Atomic transactions in AMBA CHI

Non-Confidential

Issue 01

102714

Atomic transactions in AMBA CHI

Release information

Document history

Issue	Date	Confidentiality	Change
01	February 2, 2022	Non-confidential	First release

Contents

1 Overview

This guide describes the atomic transactions that Arm added into the CHI-B specification. These transactions are required to enable support for the atomic instructions added in the Armv8.1 architecture.

The guide introduces the new transactions, potential subopcodes, and the fundamentals of atomic operations. We compare the difference between using these newer transactions and older transaction counterparts. This guide also describes the additional functionality and the interconnect or endpoint device to support atomic transactions.

1.1 Before you begin

If you are not familiar with CHI, read Introducing the AMBA Coherent Hub Interface.

2 Atomic fundamentals

An atomic operation is a read-modify-write sequence 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. These transactions also ensure that writes from other Requestors do not corrupt the data. Atomic transactions can execute several atomic operations and be performed internally or externally of the processor.

In AXI3 and 4 and CHI-A, a Requester fetches the data, performs the operation, then writes the result back for the atomic access to complete. CHI-B enables the option to transport the atomic operation to the interconnect or Subordinate Node (SN). This increased efficiency reduces the time that data is made inaccessible to other Requesters.

To perform the atomic operation, the target uses an Arithmetic Logic Unit (ALU). To use atomic operations, a Home Node (HN) and SN require an ALU. Atomic transaction support is optional in CHI-B and later, so HNs and SNs do not always have an ALU.

The full atomic transaction structure is as follows:

- 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 can return the original data of the address to the Requester.

Atomic operations performed internally by a fully coherent Request Node (RN-F) are called near-atomics. Atomic operations that are performed externally to the Requester are called far-atomics.

An RN-F can perform a near-atomic operation within its cache memory system.

Near-atomics can occur for the following reasons:

- The RN-F holds the data in its cache in a Unique state, so external transactions are not required.
- The interconnect does not support atomic transactions, so atomics must be suppressed at the Requester.

When an interconnect supports atomic transactions, Request Nodes (RNs) can issue far-atomics downstream.

Far-atomics are performed closer to where the data resides, either in the interconnect or at the Subordinate holding the data.

A Request Node can issue far-atomics for the following reasons:

- The Request Node holds the line in a Shared state for the rest of the Coherent Nodes.
- The Request Node does not hold the cache line.
- The memory location is a non-coherent memory type.

3 Transaction types

This section describes the following atomic transaction types added in CHI-B:

- AtomicStore
- AtomicLoad
- AtomicSwap
- AtomicCompare

The system in each example has a Requester, memory unit, and an ALU.

The data in the series of diagrams in this section is represented by:

- AddrData for the current data in memory
- TxnData for the data sent with the atomic request
- Solid arrows for data that is always transmitted
- Dotted arrows for data that is sent only when certain conditions are met.

3.1 AtomicStore

The following diagram shows how an AtomicStore progresses in a system that fully supports atomics:

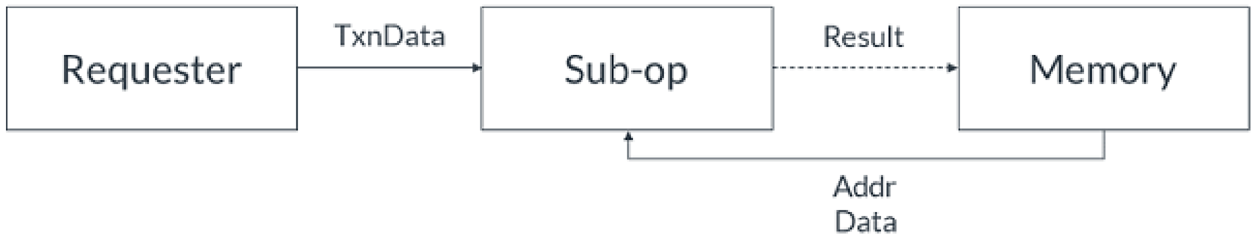

Figure 1: The AtomicStore transaction

When the ALU receives the AtomicStore transaction, it executes the suboperation that is indicated in the subopcode of the transaction. The ALU uses the TxnData and AddrData in the transaction as operands. When the operation is complete, the ALU stores the result to memory.

There are eight subopcodes that an AtomicStore can perform. These subopcodes are reviewed in Subopcodes for AtomicStore and AtomicLoad.

Some of the suboperations are conditional, so an AtomicStore transaction does not always need to write new data to memory.

3.2 AtomicLoad

The following diagram shows how an AtomicLoad progresses in a system that fully supports atomics:

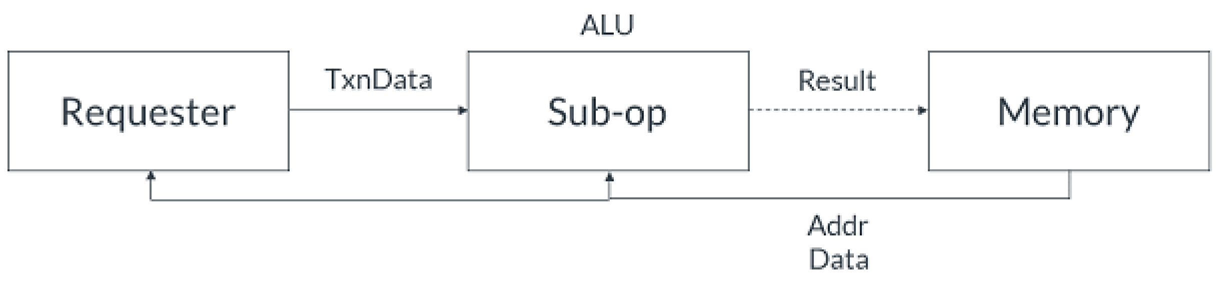

Figure 2: The AtomicLoad transaction

AtomicLoad is similar to AtomicStore, because it can execute eight subopcodes and updates memory if the condition passes. The subopcodes are reviewed in the Subopcodes for AtomicStore and AtomicLoad.

The difference between AtomicLoad and AtomicStore is that AtomicLoad returns the original AddrData value to the Requester.

3.3 AtomicSwap

The following diagram shows how an AtomicSwap progresses in a system that fully supports atomics:

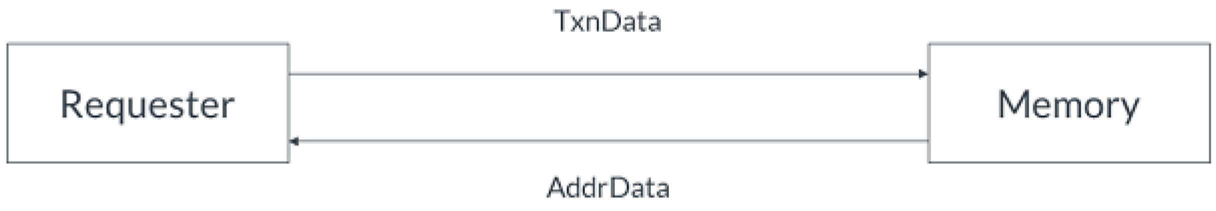

Figure 3: The AtomicSwap operation

AtomicSwap performs unconditionality and does not perform suboperations. AtomicSwap swaps the TxnData with the data at the addressed location. The new data is written to memory, and the original data is returned to the Requester.

3.4 AtomicCompare

The AtomicCompare transaction is different from the previous atomic transactions because it sends two data values with the following operations:

- SwapValue
- CompareValue

The following diagram shows how an AtomicCompare progresses in a system that fully supports atomics:

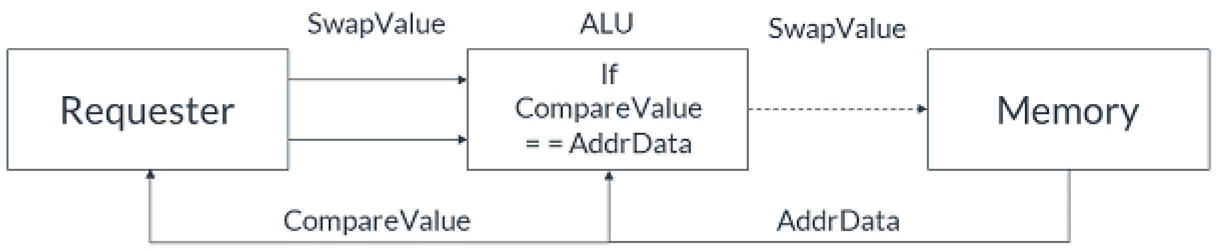

Figure 4: The AtomicCompare transaction

AtomicCompare works as follows:

- The ALU takes the current data and the CompareValue and checks if they are equal.
- The transaction performs one of the following actions:
    - If the compared values are equal, the memory location is updated with the SwapValue.
    - If the compared values are not equal, then memory is not updated.
- The original data at that memory location (AddrData) is always returned to the Requester, regardless of the comparison result.

3.5 Subopcodes for AtomicStore and AtomicLoad

There are eight subopcodes that an AtomicStore and AtomicLoad can perform. The following table summarizes these subopcodes, indicating the condition in which the value in memory is updated, and any corresponding update value:

Subopcode	Update AddrData with	Condition
ADD	TxnData + AddrData	Unconditional
CLR	[Bitwise] AddrData & ~TxnData	Unconditional
EOR	[Bitwise] AddrData ^ TxnData	Unconditional
SET	[Bitwise] AddrData | TxnData	Unconditional
SMAX	TxnData	(signed) TxnData > (signed) AddrData
SMIN	TxnData	(signed) TxnData < (signed) AddrData
UMAX	TxnData	(unsigned) TxnData > (unsigned) AddrData
UMIN	TxnData	(unsigned) TxnData < (unsigned) AddrData

Table 1: Subopcodes for AtomicStore and AtomicLoad

3.6 Result checking

The responses for an atomic transaction do not indicate whether a conditional atomic operation successfully occurred. If a Requester wants to know if an atomic transaction updated a memory location, the Requester performs a normal read of the address after the atomic operation completes. Alternatively, it can use the returned data to calculate whether the condition passed.

4 Atomics in action

This section tells you about the transaction flow for atomics in a transaction.

The following example shows the flow of an AtomicLoad transaction with an ADD suboperation (LDADD). The transaction is LDADD, and the data sent with the transaction is TxnData.

The current data in memory is called AddrData.

The full transaction flow is as follows:

1. The requester issues an AtomicLoad transaction to the memory controller with TxnData = 1. The suboperation for the transaction is an ADD operation. The following diagram shows this transaction:

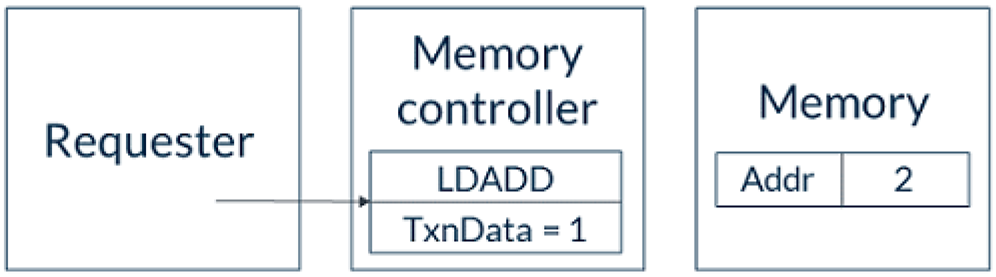

Figure 5: AtomicLoad transaction

2. The memory controller reads AddrData from the address specified in the atomic request. In this example, the value read from memory is 2 as shown in the diagram:

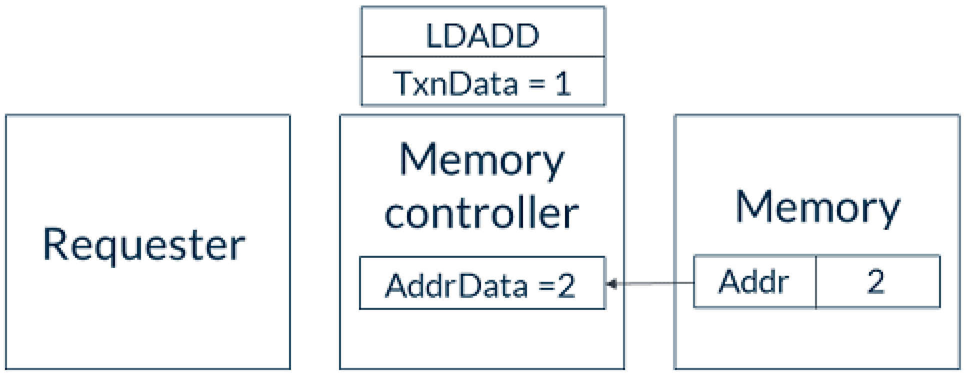

Figure 6: Memory reads AddrData

3. The memory controller computes the sum of TxnData and AddrData. This operation leads to a result of 3.
4. The operation result 3 is written to memory.

5. The memory controller returns the original AddrData value to the Requester, completing the AtomicLoad transaction, as shown in the following diagram:

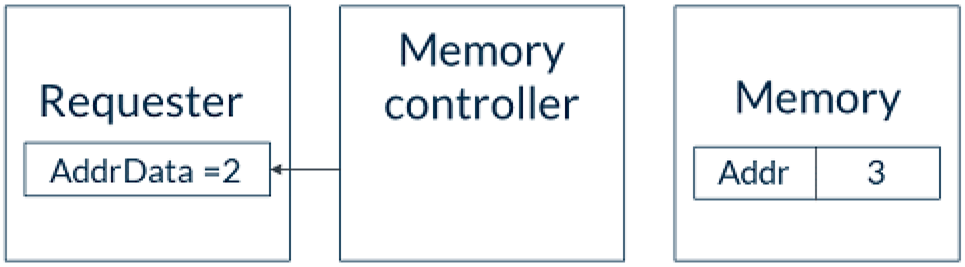

Figure 7: Memory returns the AddrData value

5 Address and data alignment requirements

This section of the guide describes how the data alignment works for the AtomicCompare transaction, using four different examples. Atomic operations have the following requirements:

- The size field must be set exactly to the size of the outbound data.
- The address in the atomic request must be aligned to the size of the data:
    - For the AtomicStore, AtomicLoad, and AtomicSwap transactions, the address in the request is always aligned to the size of the outbound data.
    - The AtomicCompare transaction issues two outbound data values, the CompareValue and the SwapValue. These values make the outbound data twice the size of the inbound data.
      To correctly align the address, the AtomicCompare request aligns the address to the size of the inbound data instead of the outbound data.
- For AtomicCompare requests:
    - The outbound data must be aligned to the data size.
    - The position of the Compare and Swap values varies depending on the address. The address of the transaction indicates the position of the CompareValue, and the SwapValue populates the remaining bytes to align to the size.

Let's look at how the data alignment works for the AtomicCompare transaction. The following diagram shows all four examples. The CompareValue is labeled C and the SwapValue is labeled S:

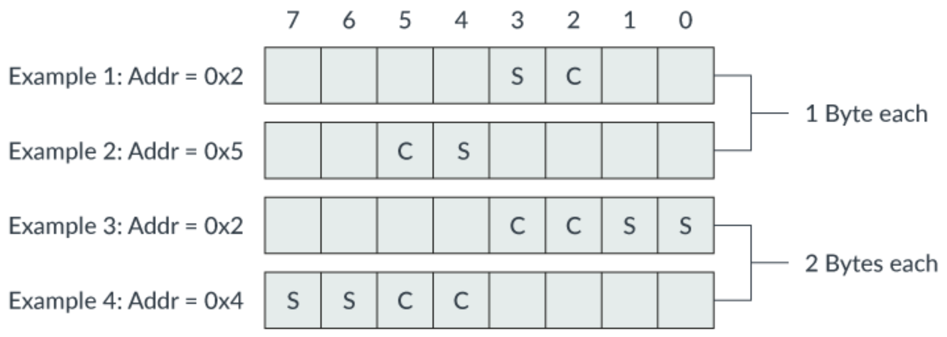

Figure 8: Four examples of AtomicCompare data alignment

The first two examples in the diagram show the inbound data is one byte.

In example 1, the address of the AtomicCompare request is 0x2, and the size of the inbound data is one byte. The width of the data bus is 8 bytes, which means that the Compare and Swap values require one byte each:

Because the address is 0x2, byte 2 in the bus carries the CompareValue.

Byte 2 of the bus is also aligned to the size of the outbound data, so byte 3 is used to carry the SwapValue.

In example 2, the size of the inbound data is one byte, but the address is 0x5 instead of 0x2. This transaction allows for the address to indicate the location of the CompareValue, and for both values to align to the request size. The address does not align to the outbound data size, so the positions of CompareValue and SwapValue change as follows:

CompareValue is sent on byte 5 of the bus, to match the requested address

SwapValue is sent on byte 4

Example 3 and examples 4 show the scenario the inbound data size as 2 bytes:

- Example 3 shows the alignment for an AtomicCompare for address 0x2.The address and the outbound data size, which is 4 bytes, do not align. In this example, the SwapValue gets positioned in the first 2 bytes of the bus. The CompareValue is positioned in bytes 2 and 3 to align with the address of the request.
- Example 4 shows the position of the Compare and Swap values when the data aligns to the 4 bytes of data being transmitted. The CompareValue is positioned at bytes 4 and 5 to align with the address, and the SwapValue is sent on bytes 6 and 7.

6 Memory types and Snoop scenarios

Atomic operations can be performed to any type of Device Memory or Normal Memory, but not all regions of memory are required to support atomic transactions.

Atomic requests can signal request Order and Endpoint Order in the transaction.

If an atomic request targets a snoopable address, the preferred snoop type is SnpUnique. This snoop type invalidates existing cache copies in case new bytes get written to that cache line.

The first example shows an atomic request for which the corresponding snoop hits, and the memory update condition for the subopcode passes.

The order of events is as follows:

1. The RN-I issues an AtomicStore transaction with the unsigned MAX subopcode to the HN-F, as shown in the following diagram:

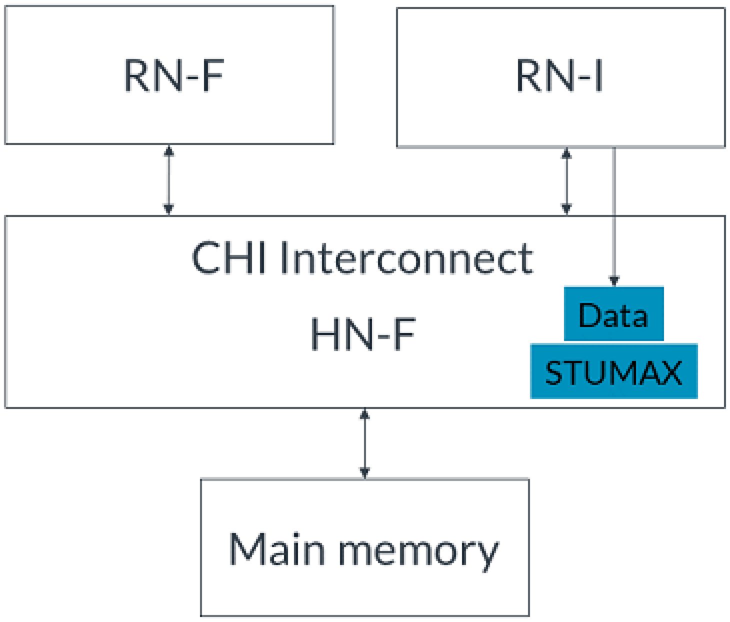

Figure 9: RN-I issues an AtomicStore transaction

2. The HN-F generates a snoop to the RN-F, as shown in the diagram:

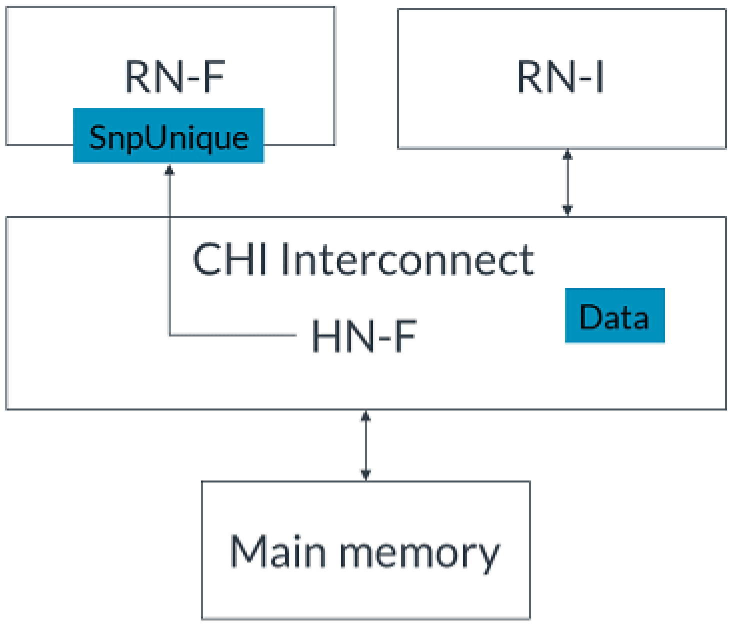

Figure 10: HN-F generates a snoop to the RN-F

3. The snoop results in a hit, so the RN-F invalidates the cache line and sends the dirty data to the HN-F.
4. The MAX function condition passes, and the HN-F overwrites the relevant bytes of the dirty line with the data from the atomic request. Atomic updates are less than 64 bytes in size, so the atomic operation modifies only part of the cache line.
5. The HN-F updates the memory with the new data.

The second example shows an atomic request for which the corresponding snoop hits. In this example, this time the memory update condition for the subopcode fails.

The order of events is as follows:

1. The RN-I issues an AtomicStore transaction with the unsigned MAX subopcode to the HN-F.
2. The HN-F generates a snoop to the RN-F.
3. The snoop results in a hit.
4. The RN-F invalidates its copy of the line and sends the dirty data to the HN-F.
5. The MAX function condition fails in the HN-F, and the cache line is not updated with the data from the request.
6. The HN-F updates the memory with the dirty data from the RN-F snoop, but the atomic operation does not modify the data following this update. Because the cache line was invalidated at the RN-F, the dirty data was written to memory to avoid discarding it.

7 Comparison of exclusive and atomic transactions

This section of the guide uses two examples to compare the performance of atomic operations with exclusives (CHI.A) and far-atomics (CHI-B).

7.1 Atomic add using exclusive transactions

This example shows an atomic add operation using exclusive accesses, which results in a near-atomic. The counter to increment is a shareable memory location that is held in the cache of another CPU.

The flow of the atomic add operation is as follows:

1. The CPU executes a load exclusive instruction that generates a ReadShared request to the HN-F, as shown in the diagram:

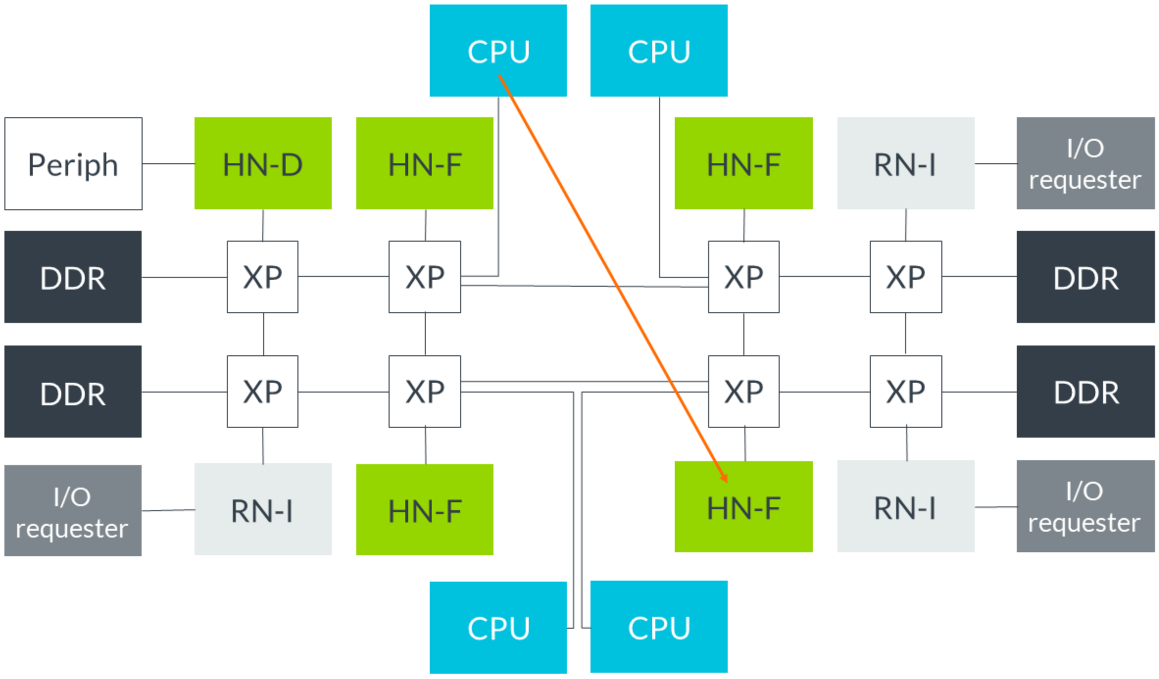

Figure 11: CPU executes the ReadShared request

2. The HN-F does not have the requested data in its cache, so it generates snoops to the CPUs that do have it, as shown in the following diagram:

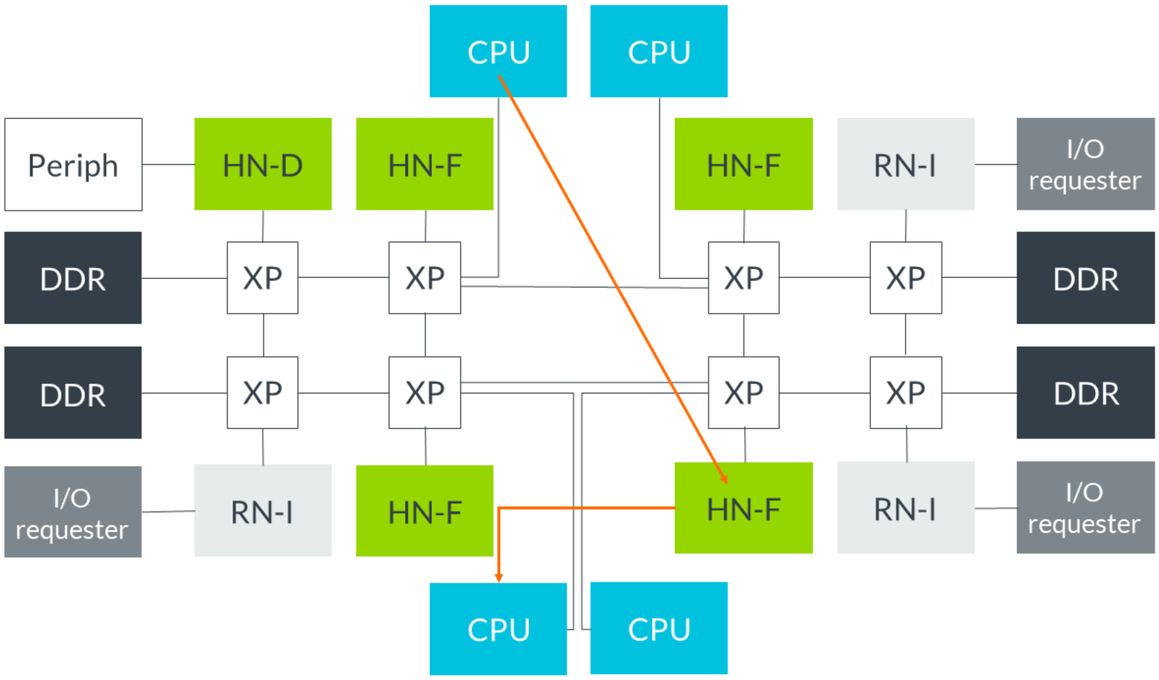

Figure 12: HN-F generates snoops to the CPU

3. The HN-F receives the snoop data from the CPU then forwards the snoop data to the requester.
4. The CPU performs the addition and executes a store exclusive instruction to update the counter.
5. The store exclusive instruction generates a CleanUnique request to the HN-F.
6. The HN-F sends invalidating snoops to the CPUs holding the cache line.
7. The HN-F receives a response from the CPU.
8. The HN-F responds to the requesting CPU and completes the CleanUnique request.

The exclusive accesses required two snoop requests, which adds latency in large systems.

7.2 Atomic add using AtomicStore transaction

This example shows how CHI-B atomics reduce latency when performing far-atomic operations. In this example, we see how far-atomics are used to increment a Shared counter.

The HN-F in the example holds the Shared counter in its cache, so a far-atomic is possible.

The flow of the atomic add operation is as follows:

1. The CPU executes a store Add instruction that generates an AtomicStore transaction to the HN-F, as shown in the diagram:

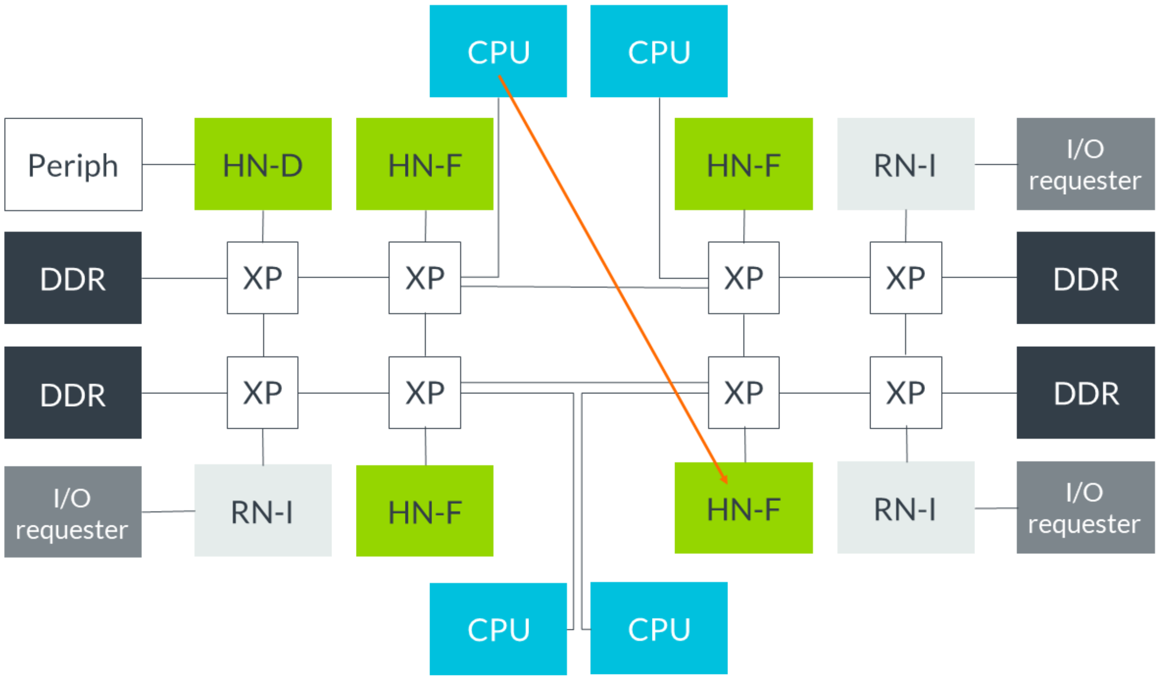

Figure 13: CPU generates an AtomicStore transaction to the HN-F

2. The HN-F contains a full Arithmetic Logic Unit (ALU) and performs the atomic operation. An AtomicStore transaction does not require a response, so the CPU does not track the transaction as outstanding.
3. Because the CPU does not track the AtomicStore, the transaction completes when it is issued.

This second example has significantly less latency than the first example. The exclusive accesses in the first example had to fetch data and invalidate the caches of other requesters. This process means that the CPU could perform the atomic operation. But in CHI-B, having a full ALU in the interconnect or at the completer reduces the number of transactions required to perform an atomic operation. This reduction leads to better system performance.

8 System support

In this section, learn about other considerations a designer can make when implementing support for atomic transactions.

8.1 Self-snooping

CHI-B permits self-snooping of the Requesting Node for atomic transactions.

Self-snooping removes the need for an RN-F to look up the line in its cache and invalidate it if the RN has the line in the Shared state.

Note: ACE-Lite and AXI Requesters do not support self-snooping because they do not have coherent caches.

The SnoopMe field controls self-snooping in the request Flit. This field is only applicable for atomic transactions. The Home Node receiving a transaction with SnoopMe = 1 must send a snoop to the Requester, if it determines that the cache line might be present at the Requester. Use of this field depends on the microarchitecture of the Requester itself. An RN-F can look in its own cache before issuing an atomic transaction externally and perform the atomic locally if the line was found in a Unique state.

8.2 Atomic support in subordinates

CHI-B makes it optional for a Subordinate Device to support atomic transactions:

- If a Subordinate does not support atomics, the interconnect must not send any atomic transactions to that Subordinate.
- Atomic transactions marked as Device Memory must be passed to the appropriate endpoint Subordinate. If that Subordinate does not support atomics, the interconnect must give an appropriate error response for that transaction.
- If a transaction is marked as Snoopable, the transaction must be forwarded to the Subordinate. This transaction occurs after the snoops have completed and dirty cached data is written back to the Subordinate.
- A Subordinate Node can limit atomic transactions support to specific memory types or address regions. If an atomic transaction targets a memory region that does not support atomics, the SN must give an appropriate error response for that transaction.

8.3 Error handling

The interconnect can send a response and read data before or after it performs the atomic operation. The response and the read data do not depend on the result of the atomic operation. The data to return is always the current value in memory, and the dataless responses indicate that the transmitted data has been received.

However, if a component supports error signaling, it must delay the response to the Requester. An example of an error can be corrupted write data or a request to a memory region that does not support atomics. In both cases, the component must wait until the operation has completed, so that an error can be signaled back to the Requester if needed.

9 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
- Introducing the AMBA Coherent Hub Interface

10 Next steps

This supplementary guide to Introducing the AMBA Coherent Hub Interface introduced the atomic transactions added to the AMBA CHI-B specification. You also learned about atomic usage with systems.

This guide explained how atomic transactions were previously supported using Exclusives and the performance improvements provided by the newer transactions. Finally, you learned about other system requirements.

As a next step, you can learn more about the AMBA CHI protocol specification. Use your knowledge to develop and use components 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.