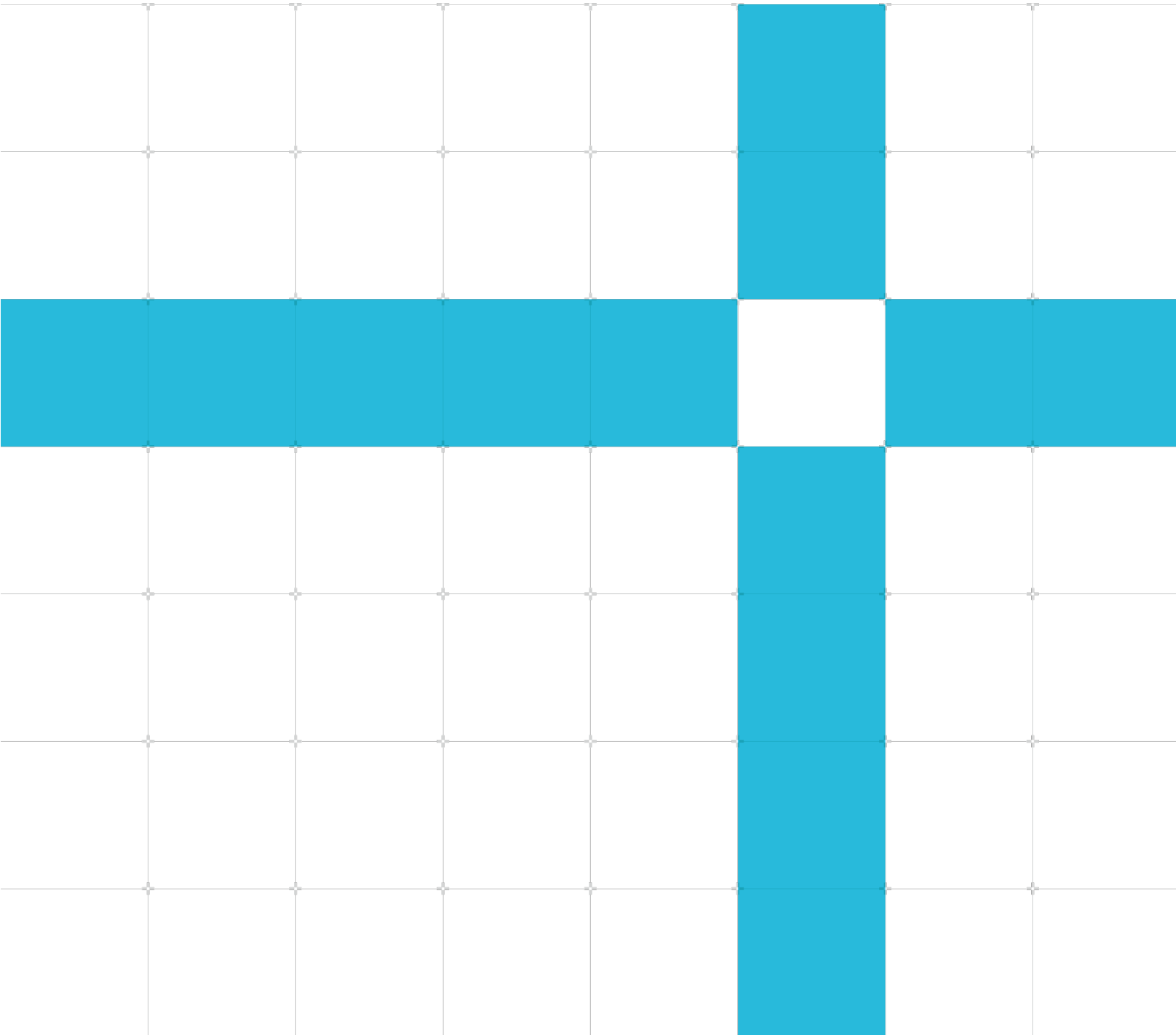
arm

Introduction to AMBA AXI4

Non-Confidential

Issue 0101
102202



Introduction to AMBA AXI4

Release information

Document history

Issue	Date	Confidentiality	Change
01	25 August 2020	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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.

Web Address

www.arm.com

Contents

1 Overview

This guide introduces the main features of Advanced Microcontroller Bus Architecture (AMBA) AXI4, highlighting the differences from the previous version AXI3. The guide explains the key concepts and details that help you implement the AXI4 protocol.

In this guide, we describe:

- What AMBA is.
- Why AMBA is so popular in modern SoC design.
- The concepts of transfers and transactions, which underpin how AMBA operates.
- The different channel signals and the functionality that they provide.
- Exclusive access transfers, which allow multiple masters to access the same slave at the same time.
- The rules and conditions that the AMBA protocol dictates.
- The key attributes and support for common elements like mixed endian structures.

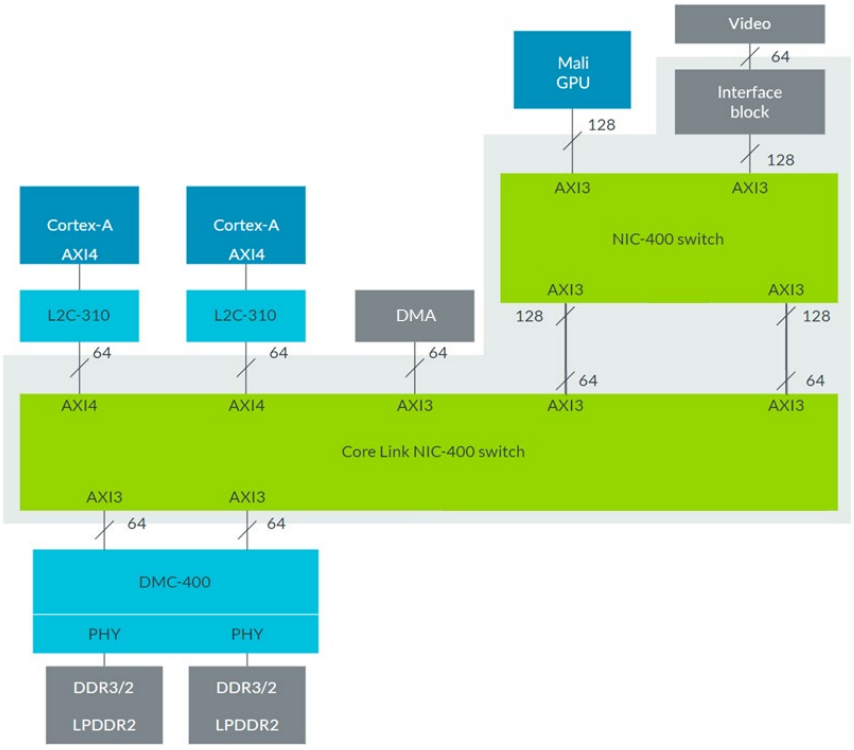
At the end of this guide, you can Check your knowledge.

2 What is AMBA, and why use it?

The Arm Advanced Microcontroller Bus Architecture, or AMBA, is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs.

Essentially, AMBA protocols define how functional blocks communicate with each other.

The following diagram shows an example of an SoC design. This SoC has several functional blocks that use AMBA protocols like AXI4 and AXI3 to communicate with each other:



2.1 Where is AMBA used?

AMBA simplifies the development of designs with multiple processors and large numbers of controllers and peripherals. However, the scope of AMBA has increased over time, going far beyond just microcontroller devices.

Today, AMBA is widely used in a range of ASIC and SoC parts. These parts include applications processors that are used in devices like IoT subsystems, smartphones, and networking SoCs.

2.2 Why use AMBA?

AMBA provides several benefits:

- Efficient IP reuse
  IP reuse is an essential component in reducing SoC development costs and timescales. AMBA specifications provide the interface standard that enables IP reuse. Therefore, thousands of SoCs, and IP products, are using AMBA interfaces.
- Flexibility
  AMBA offers the flexibility to work with a range of SoCs. IP reuse requires a common standard while supporting a wide variety of SoCs with different power, performance, and area requirements. Arm offers a range of interface specifications that are optimized for these different requirements.
- Compatibility
  A standard interface specification, like AMBA, allows compatibility between IP components from different design teams or vendors.
- Support
  AMBA is well supported. It is widely implemented and supported throughout the semiconductor industry, including support from third-party IP products and tools.

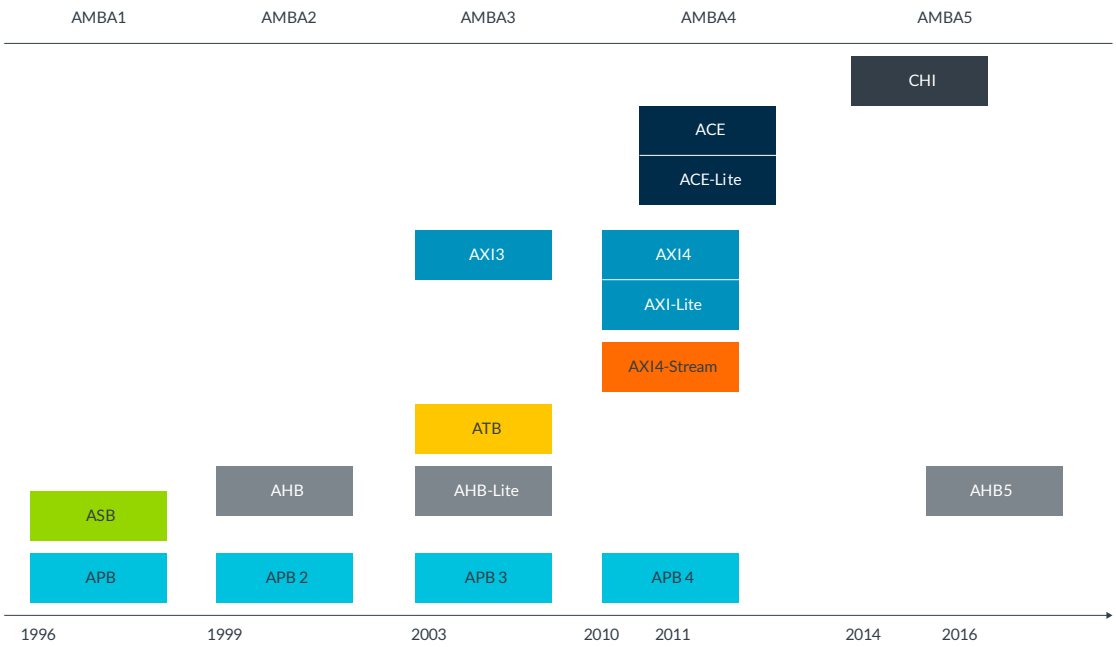
Bus interface standards like AMBA, are differentiated through the performance that they enable. The two main characteristics of bus interface performance are:

- Bandwidth
  The rate at which data can be driven across the interface. In a synchronous system, the maximum bandwidth is limited by the product of the clock speed and the width of the data bus.
- Latency
  The delay between the initiation and completion of a transaction. In a burst-based system, the latency figure often refers to the completion of the first transfer rather than the entire burst.

The efficiency of your interface depends on the extent to which it achieves the maximum bandwidth with zero latency.

2.3 How has AMBA evolved?

AMBA has evolved over the years to meet the demands of processors and new technologies, as shown in the following diagram:



2.3.1 AMBA 1

Arm introduced AMBA in the late 1990s. The first AMBA buses were the Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB). ASB has been superseded by more recent protocols, while APB is still widely used today.

APB is designed for low-bandwidth control accesses, for example, register interfaces on system peripherals. This bus has a simple address and data phase and a low complexity signal list.

2.3.2 AMBA 2

In 1999, AMBA 2 added the AMBA High-performance Bus (AHB), which is a single clock-edge protocol. A simple transaction on the AHB consists of an address phase and a subsequent data phase. Access to the target device is controlled through a MUX, admitting access to one master at a time. AHB is pipelined for performance, while APB is not pipelined for design simplicity.

2.3.3 AMBA 3

In 2003, Arm introduced the third generation, AMBA 3, which includes ATB and AHB-Lite.

Advanced Trace Bus (ATB), is part of the CoreSight on-chip debug and trace solution.

AHB-Lite is a subset of AHB. This subset simplifies the design for a bus with a single master.

Advanced eXtensible Interface (AXI), the third generation of AMBA interface defined in the AMBA 3 specification, is targeted at high performance, high clock frequency system designs. AXI includes features that make it suitable for high-speed submicrometer interconnect.

2.3.4 AMBA 4

In 2010, the AMBA 4 specifications were introduced, starting with AMBA 4 AXI4 and then AMBA 4 AXI Coherency Extensions (ACE) in 2011.

ACE extends AXI with additional signaling introducing system-wide coherency. This system-wide coherency allows multiple processors to share memory and enables technology like big.LITTLE processing. At the same time, the ACE-Lite protocol enables one-way coherency. One-way coherency enables a network interface to read from the caches of a fully coherent ACE processor.

The AXI4-Stream protocol is designed for unidirectional data transfers from master to slave with reduced signal routing, which is ideal for implementation in FPGAs.

2.3.5 AMBA 5

In 2014, the AMBA 5 Coherent Hub Interface (CHI) specification was introduced, with a redesigned high-speed transport layer and features designed to reduce congestion. There have been several editions of the CHI protocol, and each new version adds new features.

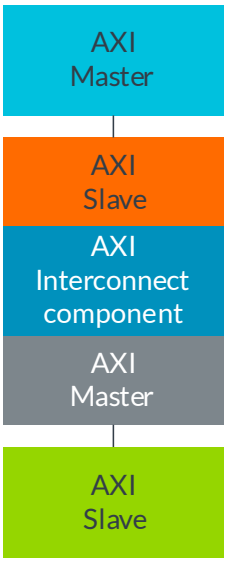
In 2016, the AHB-Lite protocol was updated to AHB5, to complement the Armv8-M architecture, and extend the TrustZone security foundation from the processor to the system.

In 2019, the AMBA Adaptive Traffic Profiles (ATP) was introduced. ATP complements the existing AMBA protocols and is used for modeling high-level memory access behavior in a concise, simple, and portable way.

3 AXI protocol overview

AXI is an interface specification that defines the interface of IP blocks, rather than the interconnect itself.

The following diagram shows how AXI is used to interface an interconnect component:

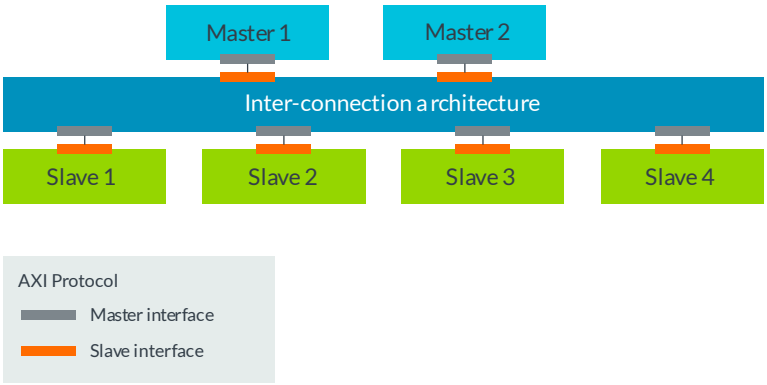


In AX3 and AXI4, there are only two AXI interface types, master and slave. These interface types are symmetrical. All AXI connections are between master interfaces and slave interfaces.

AXI interconnect interfaces contain the same signals, which makes integration of different IP relatively simple. The previous diagram shows how AXI connections join master and slave interfaces. The direct connection gives maximum bandwidth between the master and slave components with no extra logic. And with AXI, there is only a single protocol to validate.

3.1 AXI in a multi-master system

The following diagram shows a simplified example of an SoC system, which is composed of masters, slaves, and the interconnect that links them all:



An Arm processor is an example of a master, and a simple example of a slave is a memory controller.

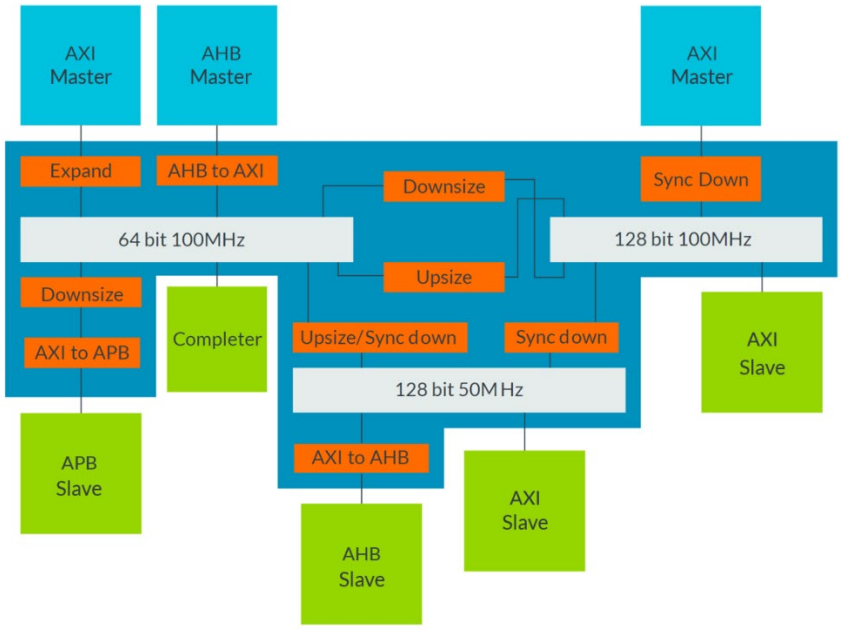
The AXI protocol defines the signals and timing of the point-to-point connections between masters and slaves.

Note

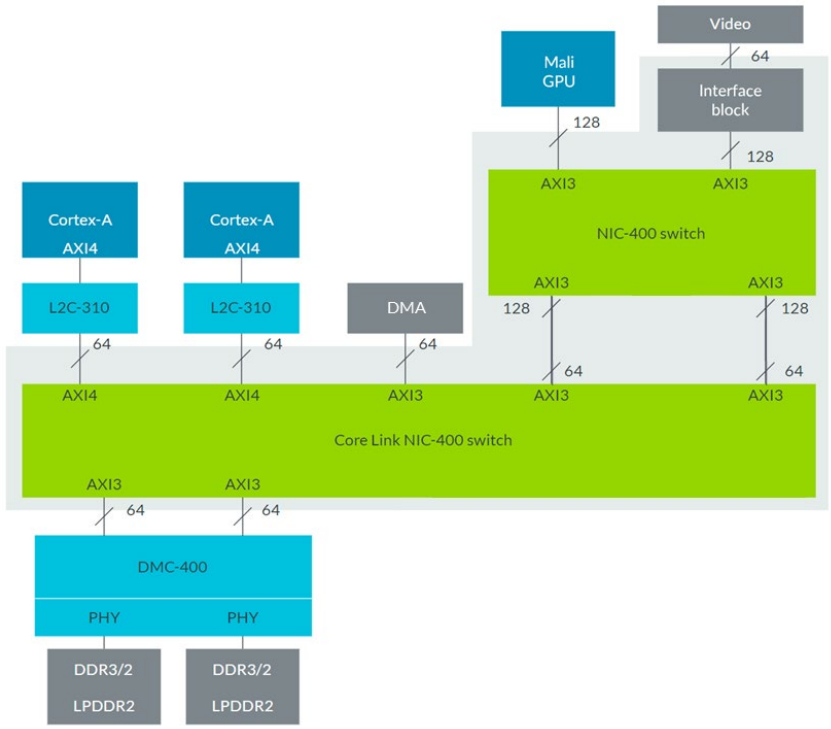
The AXI protocol is a point-to-point specification, not a bus specification. Therefore, it describes only the signals and timing between interfaces.

The previous diagram shows that each AXI master interface is connected to a single AXI slave interface. Where multiple masters and slaves are involved, an interconnect fabric is required. This interconnect fabric also implements slave and master interfaces, where the AXI protocol is implemented.

The following diagram shows that the interconnect is a complex element that requires its own AXI master and slave interfaces to communicate with external function blocks:



The following diagram shows an example of an SoC with various processors and function blocks:

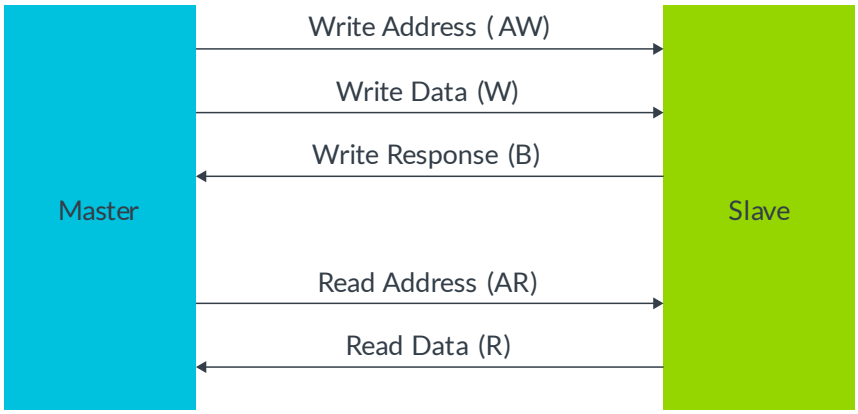


The previous diagram shows all the connections where AXI is used. You can see that AXI3 and AXI4 are used within the same SoC, which is common practice. In such cases, the interconnect performs the protocol conversion between the different AXI interfaces.

3.2 AXI channels

The AXI specification describes a point-to-point protocol between two interfaces: a master and a slave.

The following diagram shows the five main channels that each AXI interface uses for communication:



Write operations use the following channels:

- The master sends an address on the **Write Address (AW)** channel and transfers data on the **Write Data (W)** channel to the slave.
- The slave writes the received data to the specified address. Once the slave has completed the write operation, it responds with a message to the master on the **Write Response (B)** channel.

Read operations use the following channels:

- The master sends the address it wants to read on the **Read Address (AR)** channel.
- The slave sends the data from the requested address to the master on the **Read Data (R)** channel.

  The slave can also return an error message on the **Read Data (R)** channel. An error occurs if, for example, the address is not valid, or the data is corrupted, or the access does not have the right security permission.

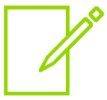

Note

Each channel is unidirectional, so a separate **Write Response** channel is needed to pass responses back to the master. However, there is no need for a **Read Response** channel, because a read response is passed as part of the **Read Data** channel.

Using separate address and data channels for read and write transfers helps to maximize the bandwidth of the interface. There is no timing relationship between the groups of read and write channels. This means that a read sequence can happen at the same time as a write sequence.

Each of these five channels contains several signals, and all these signals in each channel have the prefix as follows:

- AW for signals on the Write Address channel
- AR for signals on the Read Address channel
- W for signals on the Write Data channel
- R for signals on the Read Data channel
- B for signals on the Write Response channel


Note

B stands for buffered, because the response from the slave happens after all writes have completed.

3.3 Main AXI features

The AXI protocol has several key features that are designed to improve bandwidth and latency of data transfers and transactions, as you can see here:

Independent read and write channels

AXI supports two different sets of channels, one for write operations, and one for read operations. Having two independent sets of channel helps to improve the bandwidth performances of the interfaces. This is because read and write operations can happen at the same time.

Multiple outstanding addresses

AXI allows for multiple outstanding addresses. This means that a master can issue transactions without waiting for earlier transactions to complete. This can improve system performance because it enables parallel processing of transactions.

No strict timing relationship between address and data operations

With AXI, there is no strict timing relationship between the address and data operations. This means that, for example, a master could issue a write address on the Write Address channel, but there is no time requirement for when the master has to provide the corresponding data to write on the Write Data channel.

Support for unaligned data transfers

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

Out-of-order transaction completion

Out-of-order transaction completion is possible with AXI. The AXI protocol includes transaction identifiers, and there is no restriction on the completion of transactions with different ID values.

This means that a single physical port can support out-of-order transactions by acting as several logical ports, each of which handles its transactions in order.

Burst transactions based on start address

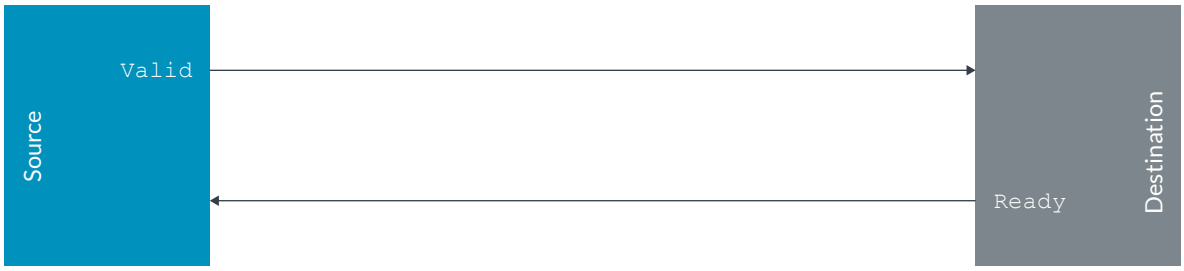
AXI masters only issue the starting address for the first transfer. For any following transfers, the slave will calculate the next transfer address based on the burst type.

4 Channel transfers and transactions

This section explains the handshake principle for AXI channels, and shows how the handshake is the underpinning mechanism for all read and write transactions.

4.1 Channel handshake

The AXI4 protocol defines five different channels, as described in AXI channels. All of these channels share the same handshake mechanism that is based on the VALID and READY signals, as shown in the following diagram:



The VALID signal goes from the source to the destination, and READY goes from the destination to the source.

Whether the source or destination is a master or slave depends on which channel is being used. For example, the master is a source for the Read Address channel, but a destination for the Read Data channel.

The source uses the VALID signal to indicate when valid information is available. The VALID signal must remain asserted, meaning set to high, until the destination accepts the information. Signals that remain asserted in this way are called sticky signals.

The destination indicates when it can accept information using the READY signal. The READY signal goes from the channel destination to the channel source.

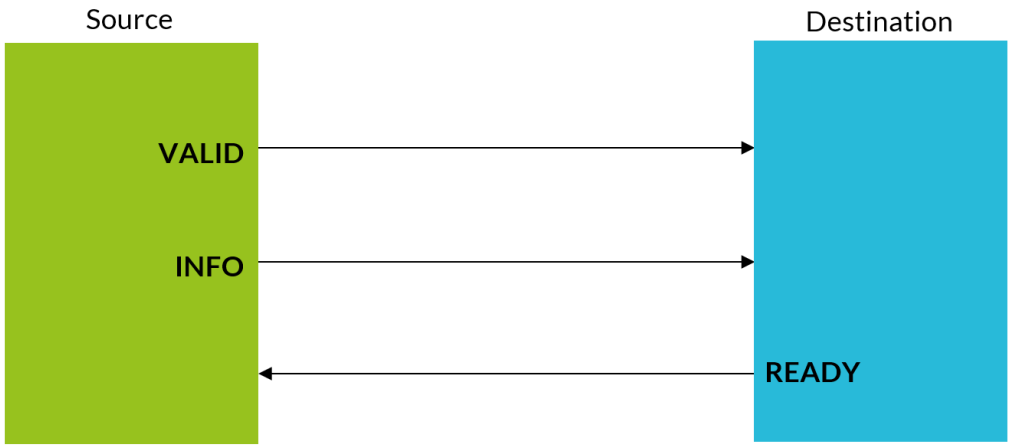
This mechanism is not an asynchronous handshake, and requires the rising edge of the clock for the handshake to complete.

4.2 Differences between transfers and transactions

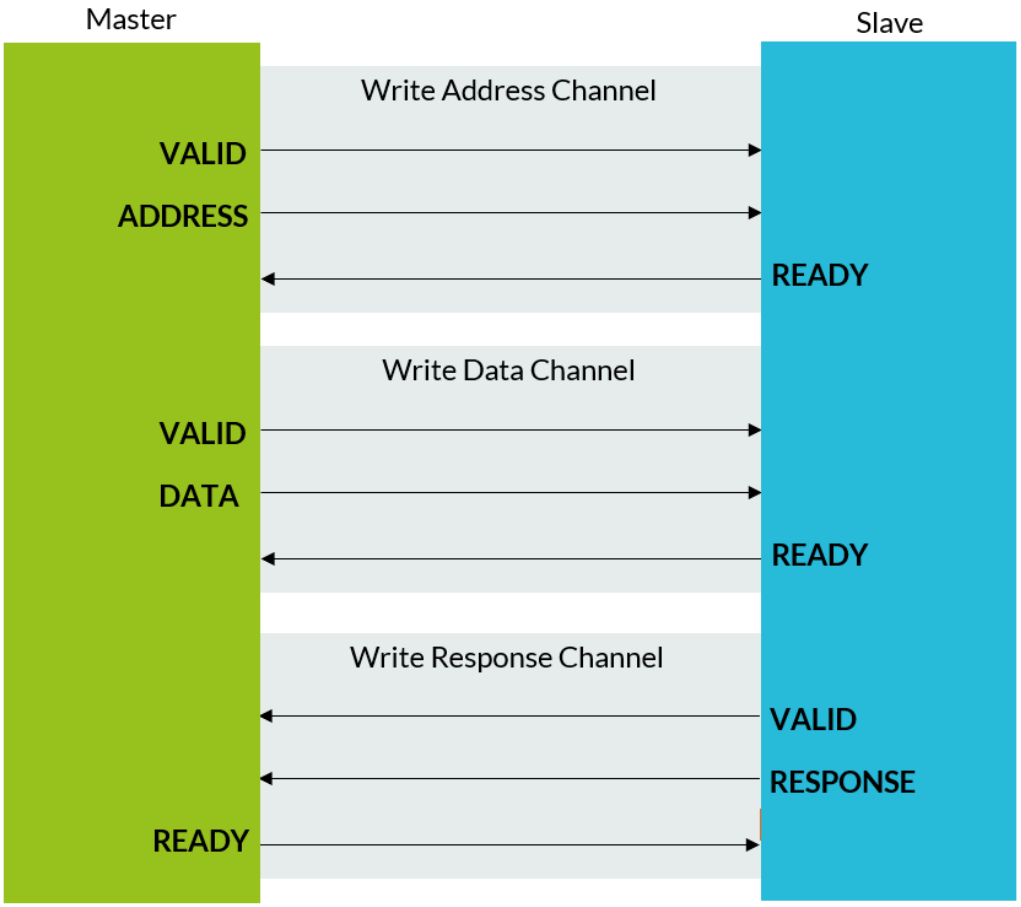
When designing interconnect fabric, you must know the capabilities of the masters and slaves that are being connected. Knowing this information lets you include sufficient buffering, tracking, and decode logic to support the various data transfer ordering possibilities that allow performance improvements in faster devices.

Using standard terminology makes understanding the interactions between connected components easier. AXI makes a distinction between transfers and transactions:

- A transfer is a single exchange of information, with one VALID and READY handshake. The following diagram shows a transfer:



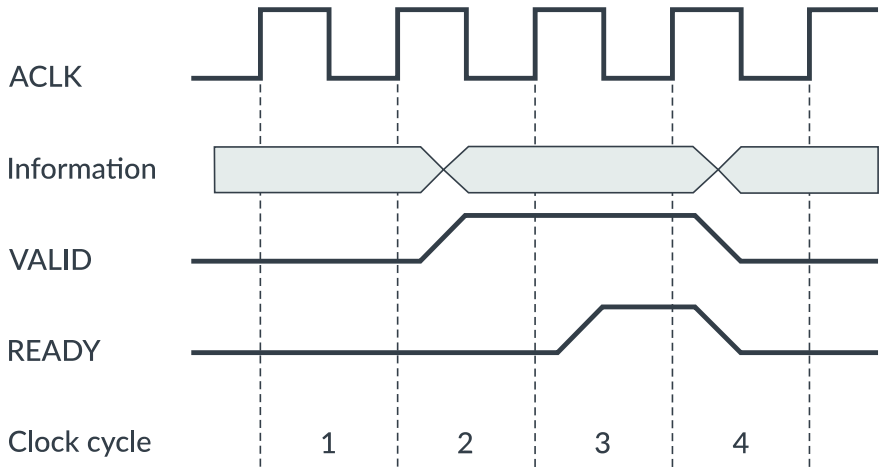
- A transaction is an entire burst of transfers, containing an address transfer, one or more data transfers, and, for write sequences, a response transfer. The following diagram shows a transaction:



4.3 Channel transfer examples

This section examines some examples of possible handshakes between source and destination. It shows several possible combinations of VALID and READY sequences that conform to the AXI protocol specifications.

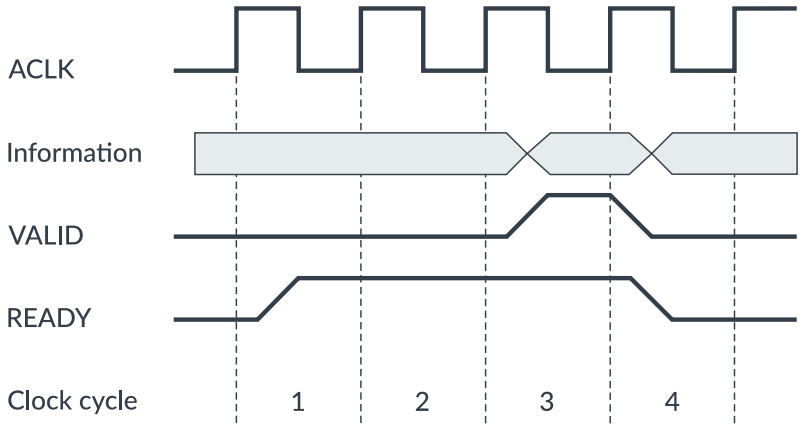
In the first example, shown in the following diagram, we have a clock signal, followed by an information bus, and then the VALID and READY signals:



This example has the following sequence of events:

1. In clock cycle 2, the VALID signal is asserted, indicating that the data on the information channel is valid.
2. In clock cycle 3, the following clock cycle, the READY signal is asserted.
3. The handshake completes on the rising edge of clock cycle 4, because both READY and VALID signals are asserted.

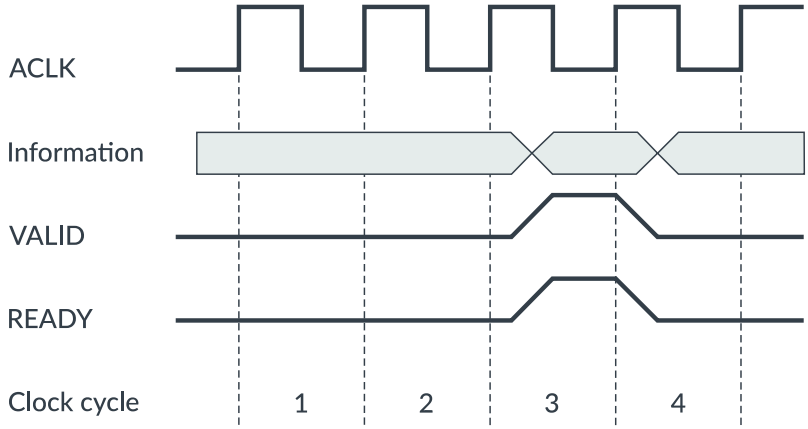
The following diagram shows another example:



This example has the following sequence of events:

1. In clock cycle 1, the READY signal is asserted.
2. The VALID signal is not asserted until clock cycle 3.
3. The handshake completes on the rising edge of clock cycle 4, when both VALID and READY are asserted.

The final example shows both VALID and READY signals being asserted during the clock cycle 3, as seen in the following diagram:



Again, the handshake completes on the rising edge of clock cycle 4, when both VALID and READY are asserted.

In all three examples, information is passed down the channel when READY and VALID are asserted on the rising edge of the clock signal.

Read and write handshakes must adhere to the following rules:

- A source cannot wait for READY to be asserted before asserting VALID.
- A destination can wait for VALID to be asserted before asserting READY.

These rules mean that READY can be asserted before or after VALID, or even at the same time.

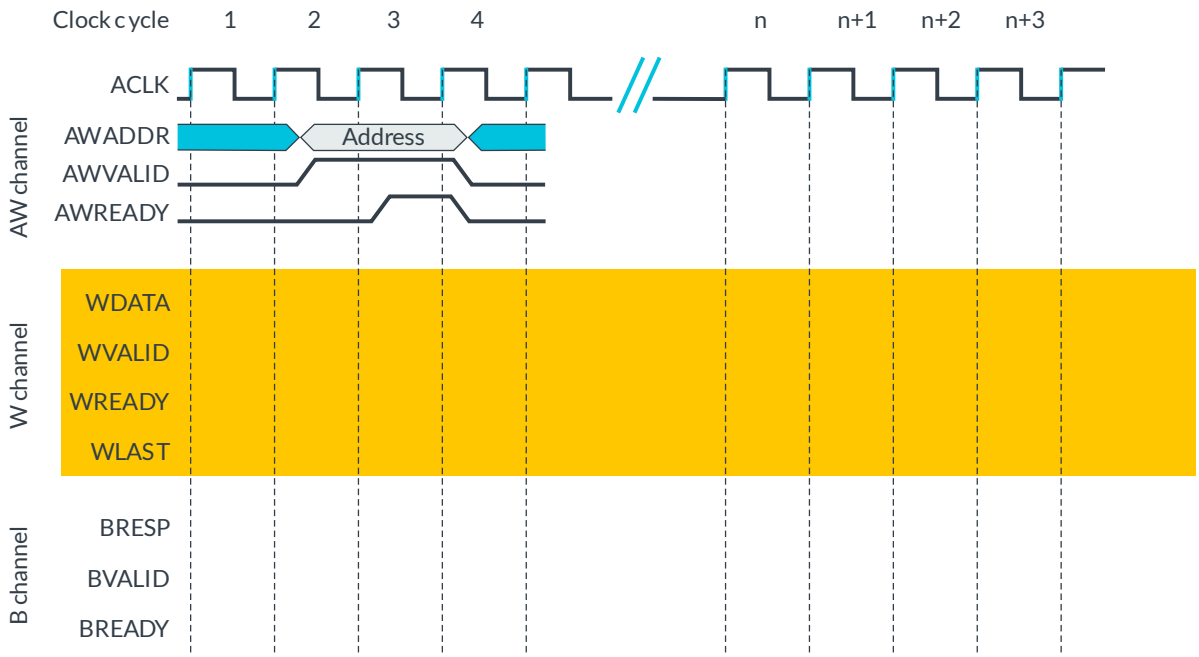
4.4 Write transaction: single data item

This section describes the process of a write transaction for a single data item, and the different channels that are used to complete the transaction.

This write transaction involves the following channels:

- Write Address (AW)
- Write (W)
- Write Response (B)

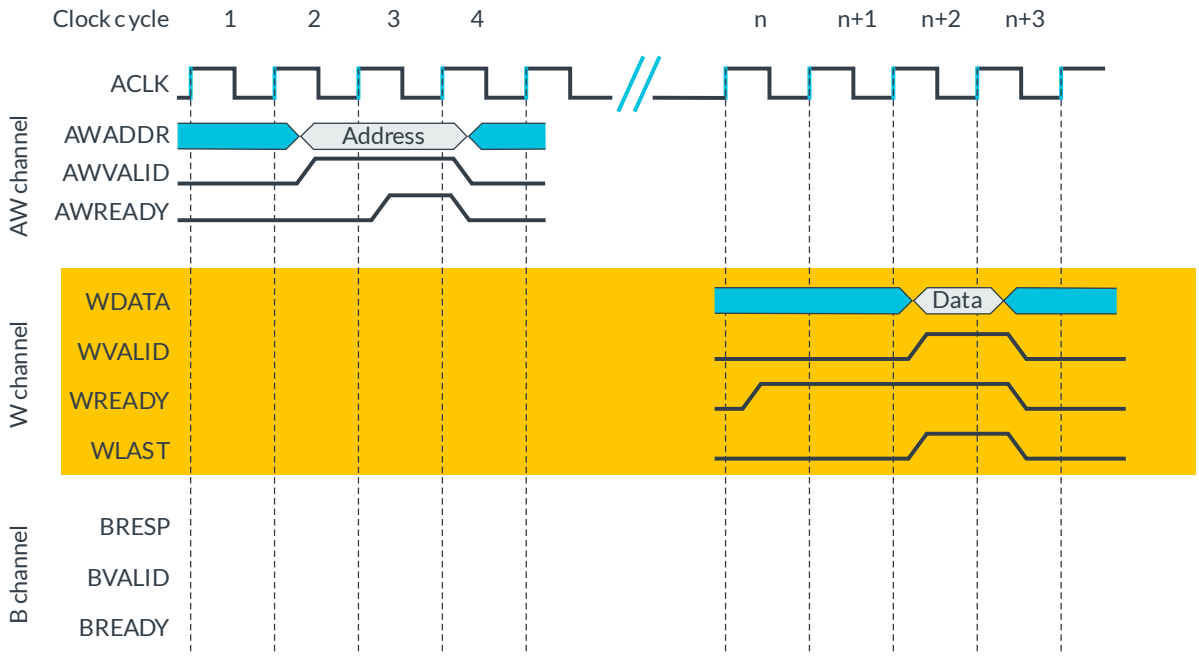
First, there is a handshake on the Write Address (AW) channel, as shown in the following diagram:



This handshake is where the master communicates the address of the write to the slave. The handshake has the following sequence of events:

1. The master puts the address on AWADDR and asserts AWVALID in clock cycle 2.
2. The slave asserts AWREADY in clock cycle 3 to indicate its ability to receive the address value.
3. The handshake completes on the rising edge of clock cycle 4.

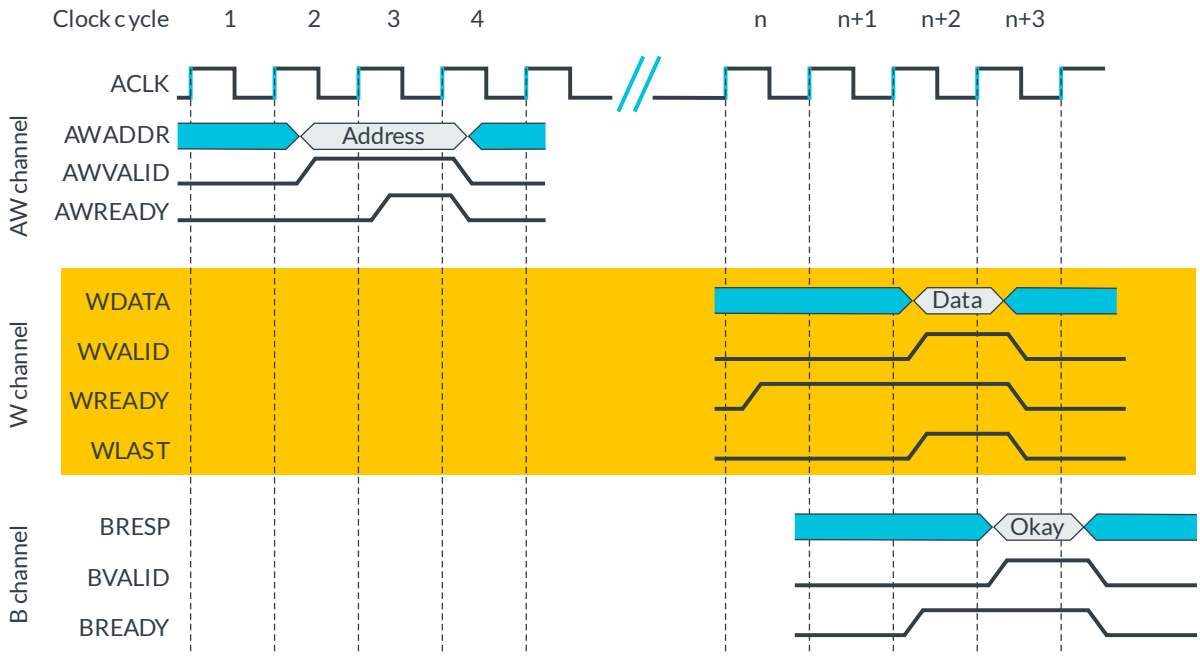
After this first handshake, the master transfers the data to the slave on the Write (W) channel, as shown in the following diagram:



The data transfer has the following sequence of events:

1. The slave is waiting for data with WREADY set to high in clock cycle n.
2. The master puts the data on the WDATA bus and asserts WVALID in clock cycle n+2.
3. The handshake completes on the rising edge of clock cycle n+3

Finally, the slave uses the Write Response (B) channel, to confirm that the write transaction has completed once all WDATA has been received. This response is shown in the following diagram:



The write response has the following sequence of events:

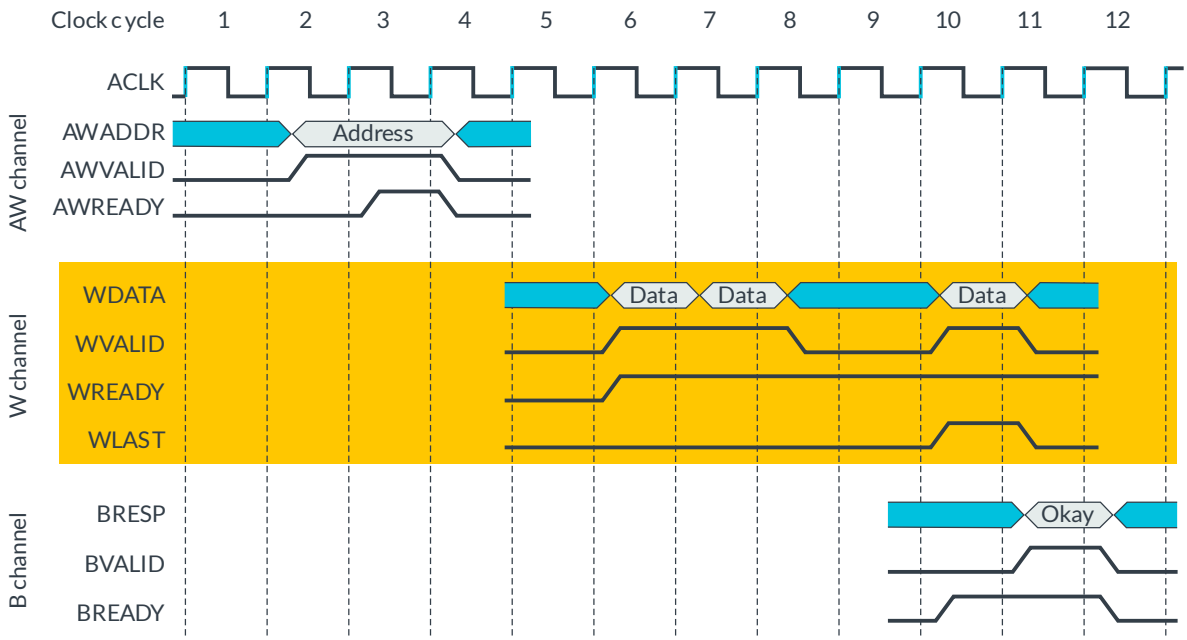
1. The master asserts BREADY.
2. The slave drives BRESP to indicate success or failure of the write transaction, and asserts BVALID.

The handshake completes on the rising edge of clock cycle n+3.

4.5 Write transaction: multiple data items

AXI is a burst-based protocol, which means that it is possible to transfer multiple data in a single transaction. We can transfer a single address on the AW channel to transfer multiple data, with associated burst width and length information.

The following diagram shows an example of a multiple data transfer:



In this case, the AW channel indicates a sequence of three transfers, and on the W channel, we see three data transfers.

The master drives the WLAST high to indicate the final WDATA. This means that the slave can either count the data transfers or just monitor WLAST.

Once all WDATA transfers are received, the slave gives a single BRESP value on the B channel. One single BRESP covers the entire burst. If the slave decides that any of the transfers contain an error, it must wait until the entire burst has completed before it informs the master that an error occurred.

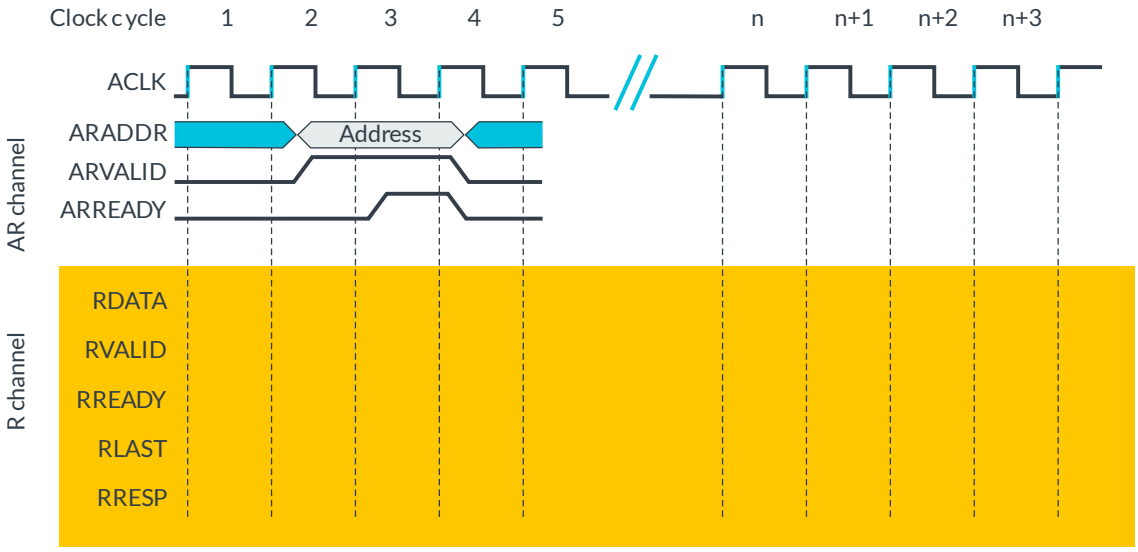
4.6 Read transaction: single data item

This section looks in detail at the process of a read transaction for a single data item, and the different channels used to complete the transaction.

This write transaction involves the following channels:

- Read Address (AR)
- Read (R)

First, there is a handshake on the Read Address (AR) channel, as shown in the following diagram:

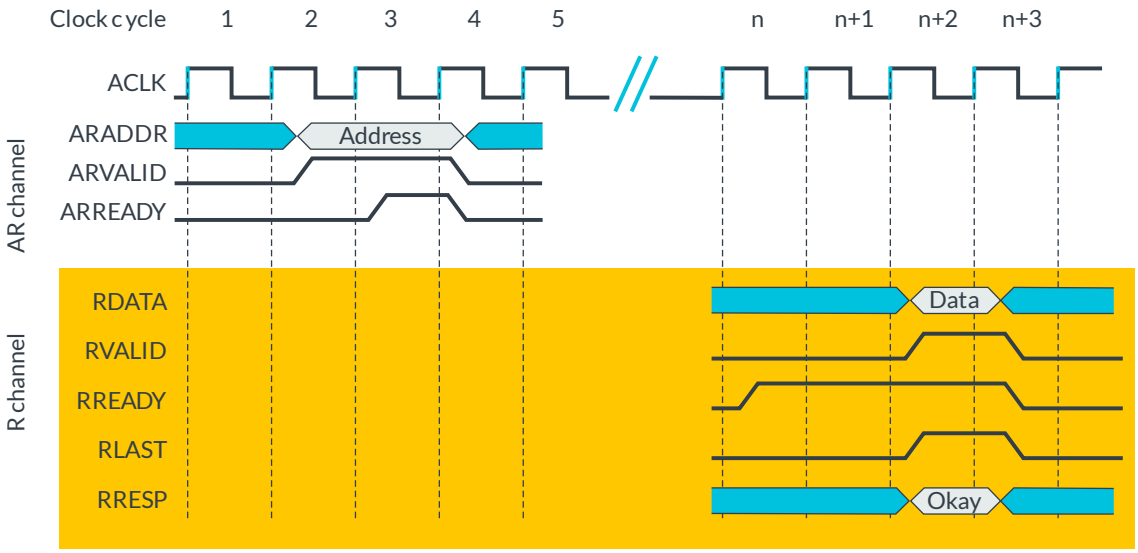


The handshake has the following sequence of events:

1. In clock cycle 2, the master communicates the address of the read to the slave on ARADDR and asserts ARVALID.
2. In clock cycle 3, the slave asserts ARREADY to indicate that it is ready to receive the address value.

The handshake completes on the rising edge of clock cycle 4.

Next, on the Read (R) channel, the slave transfers the data to the master. The following diagram shows the data transfer process:



The data transfer handshake has the following sequence of events:

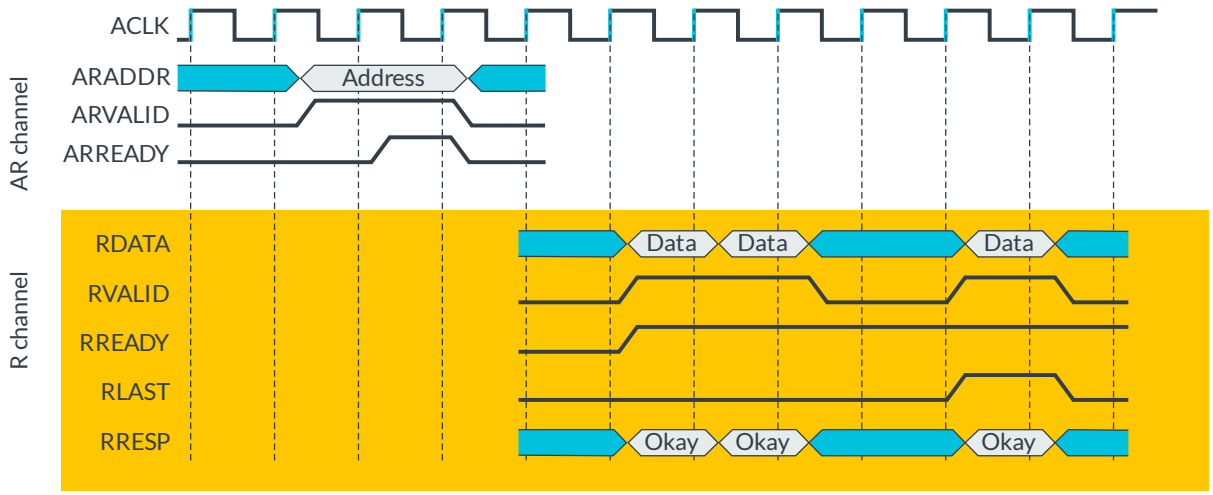
1. In clock cycle n, the master indicates that it is waiting to receive the data by asserting RREADY.
2. The slave retrieves the data and places it on RDATA in clock cycle n+2. In this case, because this is a single data transaction, the slave also sets the RLAST signal to high.

   At the same time, the slave uses RRESP to indicate the success or failure of the read transaction to the master, and asserts RVALID.
3. Because RREADY is already asserted by the master, the handshake completes on the rising edge of clock cycle n+3.

4.7 Read transaction: multiple data items

The AXI protocol also allows a read burst of multiple data transfer in the same transaction. This is similar to the write burst that is described in **Write transaction: multiple data items**.

The following diagram shows an example of a burst read transfer:



In this example, we transfer a single address on the AR channel to transfer multiple data items, with associated burst width and length information.

Here, the AR channel indicates a sequence of three transfers, therefore on the R channel, we see three data transfers from the slave to the master.

On the R channel, the slave transfers the data to the master. In this example, the master is waiting for data as shown by RREADY set to high. The slave drives valid RDATA and asserts RVALID for each transfer.

One difference between a read transaction and a write transaction is that for a read transaction there is an RRESP response for every transfer in the transaction. This is because, in the write transaction, the slave has to send the response as a separate transfer on the B channel. In the read transaction, the slave uses the same channel to send the data back to the master and to indicate the status of the read operation.

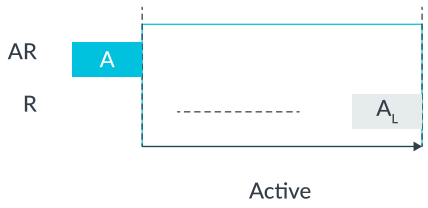
If an error is indicated for any of the transfers in the transaction, the full indicated length of the transaction must still be completed. There is no such thing as early burst termination.

4.8 Active transactions

Active transactions are also known as outstanding transactions.

An active read transaction is a transaction for which the read address has been transferred, but the last read data has not yet been transferred at the current point in time.

With reads, the data must come after the address, so there is a simple reference point for when the transaction starts. This is shown in the following diagram:

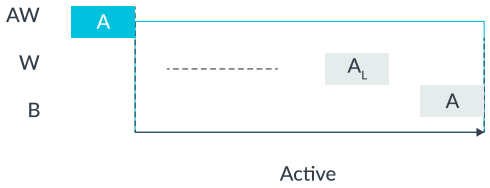


For write transactions, the data can come after the address, but leading write data is also allowed. The start of a write transaction can therefore be either of the following:

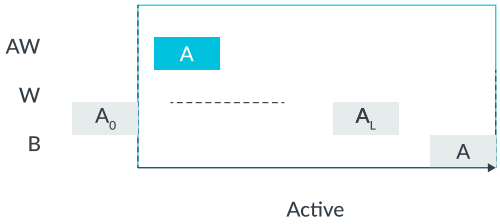
- The transfer of the write address
- The transfer of leading write information

Therefore, an active write transaction is a transaction for which the write address or leading write data has been transferred, but the write response has not yet been transferred.

The following diagram shows an active write transaction where the write address has been transferred, but the write response has not yet been transferred:



The following diagram shows an active write transaction where the leading write data has been transferred, but the write response has not yet been transferred:



5 Channel signals

This section introduces the main AXI signals and attributes, and explains how they are used to improve system performance.

The AXI protocol defines five channels: three for write signals, and two for read signals.

5.1 Write channel signals

The channels used for a write transaction are:

- Write Address
- Write Data
- Write Response

The following table shows the Write Address channel signals:

Write Address (AW) channel signals	AXI version
AWVALID	AXI3 and AXI4
AWREADY	AXI3 and AXI4
AWADDR[31:0]	AXI3 and AXI4
AWSIZE[2:0]	AXI3 and AXI4
AWBURST[1:0]	AXI3 and AXI4
AWCACHE[3:0]	AXI3 and AXI4
AWPROT[2:0]	AXI3 and AXI4
AWID[x:0]	AXI3 and AXI4
AWLEN[3:0] AWLEN[7:0]	AXI3 only AXI4 only
AWLOCK[1:0] AWLOCK	AXI3 only AXI4 only
AWQOS[3:0]	AXI4 only
AWREGION[3:0]	AXI4 only
AWUSER[x:0]	AXI4 only

The following table shows the Write Data channel signals:

Write Data (W) channel signals	AXI version
WVALID	AXI3 and AXI4
WREADY	AXI3 and AXI4
WLAST	AXI3 and AXI4
WDATA[x:0]	AXI3 and AXI4
WSTRB[x:0]	AXI3 and AXI4
WID[x:0]	AXI3 only

Write Data (W) channel signals	AXI version
WUSER[x:0]	AXI4 only

The following table shows the Write Response channel signals:

Write Response (B) channel signals	AXI version
BVALID	AXI3 and AXI4
BREADY	AXI3 and AXI4
BRESP[1:0]	AXI3 and AXI4
BID[x:0]	AXI3 and AXI4
BUSER[x:0]	AXI4 only

All the signals in each channel have the same prefix:

- AW for the Write Address channel
- W for the Write Data channel
- B for the Write Response channel

There are some differences between the AXI3 protocol and the AXI4 protocol for the write channels:

- For the write address channel, the AWLEN signal is wider for the AXI4 protocol. Therefore, AXI4 is able to generate longer bursts than AXI3.
- AXI4 reduces the AWLOCK signal to a single bit to only accommodate exclusive transfers because locked transfers are not supported.
- AXI4 adds the AWQOS signal to the AW channel. This signal supports the concept of quality of service (QoS) in the AXI4 protocol.
- AXI4 adds the AWREGION signal to the AW channel. This signal supports slave regions which allow for multiple logical interfaces from a single physical slave interface.
- AXI4 removes the WID signal from the W channel. This is because write data reordering is no longer allowed.
- AXI4 adds user-defined signals to each channel.

5.2 Read channel signals

The channels used for a read transaction are:

- Read Address
- Read Data

The following table shows the Read Address channel signals:

Read Address (AR) channel signals	AXI version
ARVALID	AXI3 and AXI4
AREADY	AXI3 and AXI4
ARADDR[31:0]	AXI3 and AXI4

Read Address (AR) channel signals	AXI version
ARSIZE[2:0]	AXI3 and AXI4
ARBURST[1:0]	AXI3 and AXI4
ARCACHE[3:0]	AXI3 and AXI4
ARPROT[2:0}	AXI3 and AXI4
ARID[x:0]	AXI3 and AXI4
ARLEN[3:0] ARLEN[7:0]	AXI3 only AXI4 only
ARLOCK[1:0] ARLOCK	AXI3 only AXI4 only
ARQOS[3:0]	AXI4 only
ARREGION[3:0]	AXI4 only
ARUSER[x:0]	AXI4 only

The following table shows the Read Data channel signals:

Read Data (R) channel signals	AXI version
RVALID	AXI3 and AXI4
READY	AXI3 and AXI4
RLAST	AXI3 and AXI4
RDATA[x:0]	AXI3 and AXI4
RRESP[1:0]	AXI3 and AXI4
RID[x:0]	AXI3 and AXI4
RUSER[x:0]	AXI4 only

All the signals in each channel have the same prefix:

- AR for the Read Address channel
- R for the Read Data channel

There are some differences between the AXI3 protocol and the AXI4 protocol for the read channels:

- For the AXI4 protocol, the read address length signal ARLEN is wider. Therefore, AXI4 is able to generate longer read bursts than AXI3.
- AXI4 reduces the ARLOCK signal to a single bit to only accommodate exclusive transfers because locked transfers are not supported.
- As with the write channel signals, the concepts of quality of service and slave regions apply to read transactions. These use the ARQOS and ARREGION signals in the AR channel.
- AXI4 adds user-defined signals to the two read channels.

5.3 Data size, length, and burst type

Each read and write transaction has attributes that specify the data length, size, and the burst signal attributes for that transaction.

In the following list of attributes, x stands for write and read, so they apply to both the Write Address channel and the Read Address channel:

- AxLEN describes the length of the transaction in the number of transfers.
  - For AXI3, AxLEN[3:0] has 4 bits, which specifies a range of 1-16 transfers in a transaction.
  - For AXI4, AxLEN[7:0] has 8 bits, which specifies a range of 1-256 data transfers in a transaction.
- AxSize[2:0] describes the maximum number of bytes to transfer in each data transfer. Three bits of encoding indicate 1, 2, 4, 8, 16, 32, 64, or 128 bytes per transfer.
- AxBURST[1:0] describes the burst type of the transaction: fixed, incrementing, or wrapping.

  The following table shows the different properties of these burst types:

Value	Burst type	Usage notes	Length (number of transfers)	Alignment
0x00	FIXED	Reads the same address repeatedly. Useful for FIFOs.	1-16	Fixed byte lanes only defined by start address and size.
0x01	INCR	Incrementing burst. The slave increments the address for each transfer in the burst from the address for the previous transfer. The incremental value depends on the size of the transfer, as defined by the AxSIZE attribute. Useful for block transfers.	AXI3: 1-16 AXI4: 1-256	Unaligned transfers are supported.
0x10	WRAP	Wrapping burst. Similar to an incrementing burst, except that if an upper address limit is reached, the address wraps around to a lower address. Commonly used for cache line accesses.	2, 4, 8, or 16	The start address must be aligned to the transfer size.
0x11	RESERVED	Not for use.	-	-

5.4 Protection level support

AXI provides access permissions signals, AWPROT and ARPROT, that can protect against illegal transactions downstream in the system. For example, if a transaction does not have the correct level of protection, a memory controller could refuse read or write access by using these signals.

This is useful for security solutions like Arm TrustZone, where a processor has two separate states, Secure and Non-secure.

AxPROT defines three levels of access protection, as shown in the following diagram:

The AxPROT bit allocations specify the following attributes:

- AxPROT[0] (P) identifies an access as unprivileged or privileged:
  - 1 indicates privileged access.
  - 0 indicates unprivileged access.

  Although some processors support multiple levels of privilege, the only distinction that AXI can provide is between privileged and unprivileged access.
- AxPROT[1] (NS) identifies an access as Secure or Non-secure:
  - 1 indicates a Non-secure transaction.
  - 0 indicates a Secure transaction.
- AxPROT[2] (I) indicates whether the transaction is an instruction access or a data access:
  - 1 indicates an instruction access.
  - 0 indicates a data access.

  The AXI protocol defines this indication as a hint.

  It is not accurate in all cases, for example, where a transaction contains a mix of instruction and data items.

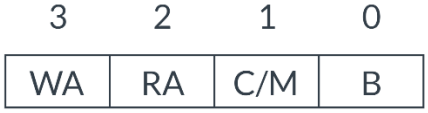
  The Arm AXI specification for both AXI 3 and AXI 4 recommends that a master sets bit 2 to zero to indicate a data access, unless the access is specifically known to be an instruction access.

5.5 Cache support

Modern SoC systems often contain caches that are placed in several points of the system. For example, the level 2 cache might be external to the processor, or the level 3 caches might be in front of the memory controller.

To support systems that use different caching policies, the AWCACHE and ARCACHE signals indicate how transactions are required to progress through a system.

The following diagram shows the AxCACHE bit allocations:



The AxCACHE bit allocations specify the following attributes:

- AxCACHE [0] (B) is the bufferable bit.

  When this bit is set to 1, the interconnect or any component can delay the transaction reaching its final destination for any number of cycles.

  The bufferable bit indicates whether the response can come from an intermediate point, or whether the response must come from the destination slave.

- AxCACHE [1] is the cacheable bit in AXI3, or the modifiable bit in AXI4.

  This bit indicates that the attributes of a transaction at the final destination do not have to match the attributes of the original transaction.

  For writes, setting the modifiable bit means that several different writes can be merged, or a single write can be broken into multiple transactions.

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

- AxCACHE [2] is the RA bit.

  The RA bit indicates that on a read, the allocation of the transaction is recommended, but not mandatory.

  If either AxCACHE [2] or AxCACHE [3] is asserted, then the transaction must be looked up in a cache as it could have been allocated in this cache by another master.

- AxCACHE [3] is the WA bit.

  The WA bit indicates that on a write, the allocation of the transaction is recommended, but not mandatory.

  If either AxCACHE [2] or AxCACHE [3] is asserted, then the transaction must be looked up in a cache as it could have been allocated in this cache by another master.


Note

If AxCACHE [1], the cacheable bit, is not asserted, then AxCACHE [2] and AxCACHE [3] cannot be asserted.

The reason for including read and write allocation on both read and write address buses is that it allows a system-level cache to optimize its performance.

For example, consider a cache that sees a read access defined as "write-allocate, but not read-allocate". In this case, the cache knows that the address might be stored in the cache because it could have been allocated on a previous write, and therefore it must do a cache lookup.

However, now consider that the cache sees a read access that is defined as "no write-allocate and no read-allocate". In this case, the cache knows that the address has not been allocated in the cache. The cache can avoid the lookup and immediately pass the transaction through to the other side. The cache can only do this if it knows both the read and write allocate for every transaction.

It is not a requirement that caches operate in this way, but the AXI protocol is defined with RA and WA for both reads and writes to allow this mode of operation if you or your cache designer want to implement it.

5.6 Response signaling

AXI provides response signaling for both read and write transactions.

For read transactions, the response information from the slave is signaled on the read data channel using RRESP.

For write transactions, the response information is signaled on the write response channel using BRESP.

RRESP and BRESP are both composed of two bits, and the encoding of these signals can transfer four responses, as shown in the following table:

Response code	Description
00 - OKAY	Normal access success or exclusive access failure. OKAY is the response that is used for most transactions. OKAY indicates that a normal access has been successful. This response can also indicate that an exclusive access has failed. An exclusive access is when more than one master can access a slave at once, but these masters cannot access the same memory range.
01 - EXOKAY	Exclusive access okay. EXOKAY indicates that either the read or write portion of an exclusive access has been successful.
10 – SLVERR	Slave error. SLVERR is used when the access has reached the slave successfully, but the slave wants to return an error condition to the originating master. This indicates an unsuccessful transaction. For example, when there is an unsupported transfer size attempted, or a write access attempted to read-only location.
11 - DECERR	Decode error. DECERR is often generated by an interconnect component to indicate that there is no slave at the transaction address.

5.7 Write data strobes

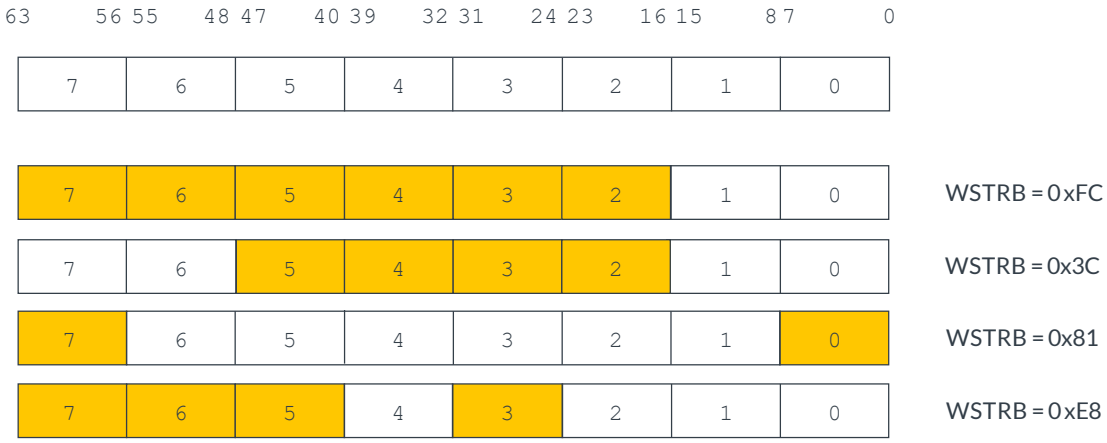
The write data strobe signal is used by a master to tell a slave which bytes of the data bus are required. Write data strobes are useful for cache accesses for efficient movement of sparse data arrays. In addition to using write data strobes, you can optimize data transfers using unaligned start addresses.

The write channel has one strobe bit per byte on the data bus. These bits make the WSTRB signal.

A master must ensure that the write strobes are set to 1 only for byte lanes that contain valid data.

For example, consider a 64-bit write data bus. The WSTRB signal has 8 bits, one for each byte. The following diagram shows how example WSTRB values specify which byte lanes are valid:

For 64 bit WDATA bus



Looking at the first example, we suppose that the valid data are only in the top six significant bytes of the data bus, from byte 7 to byte 2. This means that the master has to control the WSTRB signal with the hexadecimal value `0xFC`.

Similarly, the remaining examples specify valid data bus byte lanes as follows:

- Valid data only in bytes 2, 3, 4, and 5 of the data bus requires a WSTRB signal value of `0x3C`.
- Valid data only in bytes 0 and 7 of the data bus requires a WSTRB signal value of `0x81`.
- Valid data only in bytes 3, 5, 6, and 7 of the data bus requires a WSTRB signal value of `0xE8`.

Byte lane strobes offer efficient movement of sparse data arrays. Using this method, write transactions can be early terminated by setting the remaining transfer byte lane strobes to 0, although the remaining transfers must still be completed. The WSTRB signal can also change between transfers in a transaction.

There is no equivalent signal for the read channel. This is because the master indicates the transfer required and can mask out any unwanted bytes received from the slave.

5.8 Atomic accesses with the lock signal

The AxLOCK signal is used to indicate when atomic accesses are being performed. See Atomic accesses for more information and an explanation of the concept and operation of exclusive access transfers.

The AXI protocol provides two mechanisms to support atomicity:

- Locked accesses

  A locked transfer locks the channel, which remains locked until an unlocked transfer is generated. Locked accesses are similar to the mechanism supported with the AHB protocol.

  When a master uses the AxLOCK signals for a transaction to show that it is a locked transaction, then the interconnect must ensure that only that master can access the targeted slave region,

until an unlocked transaction from the same master completes. An arbiter within the interconnect must enforce this restriction. Because locked accesses require the interconnect to prevent any other transactions occurring while the locked sequence is in progress, they can have an important impact on the interconnect performance.

Locked transactions should only be used for legacy devices. Only AXI3 supports locked accesses. AXI4 does not support locked accesses.

- Exclusive accesses

  Exclusive accesses are more efficient than locked transactions, and they allow multiple masters to access a slave at the same time.

  The exclusive access mechanism enables the implementation of semaphore-type operations, without requiring the bus to remain locked to a particular master during the operation.

Because locked accesses are not as efficient as exclusive accesses, and most components do not require locked transactions, they have been removed from the AXI4 protocol.

In AXI3, the AxLOCK signal consists of two bits with the following values:

- `0b00` - Normal
- `0b01` - Exclusive
- `0b10` - Locked
- `0b11` - Reserved

In AXI4, the AxLOCK signal consists of one bit, with the following values:

- `0b0` - Normal
- `0b1` - Exclusive

5.9 Quality of service

The AXI4 protocol introduces extra signals to support the quality of service (QoS).

Quality of service allows you to prioritize transactions allowing you to improve system performance, by ensuring that more important transactions are dealt with higher priority.

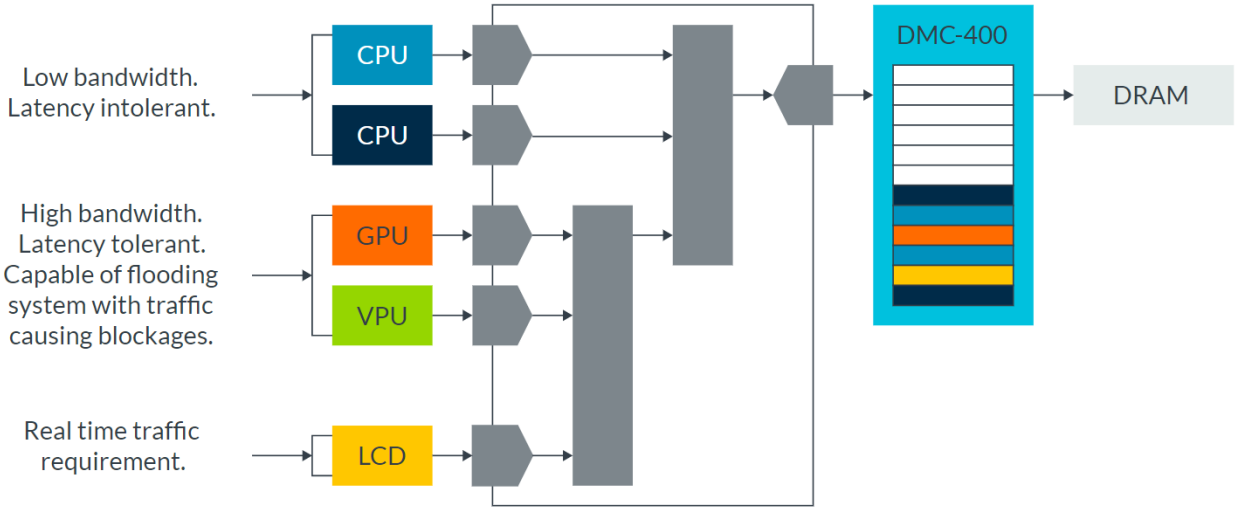
There are two quality of service signals:

- AWQOS is sent on the Write Address channel for each write transaction.
- ARQOS is sent on the Read Address channel for each read transaction.

Both signals are 4 bits wide, where the value `0x0` indicates the lowest priority, and the value `0xF` indicates the highest priority.

The default system-level implementation of quality of service is that any component with a choice of more than one transaction processes the transaction with the higher QoS value first.

The following diagram shows an example system with a Direct Memory Controller (DMC), specifically the DMC-400. This controller manages transactions to DRAM:



In practice, some elements, like the CPU, require memory accesses that are far more important than those of other components, like the GPU or the VPU.

When appropriate QoS values are assigned to transactions, the interconnect can arbitrate higher priority transaction ahead of lower priority transactions and the DMC reorders transactions to ensure that the correct priority is given.

5.10 Region signaling

Region signaling is a new optional feature in AXI4.

When you use region identifiers, it means that a single physical interface on a slave can provide multiple logical interfaces. Each logical interface can have a different location in the system address map.

When the region identifier is used, the slave does not have to support the address decode between the different logical interfaces.

Region signaling uses two 4-bit region identifiers, AWREGION and ARREGION. These region identifiers can uniquely identify up to 16 different regions.

5.11 User signals

The AXI4 interface signal set has the option to include a set of user-defined signals, called the User signals.

User signals can be used on each channel to transfer extra custom control information between master and slave components. These signals are optional and do not have to be supported on all channels. If they are used, then the width of the User signals is defined by the implementation and can be different on each channel.


Note

Because the AXI protocol does not define the functions of these User signals, interoperability issues can arise if two components use the same User signals in a way that is incompatible.

5.12 AXI channel dependencies

The AXI protocol defines dependencies between the different channels.

Three of the main dependencies are as follows:

- WLAST transfer must complete before BVALID is asserted.
  - The master must send all the write data before a write response can be seen by the master.

    This dependency does not exist in AXI3 but is introduced for AXI4:
    - In AXI3, the address does not have to be seen before a write response is sent.
    - In AXI4, all of the data and the address must have been transferred before the master can see a write response.
- RVALID cannot be asserted until ARADDR has been transferred.
  - The slave cannot transfer any read data without it seeing the address first. This is because the slave cannot send data back to the master if it does not know the address that the data will be read from.
- WVALID can assert before AWVALID.
  - A master could use the Write Data channel to send data to the slave, before communicating the address where the slave should write these data.

6 Atomic accesses

An atomic access is a term for a series of accesses to a memory region. Atomic accesses are used by masters when they would like to perform a sequence of accesses to a particular memory region, while being sure that the original data in the region are not corrupted by writes from other masters. This sequence is commonly a read, modify, and write sequence.

There are two types of atomic accesses:

- Locked

  While a master is performing a transaction sequence with locked accesses, accesses from any other masters to the same slave are rejected.

- Exclusive

  When a master successfully performs a transaction sequence with exclusive accesses, other masters can access the slave but not the memory region that is being accessed.

6.1 Locked accesses

Locked transactions should only be used for legacy devices. AXI4 does not support locked transactions, but AXI3 implementations must support locked transactions.

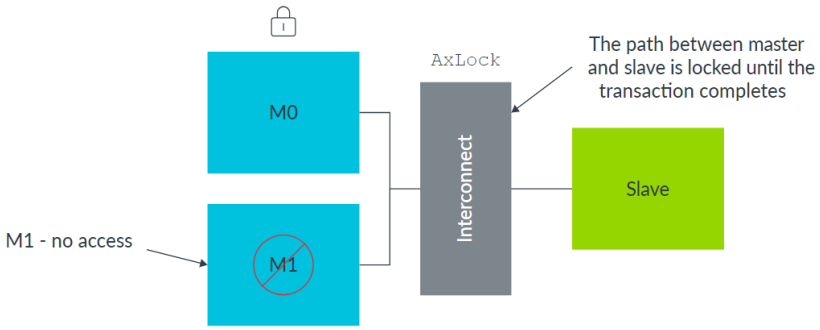
Before a master can start a locked sequence of transactions, it must ensure that it has no other transactions waiting to complete.

A transaction with the AxLOCK signal set indicates a locked transaction. A locked sequence of transactions forces the interconnect to reject access to the slave from any other masters.

The locked sequence must always complete with a final transaction that does not have the AxLOCK signal set. This final transaction is still included in the locked sequence, but effectively removes the lock to allow other masters access to the slave.

Because locked accesses require the interconnect to prevent any other transactions occurring while the locked sequence is in progress, they have an important impact on the interconnect performance.

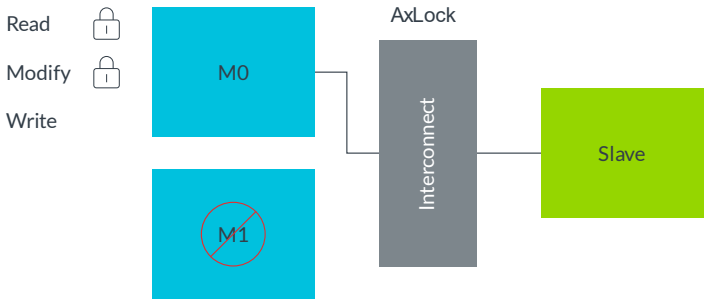
The following diagram shows the AXI locked access operation with an example using two masters, M0 and M1:



Before a master can start a locked sequence of transactions, the master must ensure that it has no other transactions that are waiting to complete.

When M0 uses a lock signal for a transaction to indicate that it is a locked transaction, then the interconnect uses an arbiter to ensure that only M0 can access the targeted slave. The interconnect blocks any accesses from M1 until an unlocked transaction from M0 completes.

The following diagram shows how locked access works with a sequence of transactions:



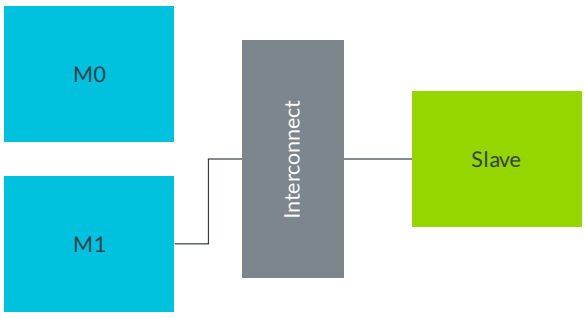
The steps in this example are as follows:

1. Master M0 initiates a sequence of READ, MODIFY, and WRITE.

   The first transaction, READ, has the LOCK signal asserted, indicating that it starts a locked transaction.
2. The interconnect locks out any other transactions.

   From this point, master M1 has no access to the slave.
3. The final transaction in the sequence, WRITE, does not have the LOCK signal asserted. This transaction indicates the end of the locked sequence.

   The interconnect removes the lock, and other masters can now access the slave.



6.2 Exclusive accesses

With AXI 4, exclusive accesses perform atomic operations more efficiently than locked accesses. This is because exclusive accesses use the interconnect bandwidth more effectively.

In an exclusive access sequence, other masters can access the slave at the same time, but only one master will be granted access to the same memory range.

The mechanism that is used for exclusive accesses can provide semaphore-type operations without requiring the bus to remain dedicated to a particular master during the operation. This means that the bus access latency and the maximum achievable bandwidth are not affected.

Exclusive accesses can be composed of more than one data transfer, but all the transactions must have identical address channel attributes.

A hardware exclusive access monitor is required by the slave to record the transaction information for the exclusive sequence so that it knows the memory range that is being accessed and the identity of the master performing the access.

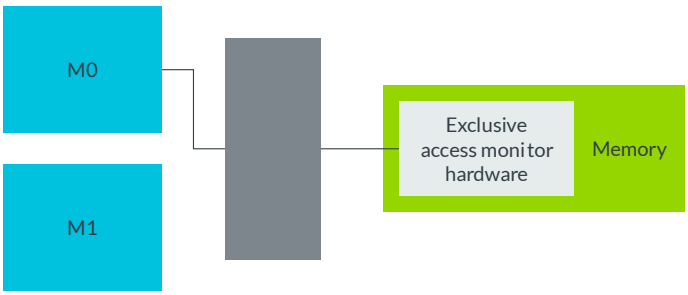
If no other master accesses the monitored range until the exclusive sequence is completed, the access is atomic.

The slave is open to accesses from other masters, resulting in overall increased fairness in bandwidth utilization for the system.

6.3 Exclusive access hardware monitor operation

The basic mechanism of an exclusive access is governed by an exclusive access monitor that you must implement.

The following diagram shows an example where the master M0 performs an exclusive read from an address:



The response from the exclusive access monitor hardware is one of the following:

- EXOKAY

  The value is read, and the ID of the transaction is stored in the exclusive access monitor hardware.
- OKAY

The value is read, but there is no support for exclusive access, and the master should treat this response as an error for the exclusive operation.

At some later time, if EXOKAY was received during the exclusive read, M0 attempts to complete the exclusive sequence by performing an exclusive write to the same address. The exclusive write uses the same transaction ID as the exclusive read.

The response from the exclusive access monitor hardware is one of the following:

- EXOKAY

  No other master has written to that location since the exclusive read access, so the write is successful. In this case, the exclusive write updates memory.

- OKAY

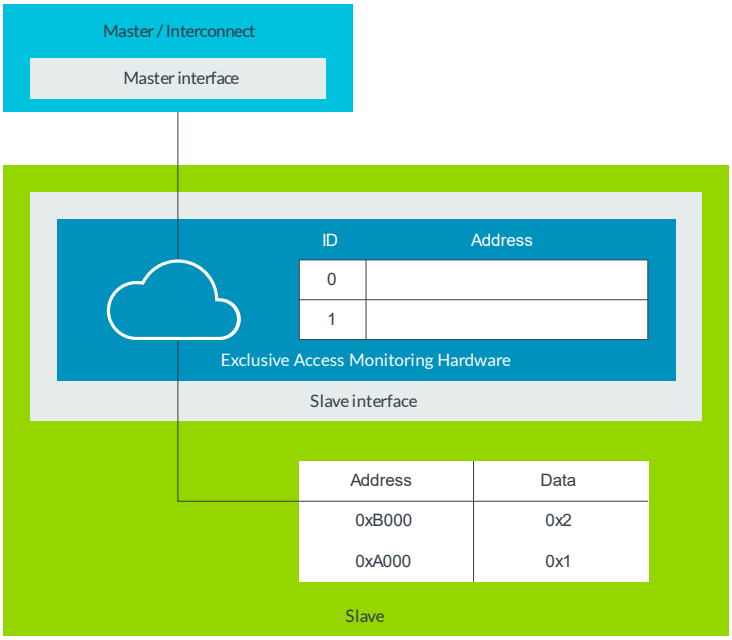
  Another master, for example M1, has written to the location since the exclusive read access, so the write fails. In this case, the memory location is not updated.

Some slaves require extra logic to support exclusive access. The exclusive access monitoring hardware monitors only one address for each transaction ID. It should be implemented so that it can monitor every possible exclusive ID that can be seen.

6.4 Exclusive transaction pairs: both pass

This section describes an example of two successful exclusive access sequences that both pass.

The following diagram shows a system containing a master, with its AXI master interface, and a slave:



The slave interface includes exclusive access monitoring hardware that can save the ID and the address accessed for each transaction.

The following table describes the different transactions in the example sequence. All transactions in the table are exclusive accesses:

Transaction number	Read or write	Transaction ID	Address	Data	xRESP
1	R	0	0xA000	0x1	EXOKAY
2	R	1	0xB000	0x2	EXOKAY
3	W	0	0xA000	0x3	EXOKAY
4	W	1	0xB000	0x4	EXOKAY

The transaction sequence shown in the previous table proceeds as follows:

1. The first transaction is the master, which performs a read exclusive transaction with ID 0 from address 0xA000.

   The exclusive access monitoring hardware saves the ID and address of this transaction in its table, and the slave responds with the read data, 0x1.

   Because exclusive accesses are correctly supported for this slave, the exclusive access monitoring hardware responds with an EXOKAY response.
2. Next, the master performs a new read exclusive transaction with ID 1 from address 0xB000.

   Again, the exclusive access monitoring hardware saves the details of this new transaction in the table, and the slave responds with the read data, 0x2.

   Because exclusive accesses are correctly supported for this slave, the exclusive access monitoring hardware again responds with an EXOKAY response.

   At this moment in our example there are two separate exclusive sequences ongoing.
3. After the master has completed its operation, it performs a write exclusive transaction with ID 0 to address 0xA000.

   The exclusive access monitoring hardware checks the detail of this transaction in the table and, because of the existing record with ID 0 and address 0xA000, responds to the master with an EXOKAY response. This means that no other master has accessed this memory location, and the slave updates it with the new value it receives, which in this example is 0x3.

   The exclusive access monitoring hardware removes the ID and address for this transaction from its table, because the exclusive access sequence for that address location is now complete.
4. Finally, the master performs a new write exclusive transaction with ID 1 to address 0xB000.

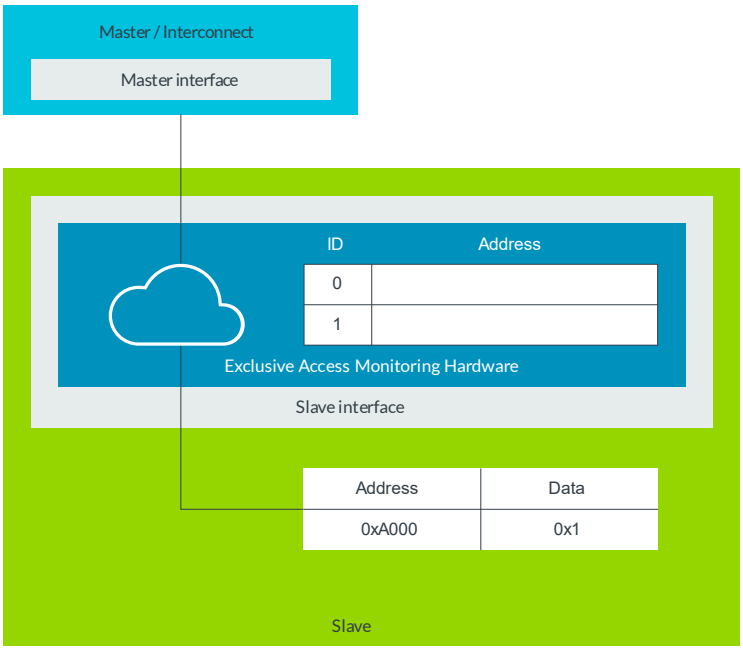
   The exclusive access monitoring hardware checks the detail of this transaction in its table. Seeing an existing record with ID 1 and address 0xB000, it again responds to the master with an EXOKAY response. This means that no other master has accessed this memory location, and the slave updates it with the new value received, which in our example is 0x4.

Again, the exclusive access monitoring hardware removes the ID and address for this transaction from its table, because the exclusive access sequence for that address location is now complete.

6.5 Exclusive transaction pairs: one pass, one fail

This section describes an example of two exclusive access sequences, where the first one succeeds and the second one fails.

The following diagram shows a system containing a master, with its AXI master interface, and a slave:



The slave interface includes exclusive access monitoring hardware that can save the ID and the address accessed for each transaction.

The following table describes the different transactions in the example sequence. All transactions in the table are exclusive accesses:

Transaction number	Read or write	Transaction ID	Address	Data	xRESP
1	R	0	`0xA000`	`0x1`	EXOKAY
2	R	1	`0xA000`	`0x1`	EXOKAY
3	W	0	`0xA000`	`0x3`	EXOKAY
4	W	1	`0xA000`	`0x4`	OKAY

The transaction sequence shown in the previous table proceeds as follows:

1. The first transaction is the master performing a read exclusive transaction with ID 0 from address `0xA000`.

   The exclusive access monitoring hardware saves the ID and address of this transaction in its table, and the slave responds with the read data, `0x1`.

Because exclusive accesses are correctly supported for this slave, the exclusive access monitoring hardware responds with an EXOKAY response.

2. Later, the master performs a new read exclusive transaction with ID 1 from the same address as the first transaction, `0xA000`.

   The exclusive access monitoring hardware saves the detail of this new transaction in the table, and the slave responds with the read data, `0x1`.

   Again, because exclusive accesses are correctly supported for this slave, the exclusive access monitoring hardware responds with an EXOKAY response.

   At this moment in our example, we have two different ongoing exclusive sequences to the same memory location.

3. After the master has completed its operation, it performs an exclusive write transaction with ID 0 to address `0xA000`.

   The exclusive access monitoring hardware checks the detail of this transaction in its table and, seeing a record with ID 0 and address `0xA000`, responds to the master with an EXOKAY response. This means that no other master has updated this memory location, and the slave can update it with the new value received, which in our example is `0x3`.

   Because the content of the address location `0xA000` has been modified, the exclusive access monitoring hardware removes from its table all the entries that match that location address.

4. Finally, the master performs a new write exclusive transaction with ID 1 again to address `0xA000`.

   The exclusive access monitoring hardware checks the detail of this transaction in its table. Not finding any records with the address `0xA000`, it responds with an OKAY response.

   The OKAY response means that a previous write operation has been performed on this memory location which updated the data. In this case, the slave cannot update the memory location with the new value, `0x4`.

   This situation is an exclusive access failure. In this case, the master must restart the full exclusive access sequence beginning with the exclusive read and then the exclusive write again.

This example demonstrates how exclusive accesses implement non-blocking behavior. It is this behavior that provides greater system throughput when compared with LOCK accesses.

7 Transfer behavior and transaction ordering

This section of the guide analyzes some example sequences of read and write transactions, to help you understand the relationships between the different AXI channels. This section also explains some of the rules that govern transactions and how transfer IDs can support out-of-order transactions.

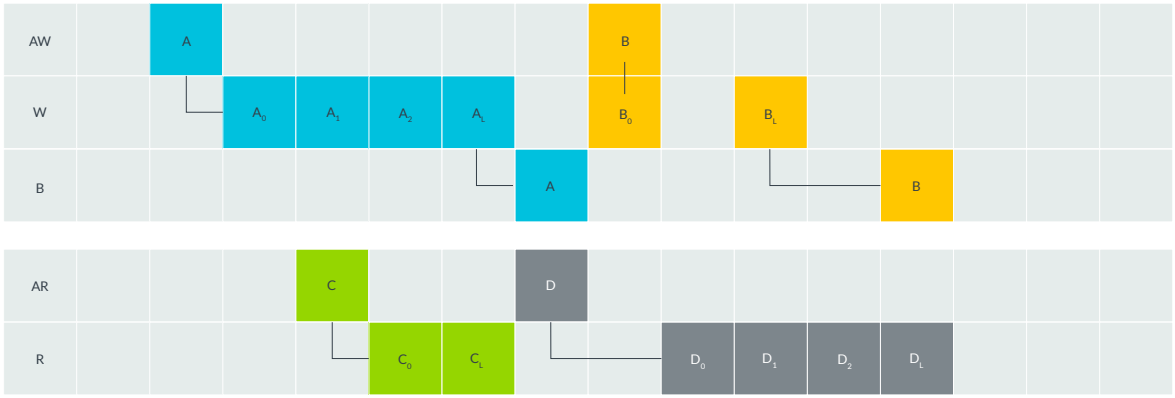
We will also look at:

- Unaligned transfers, and how they help optimize bandwidth utilization
- The differences between big-endian and little-endian encoding, with some simple examples
- The main parameters that are related to the AXI interfaces. These parameters are useful when implementing an interconnect

7.1 Examples of simple transactions

Examples of simple transactions help to explain the relationships between the different AXI channels.

The following diagram shows a time representation of several valid transactions on the five channels of an AXI3 or AXI4 interface:



The different transactions in this example are as follows:

1. Transaction A, which is a write transaction that contains four transfers.

   The master first puts the address A on the AW channel, then soon puts the sequence of four data transfers on the W channel, ending with AL where L stands for last.

   Once all four data transfers complete, the slave responds on the B channel.

2. While transaction A was occurring, the master also used the read channels to perform a read transaction, C, which contains two transfers.

Because this is a read transaction, there is no response from the slave on a different channel when the transaction completes. Instead, the response from the slave is included in the R channel at the same time as the data.

3. Once transaction C completes, the master uses the Read Address channel AR to send a new read address, D, to the slave.

   In this case, the response from the slave is not immediate. This is indicated by the empty time slot between D and D0. Delays like this can happen. The slave is not obliged to answer immediately. For example, the slave could be busy performing another operation, or it could take time to retrieve the data.

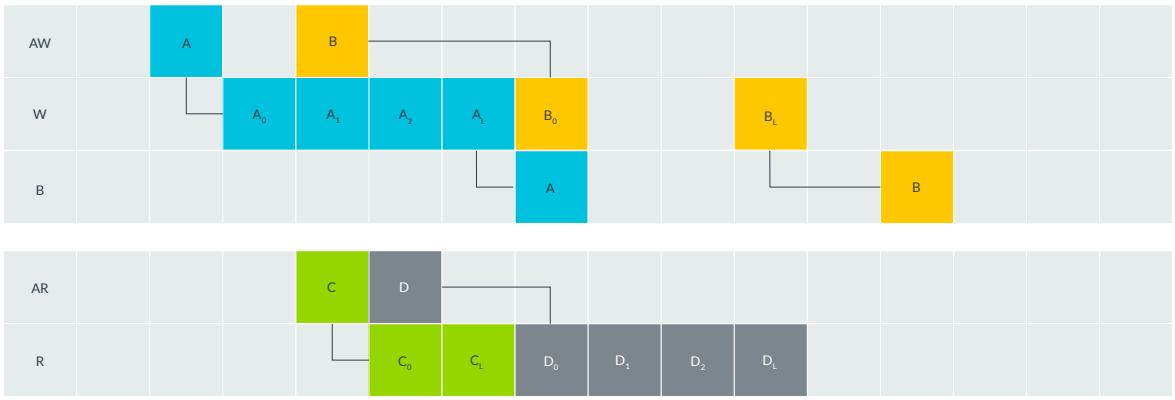
   Eventually, the slave responds with four sequential transfers, D0 through DL, on the R channel.

4. Finally, while the read transaction D is ongoing, the master uses the Write Address channel, AW, to send a new address, B, to the slave for a write operation.

   The master puts the data B0 on the W channel at the same time as it puts the corresponding address B on the AW channel. There is a delay in this example between data transfers B0 and BL, and another delay before the response B. The transaction completes only when the slave sends the response to the master.

All of these examples are valid transactions.

The following diagram shows the same sequence of read and write transactions in a different, but still valid, timeline:



In this example, the master starts transaction B before it has finished transaction A.

The master uses the Write Address channel, AW, to start a new transaction by transferring a new address B to the slave before it has finished transferring the data for transaction A on the W channel.

The data for transaction B is transferred to the slave when all the data for transaction A have completed. The master does not wait for a response on the B channel for transaction A before it starts to transfer the data for transaction B.

At the same time, the master uses the Read Address channel to transfer in sequence the read addresses C and D for the slave. The slave responds in sequence to the two read requests.

This example shows a different valid combination of read and write transactions happening on the different channels. This shows the flexibility of the AXI protocol and the possibility to optimize the interconnect performance.

7.2 Transfer IDs

The AXI protocol defines an ID signals bus for each channel. Marking each transaction with an ID gives the possibility to complete transactions out of order. This means that transactions to faster memory regions can complete without waiting for earlier transactions to slower memory regions. The use of transfer IDs enables the implementation of a high-performance interconnect, maximizing data throughput and system efficiency. This feature can also improve system performance because it reduces the effect of transaction latency.

The ID signal buses are as follows:

- AWID
- WID
- BID
- ARID
- RID

The AXI protocol supports out-of-order transactions by enabling each interface to act as multiple ordered interfaces. According to the AXI protocol specifications, all transactions with a given ID must be ordered. However, there is no restriction on the ordering of transactions with different IDs.

When working with transfer IDs, follow these rules:

- All transfers must have an ID.
- All transfers in a transaction must have the same ID.
- Masters can support multiple IDs for multiple threads.
- Slaves generally need a configurable ID width.

You should also remember these two important AXI parameters for ID signals:

- The write ID width, which is the number of bits used for the AWID, WID and BID buses
- The read ID width, which is the number of bits used for the ARID and RID buses

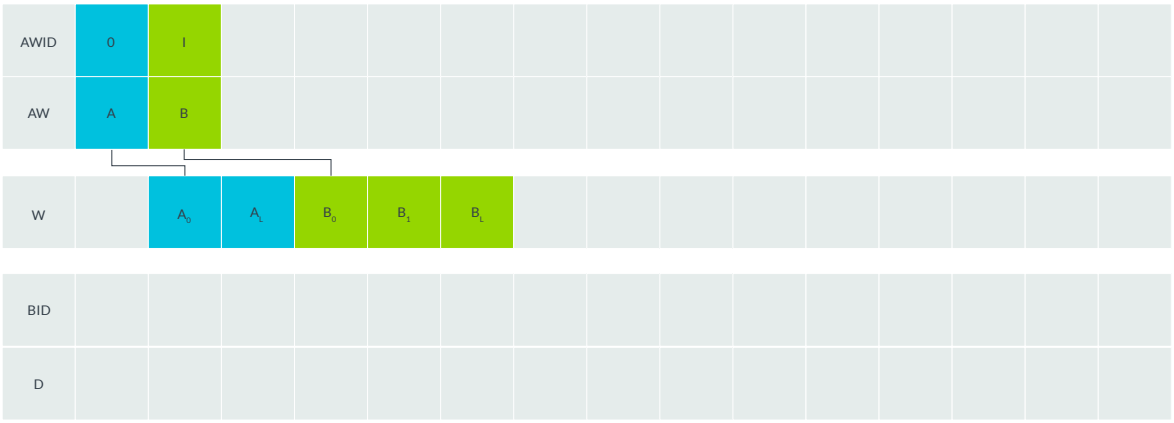
7.3 Write transaction ordering rules

There are three AXI ordering rules for write transactions.

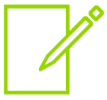
The rules are as follows:

- Write data on the W channel must follow the same order as the address transfers on the AW channel.

  The following diagram illustrates this rule:



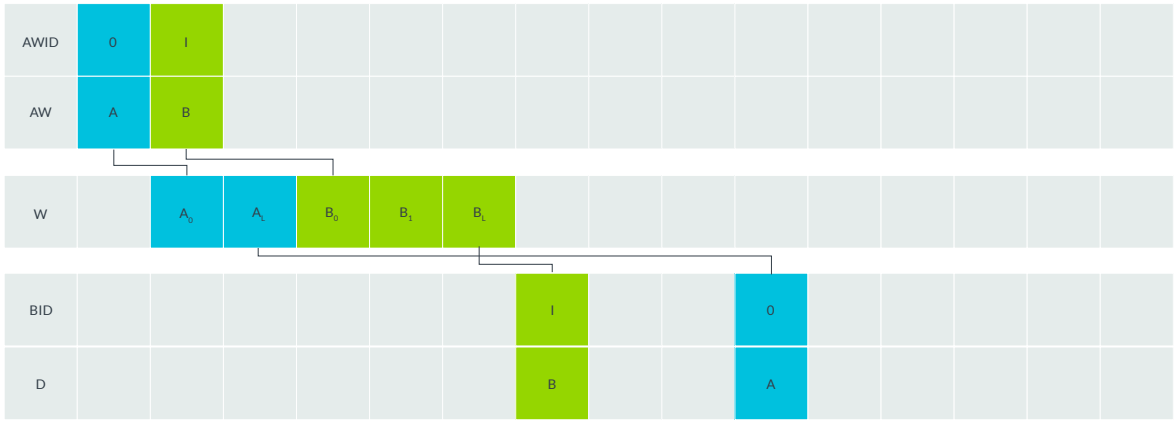
  In this example, the master issues address A then B, so data must start with A0 before B0.


Note

The interleaving of write data with different IDs on the W channel was permitted in AXI3, but is deprecated in AXI4 and later.

- Transactions with different IDs can complete in any order.

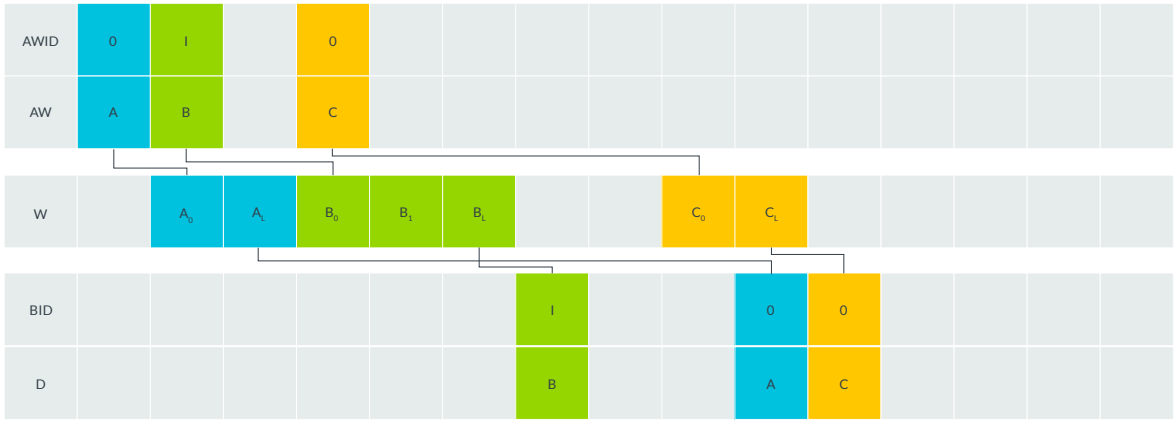
  The following diagram illustrates this rule:



In this example, transaction B completes before transaction A, even though transaction A started first.

- A master can have multiple outstanding transactions with the same ID, but they must be performed in order and complete in order.

  The following diagram illustrates this rule:



In this example, transaction B has a different ID from the other transactions, so it can complete at any point. However, transactions A and C have the same ID, so they must complete in the same order as they were issued: A first, then C.

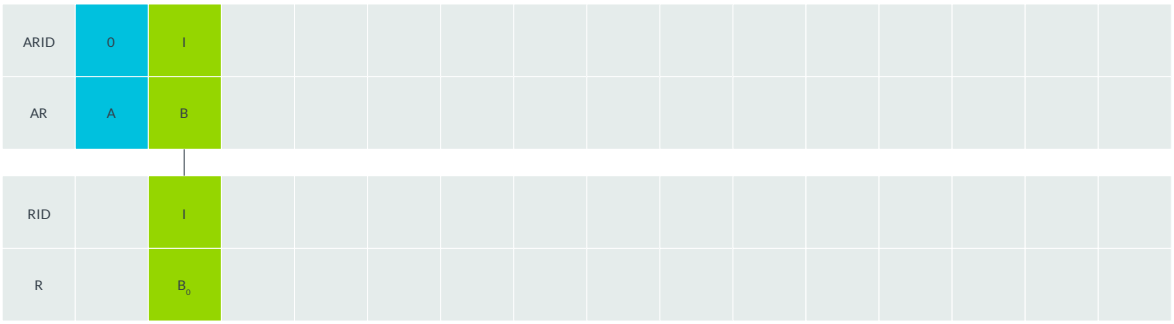
7.4 Read transaction ordering rules

There are three ordering rules for read transactions.

The rules are as follows:

- Read data for different IDs on the R channel has no ordering restrictions. This means that the slave can send it in any order.

  The following diagram shows an example where transaction B is serviced before A, even though the address for transaction A is received first:

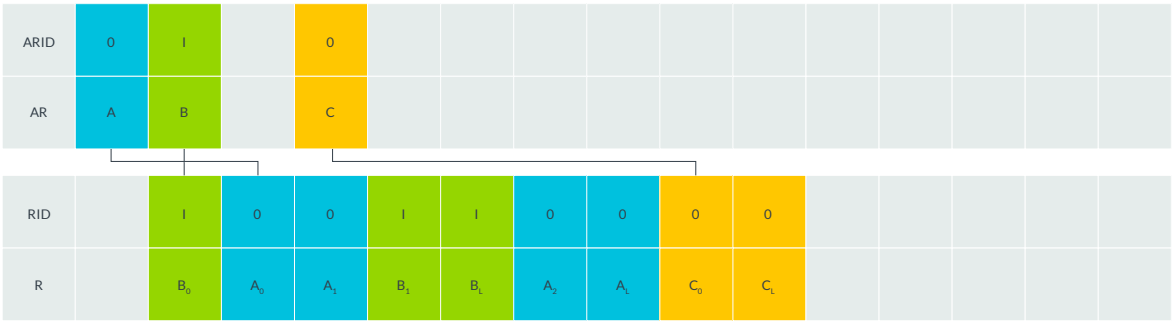


- The read data for the different IDs on the R channel can be interleaved, with the RID value differentiating which transaction the data relates to.

  The following diagram shows an example where R data for transactions A and B are interleaved:

- For transactions with the same ID, read data on the R channel must be returned in the order that they were requested.

  The following diagram shows an example where transactions A and C have the same RID value of 0:

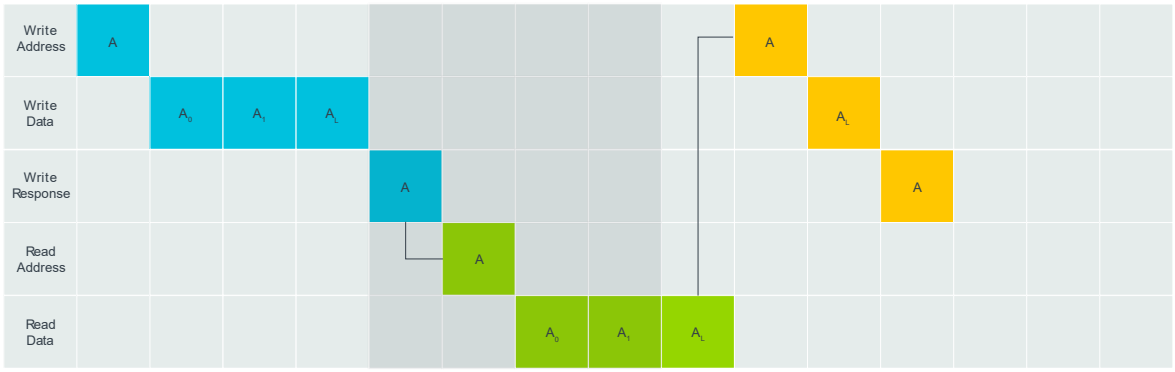


Because transaction A was requested before transaction C, the slave must return all four R data values for A before the data values for C.

7.5 Read and write channel ordering

Read and write channels have no ordering rules in relation to each other. This means that they can complete in any order. So, if a master requires ordering for a specific sequence of reads and writes, the master must ensure that the transaction order is respected by explicitly waiting for transactions to complete before issuing new ones.

The following diagram shows an example where the master requires a specific ordering for a write-read-write transaction sequence from an address:



The sequence of operations is as follows:

1. The master starts the first write transaction.
2. The master ensures that the slave has completed the write transaction by waiting for the signal on the Write Response channel.
3. The master starts the read transaction.
4. The master waits for the final response on the Read Data channel.

5. The master starts the second transaction.

7.6 Unaligned transfer start address

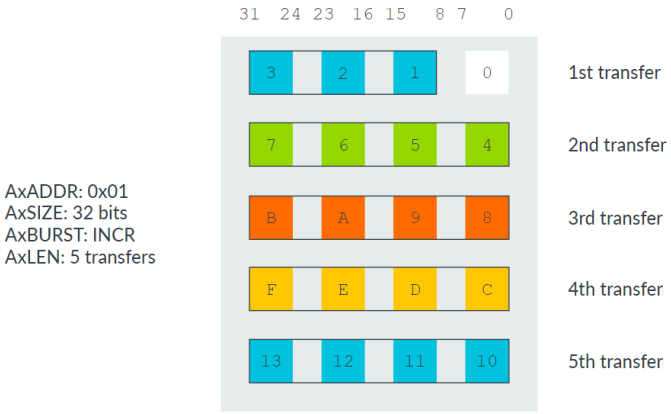
The AXI protocol supports transactions with an unaligned start address that only affects the first transfer in a transaction. After the first transfer in a transaction, all other transfers are aligned.


Note

The AXI protocol also supports unaligned transfers using the strobe signals. See Write data strobes for more information.

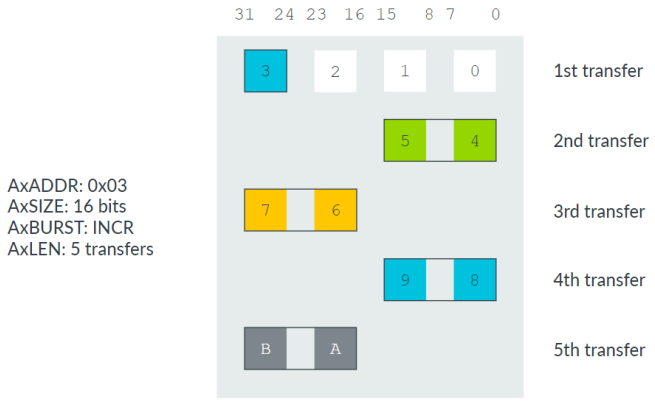
An unaligned transfer is where the AxADDR values do not have to be aligned to the width of the transaction. For example, a 32-bit data packet that starts at a byte address of `0x1002` is not aligned to the natural 32-bit address boundary because `0x1002` is not exactly divisible by `0x20`.

The following example shows a 5-beat 32-bit transfer starting at an unaligned address of `0x01`:



If the transaction were aligned to a start address of `0x00`, the result would be a five-beat burst with a width of four bytes giving a maximum data transfer of 20 bytes. However, we have an unaligned start address of `0x1`. This reduces the total data volume of the transfer, but it does not mean a final unaligned transfer to complete the burst and make up the volume. In this example, the first transfer starts at address `0x01` and contains three bytes. All the following transfers in the burst are aligned with the bus width and are composed of four bytes each.

The following example shows a five-beat 16-bit-sized transaction starting at address `0x03`:



If the transaction were aligned to a start address of `0x00`, the result would be a five-beat burst with a width of two bytes giving a maximum data transfer of 10 bytes. In this example, the first transfer starts at an unaligned address of `0x03` and contains one byte. All the following transfers in the burst are aligned with the bus width and are composed of two bytes each.

The AXI protocol does not require the slave to take special action based on any alignment information from the master.

7.7 Endianness support

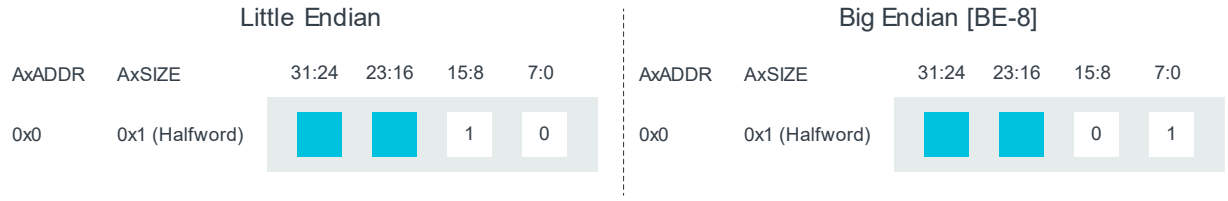
The AXI protocol supports mixed-endian structures in the same memory space by using Big Endian-8 (BE-8) mode. Compared to little-endian mode, the same byte lanes are used in BE-8 mode, but the order of the bytes is reversed.

Mixed-endian structures using BE-32 are more complicated than those using BE-8, because byte lanes are not the same as little-endian mode.

The following example shows both little-endian and big-endian representations of the same four-byte word:

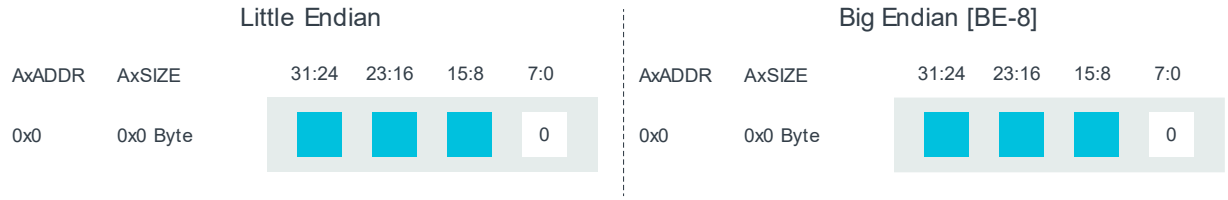
For a four-byte word in little-endian mode, the most significant byte uses the most significant byte lane, which is byte lane 3. In BE-8 mode, the most significant byte uses the least significant byte lane, which is byte lane 0.

The following example shows both little-endian and big-endian representations of the same two-byte word:



For a halfword of two bytes in little-endian mode, the most significant byte uses byte lane 1, and the least significant byte uses byte lane 0. Again, in big-endian BE-8 mode, the lanes that are used by the two bytes are switched. The most significant byte uses byte lane 0, and least significant byte uses byte lane 1.

Finally, for a single byte, there is no difference between little-endian and big-endian mode, as shown in the following example:



In both cases, the byte uses byte lane 0.

In a configurable endianness component like an Arm core, which supports BE-8, the reordering of the bytes should be performed internally, so that nothing has to be done at the interconnect level. On the other hand, a custom device that is connected to the AXI interconnect, which is BE-8 by nature, would already have the correct order of bytes. Having BE-8 in the AXI protocol eases the support for dynamic endianness switching.

7.8 Read and write interface attributes

This section of the guide highlights some of the most important attributes for configuring AXI write and read channels.

The write interface attributes include the following:

- Write issuing capability

  Represents the maximum number of active write transactions the master interface can generate
- Write interleave capability (AXI3 only)

  The number of active write transactions for which the master interface is capable of transmitting data.

- Write acceptance capability (AXI3 only)

  Represents the maximum number of active write transactions the slave interface can accept
- Write interleave depth attribute

  Represents the number of active write transactions that the slave interface can receive data from

The read interface attributes include the following:

- Read issuing capability attribute

  Represents the maximum number of active read transactions that a master interface can generate
- Read acceptance capability

  The maximum number of active read transactions that a slave interface can accept
- Read data reordering depth

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

8 Check your knowledge

Q: What burst type must a master issue if it wants to write to a FIFO: fixed, wrapping, or incrementing?

A: Fixed. A FIFO works by writing to and reading from a fixed address.

Q: All AXI4 channels share the same handshake mechanism. The VALID signal goes from the source to the destination to indicate when valid information is available. Which signal goes from the destination to the source to indicate when it can accept information?

A: The READY signal.

Q: Which signals can be used to protect against illegal transactions downstream in the system?

A: The AWPROT and ARPROT signals

Q: What is the purpose of transfer IDs?

A: Marking transactions with different IDs allows transactions with different IDs to complete out of order. This means that transactions to faster memory regions can complete without waiting for earlier transactions to slower memory regions.

9 Related information

Here are some resources related to material in this guide:

- AMBA specifications
- AMBA on Arm developer
- Arm video tutorials:
  - AXI channels
  - AXI's main features
  - The AXI protocol
  - The AXI protocol in a multi-master system design
  - Introduction to the AMBA AXI protocol
  - What is AMBA, and why use it?

10 Next steps

This guide has provided an overview of the main topics relating to AMBA AXI, including the use and operation of the different channels and signals.

This knowledge will be useful as you learn more about AMBA AXI by reading the AMBA AXI and ACE protocol specification. You can put your knowledge into action to develop interfaces that implement the AMBA AXI protocol.