-F cache-miss

3. The Memory controller obtains the data for the Read Request
4. The Memory controller sends the data to the initiating CPU instead of the HN-F

With DMT, the read data bypasses the HN-F and goes directly to the CPU that issued the Read Request. Most Read Requests can use the DMT mechanism, including the implicit DataPull reads resulting from cache stashing operations.

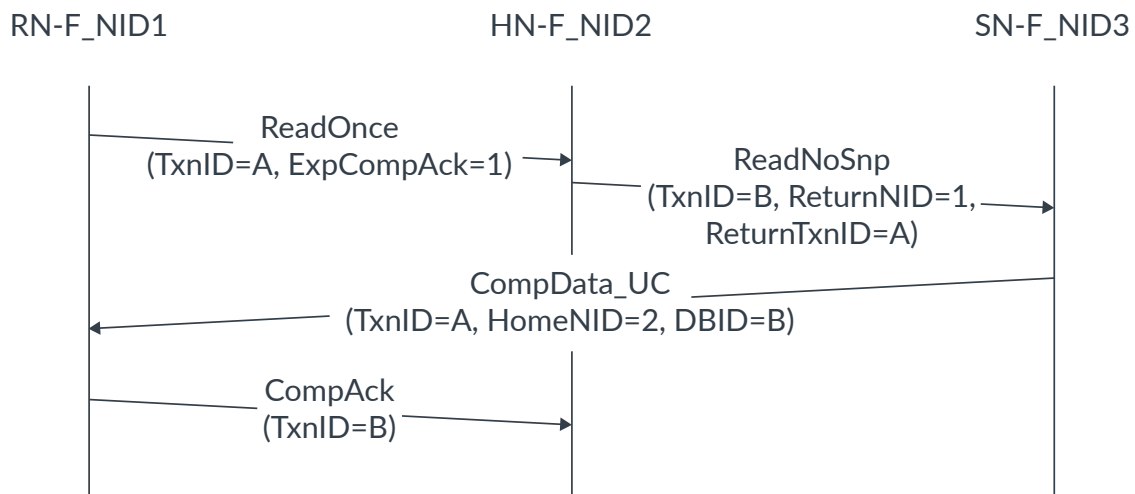
The Requests that cannot use DMT are:

- Exclusive Accesses
- ReadNoSnp Requests where `ExpCompAck = 0` and `Order != 0`
- ReadOnce Requests where `ExpCompAck = 0` and `Order != 0`

To support DMT, CHI contains the following identifier fields:

- The Request flit uses the Return Node ID (ReturnNID) and Return Transaction ID (ReturnTxnID) fields
- The Data flit uses the Home Node ID (HomeNID) field

The example in the following figure shows the timing of the DMT transaction flow, focusing on the Identifier field usage:

Figure 8-5: Arrows pointing to RN-F_NID1, HN-F_NID2, SN-F_NID3

This example system has:

- An RN-F with Node ID 1 (RN-F_NID1)
- An HN-F with Node ID 2 (HN-F_NID2)
- An SN-F with Node ID 3 (SN-F_NID3)

The DMT Transaction flow in the example uses the Identifier fields in the following ways:

- The RN-F sends a **ReadOnce** Request to the HN-F with **TxnID = A** and **ExpCompAck = 1**
- The HN-F does not have the requested data in its cache, so it issues a **ReadNoSnp** Request to the SN-F. The **ReadNoSnp** Request has:
 - **TxnID = B**
 - **ReturnNID = 1**. This indicates that the read data should be sent to the RN-F, which has Node ID 1
 - **ReturnTxnID = A**. This matches the **TxnID** from the original **ReadOnce** Request.
- When the SN-F is ready to return the read data, it sends a **CompData_UC** message with:
 - **TxnID = A**. This matches the value the SN-F received as **ReturnTxnID**
 - **HomeNID = 2**. This is the Node ID of the HN-F
 - **DBID = B**. This matches the **TxnID** of the **ReadNoSnp** sent by the HN-F
- The RN-F sends the **CompAck** message to the HN-F with **TxnID = B**. This matches the **DBID** field in the **CompData_UC** message.
- The HN-F receives the **CompAck** and can stop tracking the **ReadNoSnp** message it had sent to the SN-F

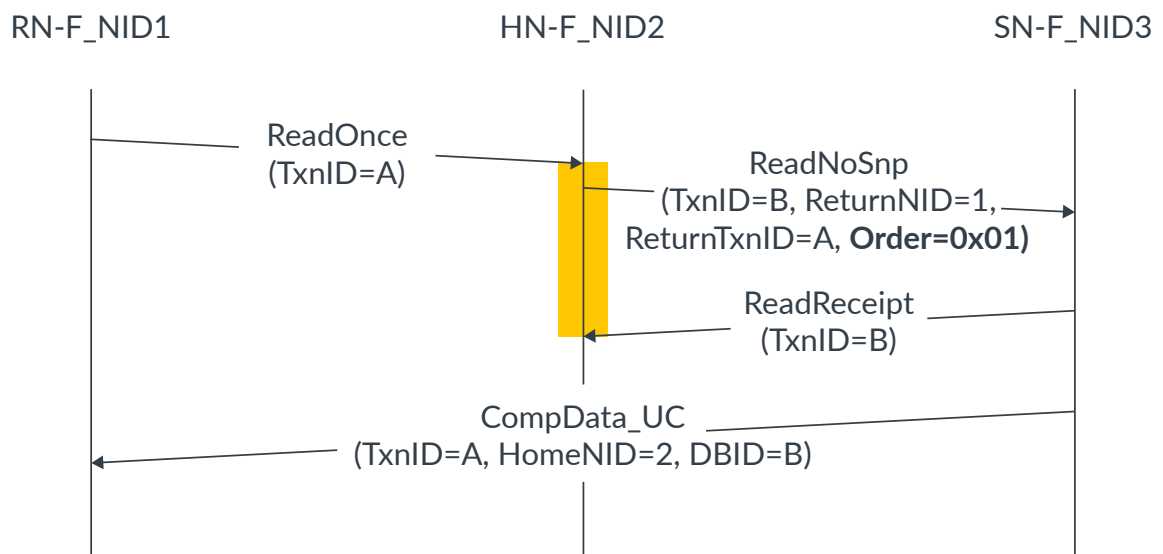
CHI-B contains an optimized DMT sequence for certain **ReadOnce** and **ReadNoSnp** transactions. The sequence adds a new requirement for SN-F nodes to recognize the value **0x1** in the Request

Order field and send a `ReadReceipt` response to the HN-F. This addition reduces the lifetime of a transaction at the HN-F node, potentially freeing resources.

CHI-A, in contrast, marks the value `0x1` as reserved in the Order field and does not require a `ReadReceipt` from the SN-F. The only nodes allowed to send `ReadReceipt` responses are HNs to RNs and SN-Is to HN-Is.

The following diagram shows an example of the optimized DMT sequence for a `ReadOnce` transaction, where the HN-F does not have the address requested in its cache:

Figure 8-6: RN-F_NID1, HN-F_NID2, and SN-F_NID3



In this example, the transaction flow is as follows:

1. The RN-F issues a `ReadOnce` Request to the HN-F with `TxnID = A`.
2. The HN-F issues a `ReadNoSnp` Request to the SN-F with:
 - `Order = 0x01`
 - `TxnID = B`
3. The `ReturnNID` field gets the Node ID of the RN-F.
4. The `ReturnTxnID` field gets the `TxnID` of the original `ReadOnce` Request.
5. The SN-F accepts the transaction.
6. The SN-F issues a `ReadReceipt` to the HN-F.
7. When the data is ready, the SN-F sends the read data to the RN-F using the original `TxnID`.

The HN-F receives the Read Receipt and immediately deallocates the Request. This deallocation reduces the lifetime of the transaction at the HN-F and freeing up resources. If this were CHI-A,

the HN-F would need to wait until it received a `CompAck` response from the RN-F to stop tracking the `ReadNoSnp` Transaction.

Prefetch Target

To enhance Direct Memory Transfer further, CHI-B provides the Prefetch Target (`PrefetchTgt`) Request to reduce memory access latency at the SN-F.

The `PrefetchTgt` message is a hint that is sent directly from an RN to an SN-F. This request does not require a response, so it is not tracked as an outstanding request by the RN. The SN-F can choose to ignore the request or fetch the data for the specified address.

If the SN-F decides to fetch the data, it buffers it until a normal Read Request for that address is received. The assumption is that a separate Read Transaction will come on the normal path through a Fully Coherent Home Node, HN-F, in the near future.

Having the Data Buffered at the SN-F reduces the latency of the memory access from a Read Transaction and hides any additional latency with local lookups in the HN-F system cache first.

Because a response is not needed, the `TxnID` field in `PrefetchTgt` is inapplicable, and CHI-B requires that it is set to 0 when the request is sent. For example, the RN issues the request to the SN-F. This is the only step required to complete the `PrefetchTgt` transaction, no additional flits are sent in either direction.

A `PrefetchTgt` Request can be sent so far in advance that the SN-F evicts the buffered data to make room for other Read Requests. To avoid congestion with `PrefetchTgt` Requests, CHI-B uses the `DataSource` field in the Data Flit to report the effectiveness of using `PrefetchTgt`. This field is set by the memory controller and indicates if the read data benefited from an earlier `PrefetchTgt` hint. The `DataSource` field possible values are:

- 0x6 to that the `PrefetchTgt` Request was useful
- 0x7 to indicate that the read data did not benefit from `PrefetchTgt` and was not useful

The RN can stop issuing `PrefetchTgt` Requests if enough are determined to be ineffective.

Usually, RNs only implement a RN System Address Map (RN SAM). This SAM targets HN-Fs and will have no awareness of SN-F Node IDs. To support the `PrefetchTgt` Transaction, RNs need an HN System Address Map, too. The HN SAM translates addresses to SN-F `TgtIDs`.

For example, the `PrefetchTgt` hint can optimize DMT. The CPU issues a `PrefetchTgt` Request before the DMT reads miss. Following the `PrefetchTgt` transaction, the DDR controller has the read data ready when it receives the Read Request. The full flow in the example is as follows:

1. The CPU issues a `PrefetchTgt` hint to the DDR controller.
2. The DDR controller accepts the hint and begins the process of retrieving the data
3. Two things happen in parallel:
 - The CPU issues a Read Request to the HN-F for the same address the `PrefetchTgt` was for.
 - The DDR controller starts receiving the read data and buffers it for a subsequent read.

4. The CPU issues a Read Request which is sent to the HN-F
5. The Read Request results in a cache miss at the HN-F.
6. The HN-F issues a Read Request to DDR memory.
7. Because the data has already been buffered at the DDR controller, the DDR immediately returns the read data to the CPU. By using the `PrefetchTgt` Request, the DMT Read Transaction has almost no latency in the DDR memory access.

Direct Cache Transfer

To reduce snoop hit latency, CHI-B uses the Direct Cache Transfer mechanism (DCT). DCT is similar to DMT for snoops and allows snoop data from an RN-F to bypass the HN-F and go directly to the original requester. This mechanism helps improve system performance when data needs to travel back and forth between requesters.

Use cases that benefit from DCT include semaphores and producer-consumer workloads.

For example, you can compare the path that read data takes for a Read Request with and without DCT. Without DCT the overall system level flow is as follows:

1. CPU A issues a Read Request to the HN-F.
2. The Request results in a cache miss at the HN-F.
3. The HN-F issues a Snoop to the CPU B, which holds the cache line.
4. CPU B returns the data for the cache line to the HN-F.
5. The HN-F returns the data to CPU A, which originally requested it.

Using the same initial transaction with the addition of DCT, the overall system level flow is optimized as follows:

1. CPU A issues a Read Request to the HN-F.
2. The Request results in a cache miss at the HN-F.
3. The HN-F issues a Snoop to CPU B, which holds the cache line.
4. CPU B bypasses the HN-F and returns the data directly to CPU A, which issued the Read Request.

By using DCT, the access latency of a snoop hit is reduced.

Forward Snoop Requests

To support DCT, one of the elements that CHI-B added was the Forwarding Snoop Request.

Forwarding Snoop Requests tell the snooped RN-F to send the snoop data directly to the original Requester. All snooppable reads, except Atomic Transactions and exclusive reads, can use DCT.

The Forwarding type snoops introduce new Identifier fields to the Snoop Flit as follows:

- Forward Node ID (`FwdNID`), which functions like `ReturnNID` in DMT. It holds the Node ID of the original Requester.

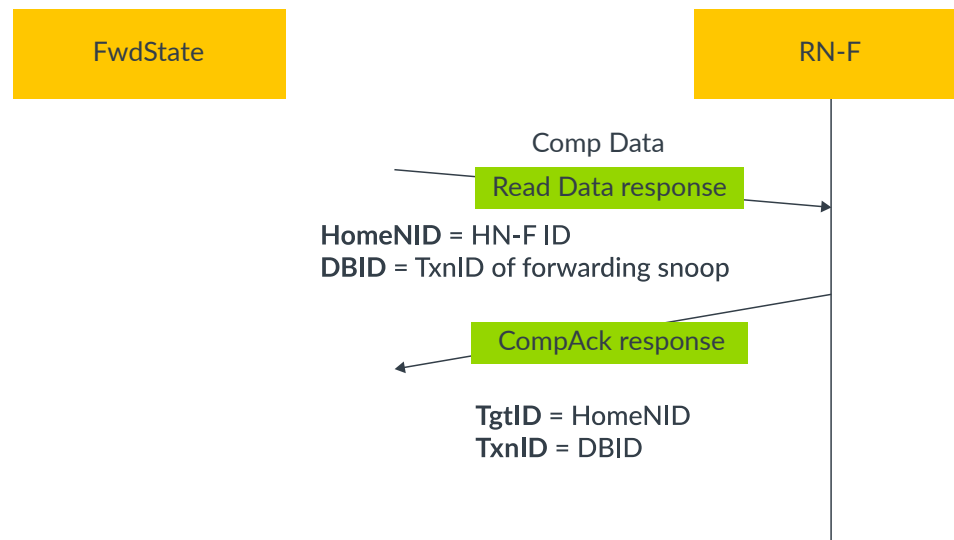
- Forward Transaction ID (`FwdTxnID`) which functions like `ReturnTxnID` in DMT. It holds the `TxnID` of the original Read Request.
- Return To Source (`RetToSrc`) instructs the RN-F to send the snoop data to the HN-F, in addition to the requesting RN-F. Sending the data to the RN-F allows future Requests to that address to hit in the HN-F cache and avoid generating additional snoops.

In responses to Forwarding Snoops, both the Response and Data Flits use a new Forward State (`FwdState`) field. This field tells HN-F what cache state was provided to the requesting RN-F for any local snoop filter tracking.

The cache state in the snooped RN-F, which is the result of the Forwarding Snoop, is provided to the Requesting RN-F in the `RESP` field as usual.

The original Requester receives the snoop data in a `CompData` message as a normal read data response, as is shown in the following diagram:

Figure 8-7: CompData message



In this diagram, the response includes the same `HomeNID` and `DBID` fields that DMT responses include:

- The `HomeNID` field has the Node ID of the HN-F that was bypassed
- The `DBID` field has the `TxnID` of the Forwarding Snoop

The RN-F then uses these fields as the `TgtID` and `TxnID` for the `CompAck` response to the HN-F.

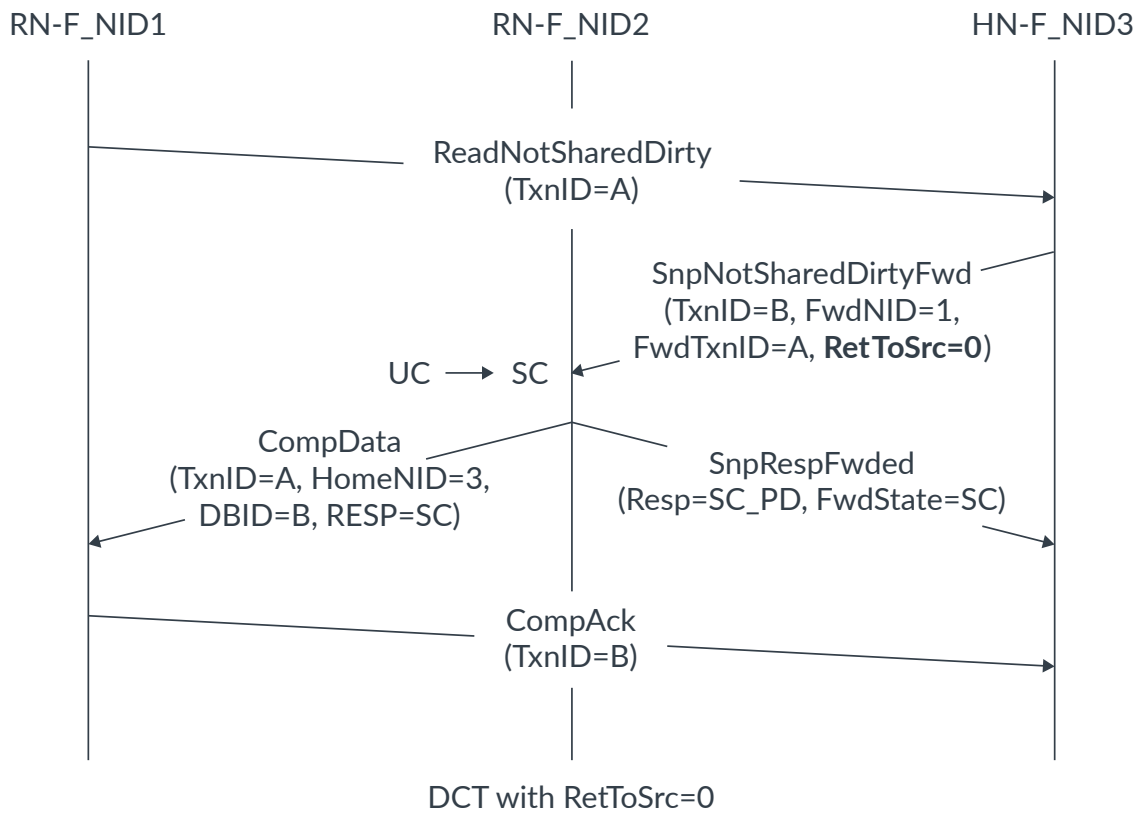
The following two examples show how the identifier fields are populated when `RetToSrc` is set to either 0 or 1.

The systems in these examples have:

- Two RN-Fs with Node IDs 1 and 2 (RN-F_NID1 and RN-F_NID2)
- RN-F2 holds the requested address in its cache in both instances
- One HN-F with Node ID 3

The following diagram shows the transaction flow of DCT when the `RetToSrc = 0`:

Figure 8-8: Transaction flow of DCT



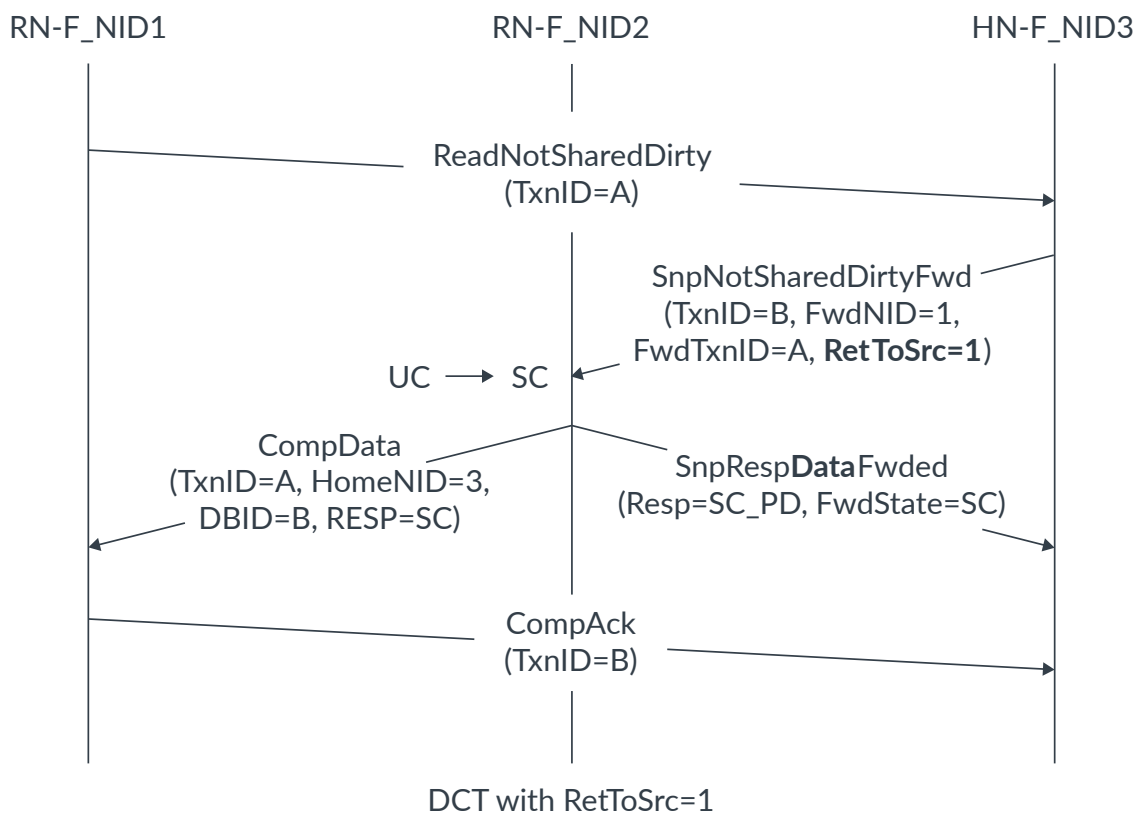
In this diagram, the transaction flow is as follows:

1. RN-F_NID1 issues a `ReadNotSharedDirty` Request to the HN-F. The Request has `TxnID = A`.
2. The Request results in a cache miss at the HN-F.
3. The HN-F issues an `SnpNotSharedDirtyFwd` Snoop to RN-F_NID2. The Snoop has:
 - `TxnID = B`
 - `FwdNID = 1`. This value matches the Node ID of RN-F_NID1. This indicates that it is the destination for the Snoop data.
 - `FwdTxnID = A`. This value matches the original `TxnID` of the Read Request.
 - `RetToSrc = 0`

4. Because `RetToSrc` in the Request is set to 0, `RN-F_NID2` responds to the `HN-F` with an `SnpRespFwded` message. There are two important fields in this response:
 - `RESP` shows that the cache line moved from the Unique Clean to the Shared Clean state
 - `FwdState` tells the `HN-F` what cache state was sent to the original Requester. In this example, that is Shared Clean.
5. `RN-F_NID2` sends a `CompData` message to `RN-F_NID1` with:
 - `TxnID` = A. This is the `FwdTxnID` value in the Snoop Request.
 - `HomeNID` = 3. This is the Node ID of the `HN-F`.
 - `DBID` = B. This is the `TxnID` of the Snoop Request
 - `RESP` = `sc` (Shared Clean). This shows the data is provided in the Shared Clean state, which matches the value in the `FwdState` field of the Snoop response.
6. `RN-F_NID1` sends a `CompAck` message to the `HN-F` with `TxnID` = B. This completes the `ReadNotSharedDirty` Request.

The second example in the following diagram shows the same `ReadNotSharedDirty` Request, but with `RetToSrc` = 1:

Figure 8-9: DCT Transaction when `RetToSrc` is 1



In this diagram, the transaction flow is as follows:

1. RN-F_NID1 issues a ReadNotSharedDirty Request to the HN-F with TxnID = A.
2. The Requests result in a cache miss at the HN-F.
3. The HN-F issues a SnpNotSharedDirtyFwd Snoop to RN-F_NID2. The Snoop has:
 - TxnID = B
 - FwdNID = 1
 - FwdTxnID = A
 - RetToSrc = 1
4. RN-F_NID2 forwards the cache line to RN-F_NID1 with a CompData message with the following fields:
 - TxnID sets to the value of FwdTxnID from the Snoop.
 - DBID sets to the value of the TxnID from the Snoop.
 - RESP shows that the cache line being returned can be cached in the Shared Clean state.
5. Because RetToSrc = 1, RN-F_NID2 sends the cache line to the HN-F. The cache line is sent in a SnpRespDataFwded message with FwdState = sc (Shared Clean) and RESP = sc_PD (Shared Clean Pass Dirty). This value in RESP tells the HN-F that the cache line is in the Shared Clean state at RN-F2 and that RN-F2 is passing the writeback responsibility for that cache line to the HN-F.
6. After receiving the Snoop data, RN-F_NID1 sends a CompAck response to the HN-F with TxnID = B.

9. Atomic operations

To support the atomic instructions added in the Armv8.1 architecture, CHI-B provides Atomic Transactions. An interconnect uses Atomic Transactions to transport an atomic operation and its operands from one device to another. Using atomics instead of exclusive access reduces the amount of time during which data is inaccessible to other agents.

Atomic Transactions can execute several atomic operations and can be performed internally or externally to the processor.

This section introduces the basic concepts of Atomic Transactions. A future guide will contain more detailed information about atomics.

An atomic operation is a read-modify-write sequence that is performed without interference from another requester. Like exclusive accesses in AXI, Atomic Transactions allow a requester to modify data in a particular region of memory, while ensuring that writes from other requestors do not corrupt the data.

In AXI3 and 4 and CHI-A, a requester fetches the data, performs the operation, and then writes the result back for the atomic access to complete. CHI-B contains the option to transport the atomic operation to the interconnect, which allows the operation to perform closer to where the data resides. This increases efficiency and reduces the time that data is made inaccessible to other requesters.

To perform the atomic operation, the target needs an arithmetic logic unit (ALU). That is, to use atomic operations an HN, SN or both need an ALU. Atomic Transactions support is optional from CHI-B, so HNs and SNs do not always have to have an ALU. The requester has a configuration pin, BROADCASTATOMIC, that can be used to stop the requester from generating Atomic Transactions if the downstream system does not support them.

The full Atomic Transaction structure is:

- The requester issues an Atomic Transaction to the interconnect
- The HN or SN has an ALU, so it performs the atomic operation
- Depending on the operation, the interconnect may return the original data of the address to the requester

10. RAS features

CHI-B adds Reliability, Availability, and Serviceability (RAS) features to support the Armv8 RAS specification. RAS features help error detection and system debug and are described in the following list:

- Data Poisoning and DataCheck indicate corrupted data
- The Trace Tag feature is used for profiling and debug

Data poisoning and DataCheck

A typical system can only detect multi-bit errors and cannot correct them. This is why multi-bit errors are often called Uncorrectable Errors. By contrast, many systems can correct single-bit errors.

The Arm RAS specification allows propagating Uncorrectable Errors from producer to consumer without immediately raising an exception. To allow passing corrupted data, CHI-B contains the RAS features of Data Poisoning and DataCheck. Both features indicate whether Data has been corrupted at some point in the system.

CHI-B allows interoperability between the `Poison`, `Datacheck`, and `RespErr` fields in the Data Flit. For more information on converting between these RAS features, see the [CHI Issue B Architecture Specification](#).

Marking data as Poisoned does not immediately indicate that an error has occurred, which allows the data to propagate through a system until the data is consumed. Deferring the error indication means that a system does not have to raise an exception every time that an Uncorrectable Error is detected. Instead, the Poison field can be allocated into a cache alongside the corresponding cache line. This allows the system to access and use the uncorrupted data.

Data poisoning operates on a 64-bit granularity, which means that the Poison field in the Data Flit sets one bit for every 64 bits of data to indicate that it has been corrupted. For example, a 256-bit Data field would have a Poison field that is 4 bits wide.

Data poisoning must be accurate for valid portions of the data. If a 64-bit piece of data is invalid, the poison for that piece will return a Don't Care value.

Corrupted Data is considered consumed when one of the following occurs:

- The data is used in a computation
- The data is propagated to a component that does not support data poisoning. Because this component cannot use the Poison field, it will stop tracking the poisoned data. To keep track of that data, the system must get an exception.

The DataCheck feature provides odd byte parity protection for the Data field. An implementation can test the `Datacheck` field for corrupted data at various points in the interconnect. DataCheck operates at an 8-bit granularity, so every bit in the `Datacheck` field corresponds to a byte in the Data field.

Trace Tag

CHI-B contains the `TraceTag` field to aid debugging and profiling. The width of the `TraceTag` is only 1 bit, and the field was added to every channel.

If the `TraceTag` field in a Flit is set, it indicates to the system that the Flit is marked for tracing purposes. All subsequent Flits in the transaction must also have `TraceTag` set. This includes all new transactions spawned from the original Request.

For example, a Read Request from an HN-F to an SN-F must also have the `TraceTag` field set, if it was spawned from a Request from an RN-F to an HN-F that had `TraceTag` set.

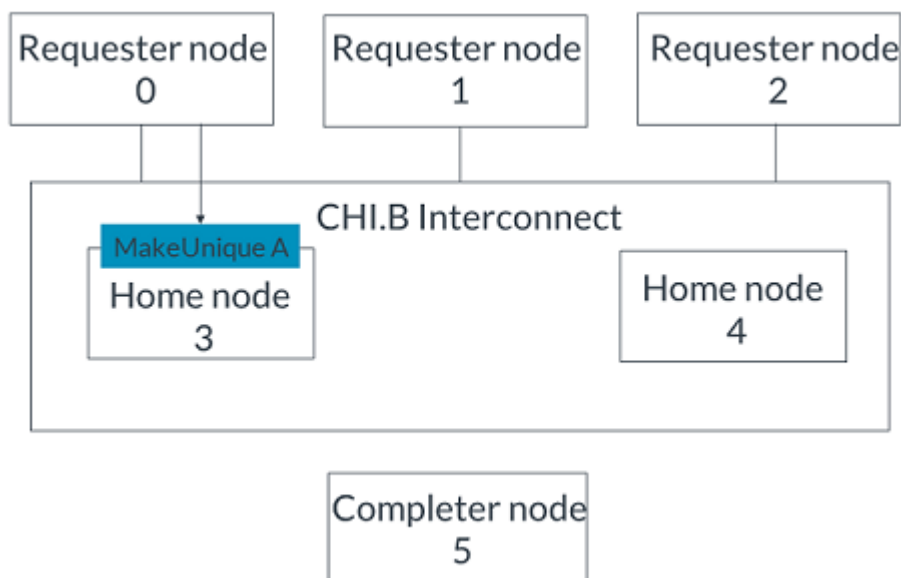
A Request Node can set `TraceTag` in an initial Request, or `TraceTag` can be set at intermediate points in the interconnect. For example, the watchpoint of an interconnect can be programmed to set the `TraceTag` for Requests to address A at the HN-F. This programming sets `TraceTag` for any Flits issued for address A at the HN-F, but the initial Request from an RN to the HN-F may not have `TraceTag` set.

For example, the initial `TraceTag` is set in the interconnect and then is set for subsequent Flits. The interconnect is programmed to trace MakeUnique Requests.

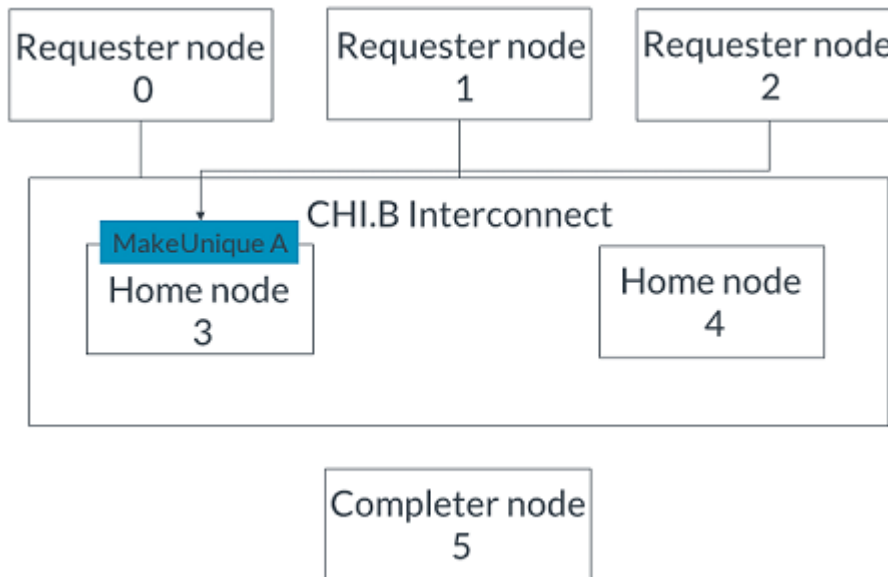
The transaction flow is as follows:

1. Request Node 0 issues a MakeUnique Request for address A at the Home Node, as shown in the following diagram:

Figure 10-1: The request node issuing a MakeUnique request



2. The interconnect sets `TraceTag` in the snoops and snoop responses that are generated by the MakeUnique Request.
3. Request Node 2 sends a ReadShared request for address A to Home node 3. The snoops generated from the ReadShared do not have `TraceTag` set. This step is shown in the following diagram:

Figure 10-2: The request node 2 request

4. The HN-F sends a completion to Request Node 0. The completion has TraceTag set.

In this example, both `MakeUnique` and `ReadShared` target address A, but the `TraceTag` field was only set after a `MakeUnique` Request was seen at Home Node 3. None of the Flits generated for the `ReadShared` Request were marked for tracing.

11. Other protocol changes and extensions

This section describes changes between CHI issues A to C and provides examples of some of the added features.

Changes from CHI-A to CHI-B

The following table describes additions that were made in CHI -B

CHI-B additions	Description
MESI protocol support	CHI-B adds support for the MESI coherency protocol. This supports RN-Fs that do not use the Owned or Shared Dirty cache state and simplified Snoop filters. This change adds the new opcodes <code>ReadNotSharedDirty</code> and <code>SnpNotSharedDirty</code> , which guarantee that data is not returned in the <code>SharedDirty</code> state. The SNP Flit also added the field <code>DoNotGoToSD</code> to ensure that a snooped RN-F does not leave a cache line in the <code>SharedDirty</code> state.
SharedClean state return	CHI-B adds the ability to return a cache line in the Shared Clean state. An HN-F can request a copy of a cache line in the Shared Clean state by using the <code>RetToSrc</code> field in non-forwarding type Snoops. If multiple RN-Fs hold the cache line, the HN-F sets the <code>RetToSrc</code> field for only one snooped RN-F. This encourages more Shared data to be present in the system cache, rather than only in the CPU caches. This can help reduce the latency that would be present when having to snoop an RN-F to get a copy of the cache line.
<code>WriteDataCancel</code> opcode to allow cancelling Write Requests	<p>This opcode is issued on the Data Flit, and only applies to three transactions: <code>WriteUniquePtl</code>, <code>WriteUniquePtlStash</code>, and <code>WriteNoSnpPtl</code>.</p> <p>To avoid deadlock scenarios, CHI-A allows one RN in the system to use the Streaming Ordered WriteUniques Optimization (WUO). With the <code>WriteDataCancel</code> opcode, a deadlock scenario can be broken if multiple RNs are using WUO. When <code>WriteDataCancel</code> is used:</p> <ul style="list-style-type: none"> All responses and Data Flits must still be sent to complete transactions. The <code>WriteDataCancel</code> message must be sent with the Data and BE fields zeroed out
CleanSharedPersist	<p>To clean a cache line to the Point of Persistence, the DC CVAP instruction was added in Armv8.2. For more information on the Point of Persistence, see the Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile.</p> <p>To support Persistent Memory Transactions, the <code>CleanSharedPersist</code> opcode was added. Execution of a DC CVAP instruction generates a <code>CleanSharedPersist</code> Transaction.</p> <p>Support for persistent memory is optional and is indicated by the configuration input pin <code>BROADCASTPERSIST</code>:</p> <ul style="list-style-type: none"> If <code>BROADCASTPERSIST</code> = 1, a <code>CleanSharedPersist</code> Transaction can be issued downstream. If <code>BROADCASTPERSIST</code> = 0, a <code>CleanSharedPersist</code> Transaction must be converted to a <code>CleanShared</code> Transaction by the requester
CMO propagation to the SN	Cache Maintenance Operations can be propagated from the HN to SN. This allows SNs to support caches downstream of the HN. Support for CMO propagation can be controlled by the optional <code>BROADCASTCACHEMAINTENANCE</code> , <code>BROADCASTINNER</code> , and <code>BROADCASTOUTER</code> signals. When these are all present and deasserted, <code>CleanShared</code> , <code>CleanInvalid</code> , and <code>MakeInvalid</code> transactions are not issued downstream.
DVM enhancements	The Virtual Machine Identifier field was extended from 8 bits to 16 bits. To support this enhancement, the <code>VMIDExt</code> field was added to the SNP Flit to transport the additional 8 bits (<code>VMID[15:8]</code>). This field is populated in the first part of the DVM Snoops and is set to 0 in the second part. <code>VMID[15:8]</code> is transported in the second part of the DVM Request, populating bits 63:56 of the <code>Data</code> field.

The following features are deprecated in CHI-B

- Barrier Transactions and the Inner and Outer shareability domains. All Barrier Transactions that are generated by DMB or DSB instructions must terminate within the cores. CHI-B interconnects do not support barriers, so an RN should not issue them externally.
- The Inner and Outer shareability domains. Requests are only marked as Snoopable or Non-Snoopable. Because only two types are supported, the SnpAttr field of the Request Flit was reduced from 2 bits to 1 bit.

Changes from CHI-B to CHI-C

The following table describes additions that were made in CHI -C

CHI-C additions	Description
CompAck Response sent earlier	RNs in CHI-C can issue a CompAck message after receiving the first Data Flit. Previously, an RN was required to wait for all read Data Flits to arrive before issuing a CompAck response.
Data Flit Opcode field width increase	The size of the opcode field of the Data Flit increased from 3 bits to 4 bits. This increases the Data Flit width by 1 bit over CHI-B. The wider opcode field is required to support the new Data message DataSepResp. Because the Data Flit increased by 1 bit, a CHI-C device is not directly compatible with a CHI-B device
Combined Write Data and CompAck	CHI-C added a new message, NCBWrDataCompAck, to allow a CompAck response to be sent with Write data as part of WriteUnique transactions.
Separate response and data for reads	<p>A Read Transaction can receive separate responses for completion and read data. To support this, two new messages were added:</p> <ul style="list-style-type: none"> The RespSepData message, which is sent on the RSP channel. This message indicates that a read has reached a Point of Serialization. The HN sends this message to the RN. The message decreases the lifetime of a Read Request at the HN, because the RN can immediately send a CompAck response after for an unordered read. A few restrictions apply for ordered reads from an RN to HN: <ul style="list-style-type: none"> An RN must wait for at least one DAT Flit before sending CompAck to the HN for an ordered read. The HN must not send a ReadReceipt response because the RespSepData message acts as the ReadReceipt. The DataSepResp message sent on the Data channel. This is a data-only message to send read data. This message can be sent by a HN or SN, depending on whether DMT is used.

Read Transaction examples

The read sequence uses the ReadNoSnpSep Request, which is designed to use with Data Memory Transfer (DMT) and is sent from an HN to an SN. When an SN sees this request, it knows to return the read data to the RN using the DataSepResp message.

The separate response and data read sequence can be used for most read types. The exceptions for this new sequence are

- Atomic Transactions
- Exclusive accesses
- ReadNoSnp Or any ReadOnce variant requiring ordering and no completion acknowledge

The following two examples demonstrate the read sequence with separate response and data messages.

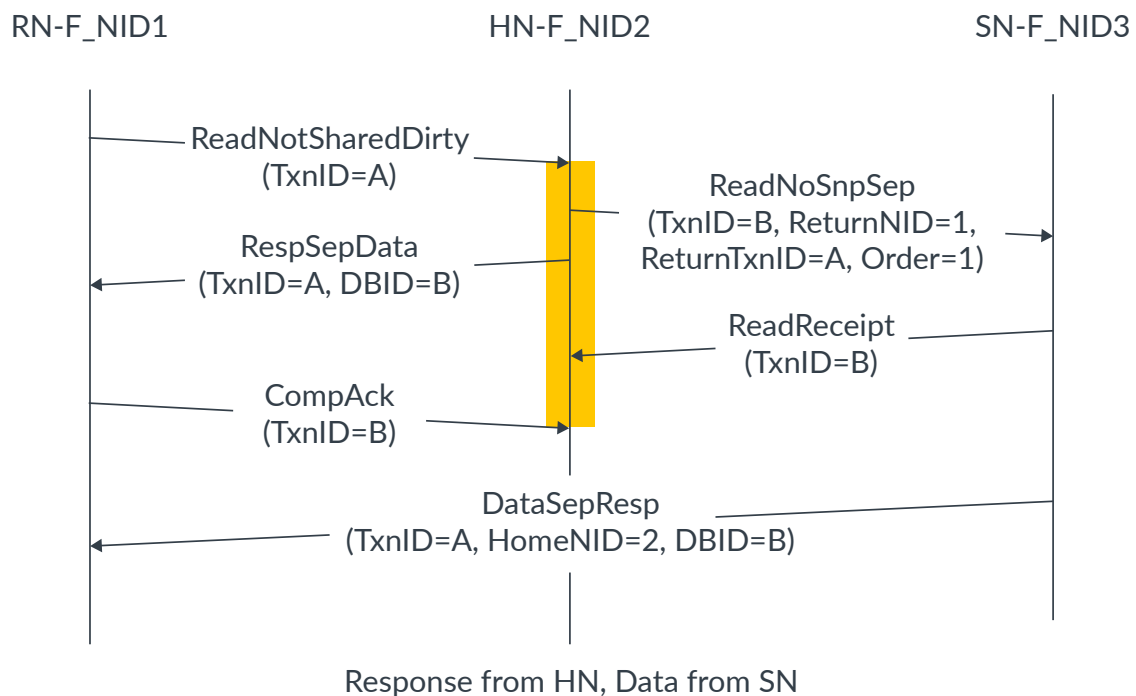
The first example shows the SN-F sending the data separately. The second example shows the HN-F doing the same

In both cases, the system is made of

- One RN-F, RN-F_NID1
- One HN-F, HN-F_NID2
- One SN-F, SN-F_NID3

The following diagram shows the full transaction flow when the response is from the HN, and the data is from the SN:

Figure 11-1: Full transaction flow when the response is from the HN and the data is from the SN



In this diagram, the transaction flow is as follows

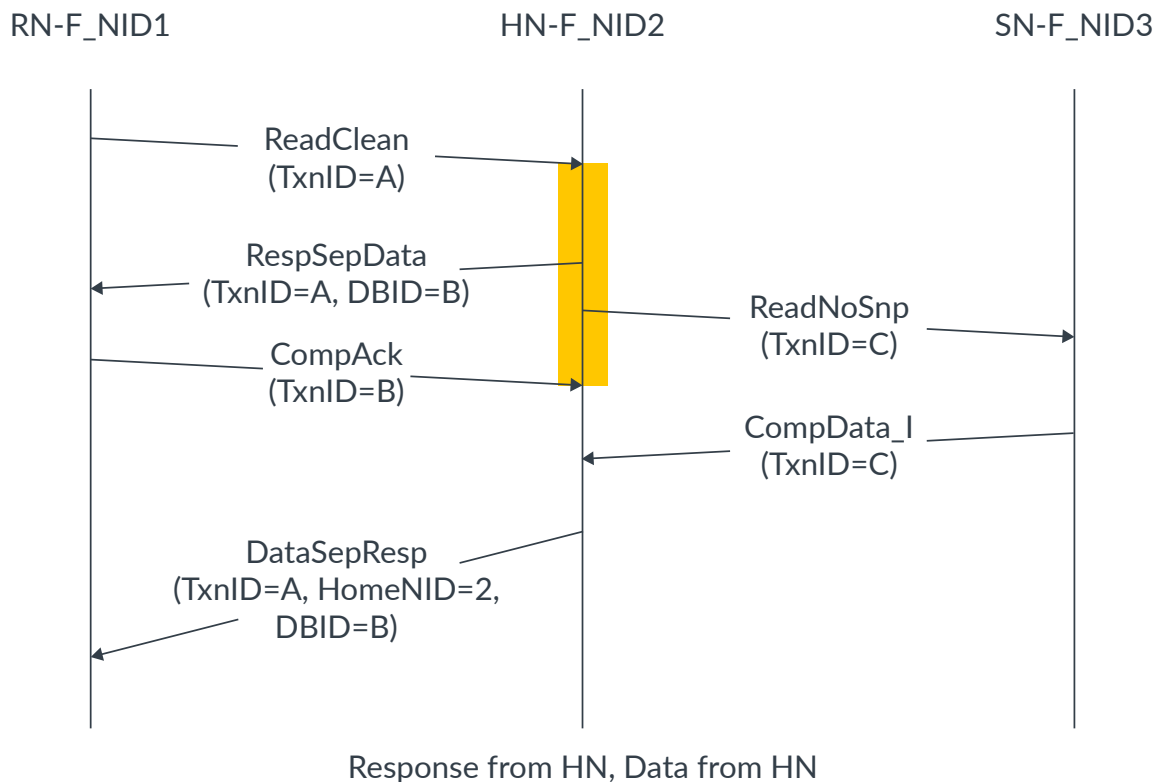
1. The RN-F issues a `ReadNotSharedDirty` Request to the HN-F with `txnID = A`.
2. The HN-F issues a `ReadNoSnpSep` Request to the SN-F, indicating that it should not send a `compData` response to the RN-F. This means that SN-F needs to return the read data using the `DataSepResp` message, and the `Order` field set to 1. This is similar to the Optimized DMT sequence.
3. The HN-F serializes the Request.
4. The HN-F issues a `RespSepData` message to the RN-F. The `txnID` matches the `txnID` of the original Request, and the `DBID` field matches the `txnID` of the `ReadNoSnpSep`.

5. The SN-F acknowledges the Request and responds with a `ReadReceipt` message to the HN-F.
6. The RN-F receives the `RespSepData`.
7. The RN-F issues a `CompAck` message to the HN-F without waiting for the Read Data.
8. When the data is available, the SN-F sends the Read Data to the RN-F with a `DataSepResp` message. The `TxnID`, `HomeNID`, and `DBID` fields of the `DataSepResp` message are the same as those for a DMT sequence.

In previous issues of CHI, an RN-F had to wait to send `CompAck` until it had received the data for a `ReadNotSharedDirty` Request. By using the separate response and data sequence, the lifetime of the transaction at the HN-F was significantly reduced

In this second example, the read data is returned by the HN-F through a `DataSepResp` message. The following diagram shows the full transaction flow when the response and data are both from the HN:

Figure 11-2: Full transaction flow when the response and data are both from the HN



In this diagram, the transaction flow is as follows

1. The RN-F issues a `ReadClean` Request to the HN-F.
2. The HN-F responds with a `RespSepData` message to RN-F.

3. Unlike the previous example, the HN-F sends a normal `ReadNoSnp` to the SN-F.
4. The RN-F receives the `RespSepData` response.
5. The RN-F sends the `CompAck` message to the HNF.
6. The SN-F sends the data to the HN-F using a `CompData` message.
7. The HN-F sends the data to the RN-F using the `DataSepResp` message. The fields values are:
 - `TxnID` matches the original Request.
 - `HomeNID` is 2, which is the Node ID of the HN-F.
 - The `DBID` field is B.

Like the first example, the HN-F was able to stop tracking the `ReadClean` Request after receiving the `CompAck` message. With the Separate response and data sequence, the HN-F received the `CompAck` message before the RN-F received the data

Combined write data and CompAck examples

CHI-C provides a message that combines a `CompAck` response and write data called `NCBWrDataCompAck`, where NCBW stands for Non Copy Back Write

This new message can be used for:

- Any `WriteUnique` variant
- Streaming Ordered `WriteUnique`

Because `NCBWrDataCompAck` transports write Data, it has to be sent on the Data Channel

Before sending the data, an RN has to wait for:

- A `DBIDResp` message
- A `Comp` response

The `Comp` and `DBIDResp` messages can be sent as two separate responses, or as a combined `CompDBIDResp`

Let's look at two examples of how `NCBWrDataCompAck` works. The first example uses the combined `CompDBIDResp` message, and the second example uses separate messages. The example system has one RN-F and one HN-F

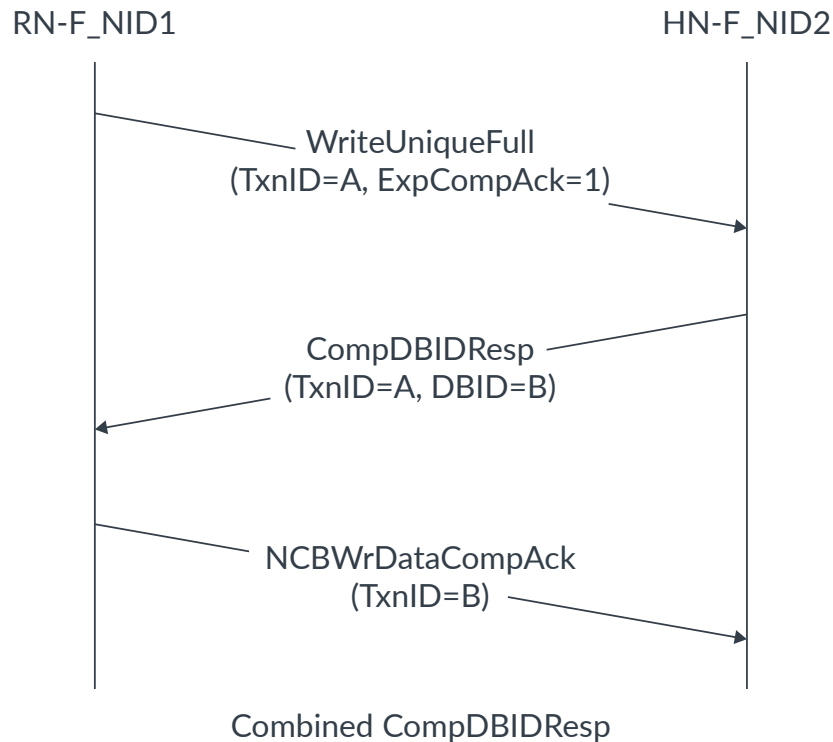
This example uses the combined `CompDBIDResp`:

- The RN-F sends a `WriteUniqueFull` Request to the HN-F with the values:
 - `TxnID` = A
 - `ExpCompAck` = 1
- The HN-F sends `CompDBIDResp` with:
 - The same `TxnID` as the original Request.
 - `DBID` = B

- The RN-F sends the combined `compAck` with Write data in a `NCBWrDataCompAck` message.

The following diagram shows the combined `CompDBIDResp` transaction flow:

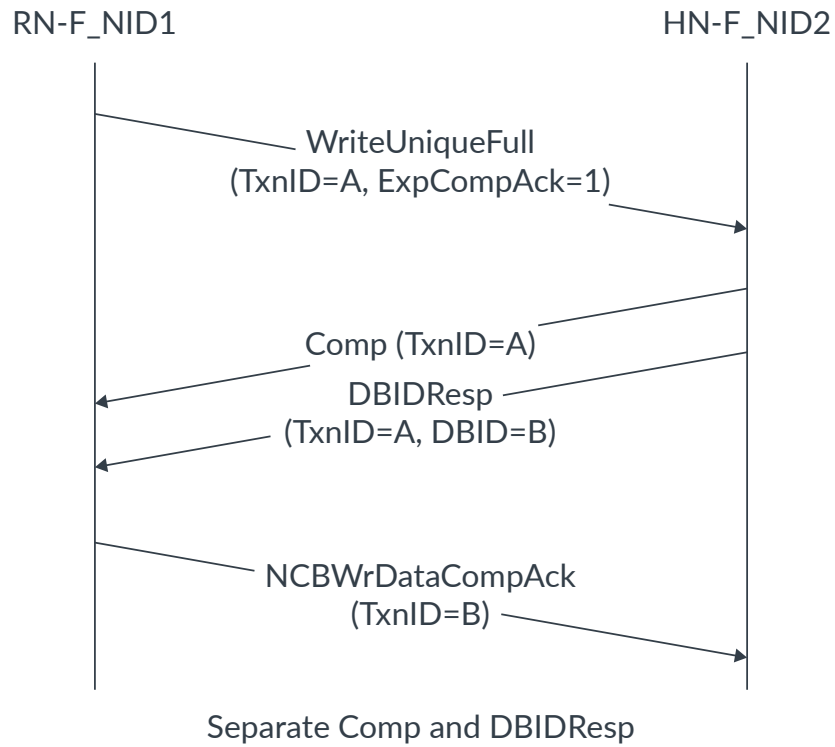
Figure 11-3: Combined CompDBIDResp transaction flow



In the second example, the system remains the same, but the HN-F sends separate `Comp` and `DBIDResp` messages

- The RN-F issues a `WriteUniqueFull` Request to the HN-F with the values:
 - `TxnID = A`
 - `ExpCompAck = 1`
- The HN-F performs two actions:
 - Responds with a `comp` message matching the `TxnID` of the `WriteUniqueFull`
 - Sends a `DBIDResp` response indicating it can receive the write data. The `TxnID` matches the `TxnID` of the Write Request, and `DBID = B`
- The RN-F receives both messages.
- The RN-F sends the `NCBWrDataCompAck` message with `TxnID = B`.

The following diagram shows the separate `comp` and `DBIDResp` transaction flow

Figure 11-4: Separate Comp and DBIDResp

12. Related information

Here are some resources related to material in this guide:

- [AMBA 5 CHI Architecture Specification](#)
- [AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite](#)
- [Arm Community](#)

13. Next steps

This guide introduced the fundamentals of the original AMBA CHI-A protocol and further detailed the changes that were made in the CHI-B and CHI-C issues. It defined the concepts of requester and completer, and the different node types. The guide has shown how these nodes can be connected up in a system to support different topologies.

In describing some of the transaction types and their associated transaction flows, we have demonstrated how information flows around a CHI-based system. By introducing concepts like Cache Stashing and I/O deallocation, we have shown how the flow of information around the system can be optimized to increase overall performance. Additionally, by describing the more recent RAS features, we have shown how CHI can be used to improve error detection and system debug.

This knowledge will be useful as you learn more about the [AMBA CHI protocol specification](#). You can put your knowledge into action to develop interfaces that implement the AMBA CHI protocol.

To learn more about the AMBA CHI protocol, you can access subscription-based technical training at [Introduction to the AMBA CHI protocol training](#).