AMBA AXI Protocol

Revision: r0p0

Specification

ARM®

ARM IHI 0022A

AMBA AXI Protocol
Specification

Release Information

Change history

Date	Issue	Change
16 June, 2003	A	First release

Proprietary Notice

AMBA Specification License

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is Final (information on a developed product).

Web Address

`http://www.arm.com`

Contents
AMBA AXI Protocol Specification

List of Tables
AMBA AXI Protocol Specification

List of Figures
AMBA AXI Protocol Specification

Preface

This preface introduces the *AMBA Advanced eXtensible Interface (AXI) Protocol Specification* and its reference documentation. It contains the following sections:

About this document

This is the AXI specification.

Intended audience

This manual is written to help hardware and software engineers who are familiar with the *Advanced Microcontroller Bus Architecture* (AMBA) to design systems and modules that are compatible with the AXI.

Using this manual

This manual is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter to learn about AXI architecture and basic transactions defined by the AXI protocol.

Chapter 2 ***Signal Descriptions***

Refer to this chapter for definitions of the AXI global, address, read, write, and write response signals.

Chapter 3 ***Channel Handshake***

Read this chapter to learn about the AXI channel handshake process.

Chapter 4 ***Addressing Options***

Read this chapter to learn about AXI burst types and how to calculate addresses and byte lanes for transfers within a burst.

Chapter 5 ***Additional Control Information***

Read this chapter to learn how to configure the AXI to support system level caches and protection units.

Chapter 6 ***Atomic Accesses***

Read this chapter to learn how to perform exclusive accesses and locked accesses.

Chapter 7 ***Response Signaling***

Read this chapter to learn about the four transaction responses of AXI slaves.

Chapter 8 ***Ordering Model***

Read this chapter to learn how the AXI uses transaction ID tags to enable out-of-order transaction processing.

Chapter 9 ***Data Buses***

Read this chapter to learn how to do transactions of varying sizes on the AXI read and write data buses and how to use byte-invariant endianness to handle mixed-endian data.

Chapter 10 ***Unaligned Transfers***

Read this chapter to learn how the AXI handles unaligned transfers.

Chapter 11 ***Clock and Reset***

Read this chapter to learn about the timing of the AXI clock and reset signals.

Chapter 12 ***Low-power Interface***

Read this chapter to learn how to use the AXI clock control interface to enter into and exit from a low-power state.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this document, where:

r*n* identifies the major revision of the product.

p*n* identifies the minor revision or modification status of the product.

Typographical conventions

The following typographical conventions are used in this book:

bold Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists where appropriate.

italic Highlights important notes, special terminology, internal cross-references, and citations.

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic

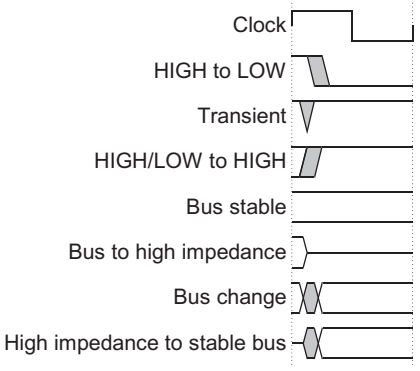
Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

monospace bold

Denotes language keywords when used outside example code and ARM processor signal names.

Timing diagram conventions

This manual contains timing diagrams. The figure below explains the components used in these diagrams. Any variations are clearly labeled when they occur. Therefore, no additional meaning must be attached unless specifically stated.



Key to timing diagram conventions

Shaded bus and signal areas are undefined, so the bus or signal can assume any value in the shaded area at that time. The level is unimportant and does not affect normal operation.

Further reading

This section lists publications by ARM Limited.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

This document contains information that is specific to the AXI. Refer to the following documents for other relevant information:

- *AMBA Specification (Rev 2.0)* (ARM IHI 0011)
- *ARM Architecture Reference Manual* (ARM DDI 0100).

Feedback

ARM Limited welcomes feedback both on the AXI and on the documentation.

Feedback on the AXI

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this document

If you have any comments about this manual, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter describes the architecture of the AXI and basic transactions defined by the AXI protocol. It contains the following sections:

1.1 About the AXI

AXI is the latest generation AMBA interface. It is targeted at high-performance, high-frequency system designs and includes a number of features that make it suitable for a high-speed submicron interconnect.

The objectives of the latest generation AMBA interface are to:

- be suitable for high-bandwidth and low-latency designs
- enable high-frequency operation without using complex bridges
- meet the interface requirements of a wide range of components
- be suitable for memory controllers with high initial access latency
- provide flexibility in the implementation of interconnect architectures
- be backward-compatible with existing AHB and APB interfaces.

The key features of the AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers using byte strobes
- burst-based transactions with only start address issued
- separate read and write data channels to enable low-cost *Direct Memory Access* (DMA)
- ability to issue multiple outstanding addresses
- out-of-order transaction completion
- easy addition of register stages to provide timing closure.

As well as the basic data transfer protocol, the AXI protocol includes optional extensions to cover signaling for low-power operation.

1.2 Architecture

The AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred. The data is transferred between master and slave using a write channel to the slave or a read channel to the master. In write transactions, in which all the data flows from the master to the slave, the AXI has an additional write response channel to allow the slave to signal to the master the completion of the write transaction.

The AXI protocol allows address information to be issued ahead of the actual data transfer and enables support for multiple outstanding transactions as well as out-of-order completion of transactions.

Figure 1-1 shows how a read transaction uses the address and read channels.

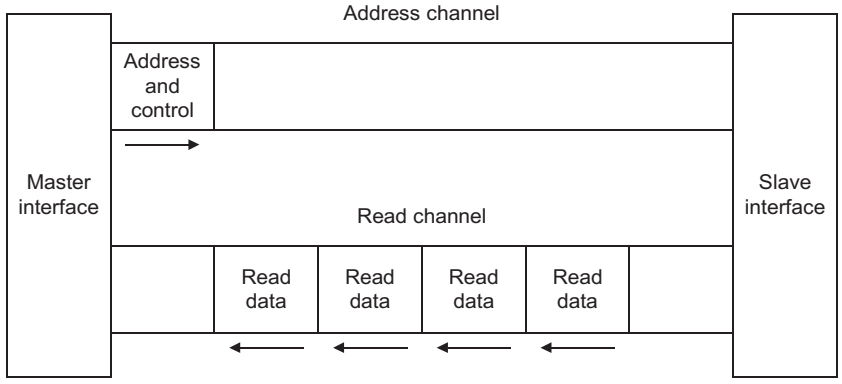


Figure 1-1 Channel architecture of reads

Figure 1-2 on page 1-4 shows how a write transaction uses the address, write, and write response channels.

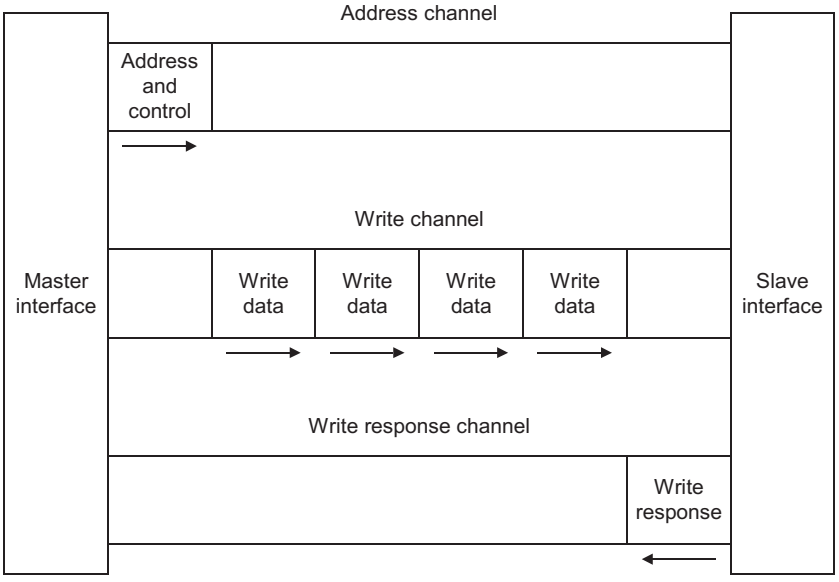


Figure 1-2 Channel architecture of writes

1.2.1 Channel definition

Each of the four independent channels consists of a set of information signals and uses a two-way **VALID** and **READY** handshake mechanism.

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

Address channel

The address channel is used in every transaction and carries all the required address and control information for that transaction. The AXI supports the following mechanisms:

- variable-length bursts, from 1 to 16 data transfers per burst
- bursts with a transfer size of eight bits up to the maximum data bus width
- wrapping, incrementing, and non-incrementing bursts
- atomic operations, using exclusive and locked access
- system-level caching and buffering control
- secure and privileged access.

Read channel

The read channel conveys both the read data and any read response information from the slave back to the master. The read channel includes:

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

Write channel

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

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

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

Write response channel

The write response channel provides a way for the slave to respond to write transactions. All write transactions use completion signaling.

Note

The completion signal occurs once for each burst, not for each individual data transfer within the burst.

1.2.2 Interface and interconnect

A typical system consists of a number of master and slave devices connected together through some form of interconnect, as shown in Figure 1-3.

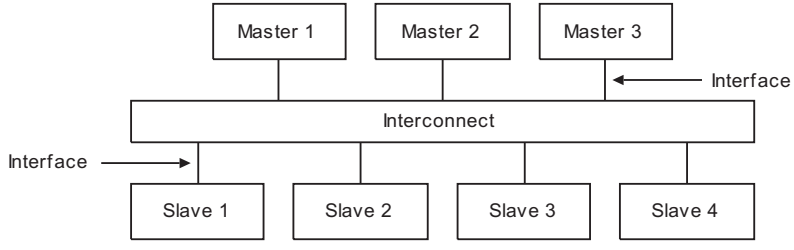


Figure 1-3 Interface and interconnect

The AXI protocol provides a single interface definition for describing interfaces:

- between a master and the interconnect
- between a slave and the interconnect
- between a master and a slave.

The interface definition enables a variety of different interconnect implementations. The interconnect between devices is equivalent to another device with symmetrical master and slave ports to which real master and slave devices can be connected.

Most systems use one of three interconnect approaches:

- shared address and data buses
- shared address buses and multiple data buses
- multilayer, with multiple address and data buses.

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

1.2.3 Register slices

Each AXI channel transfers information in only one direction, and there is no requirement for a fixed relationship between the various channels. This is important because it enables the insertion of a register stage in any channel, at the cost of an additional cycle of latency. This makes possible a trade-off between cycles of latency and maximum frequency of operation.

It is also possible to use register slices at almost any point within a given interconnect. It can be advantageous to use a direct, fast connection between a processor and high-performance memory, but to use simple register slices to isolate a longer path to less performance-critical peripherals.

1.3 Basic transactions

This section gives examples of basic AXI transactions. Each example shows the **VALID** and **READY** handshake mechanism. Transfer of either address information or data occurs when both the **VALID** and **READY** signals are HIGH.

Figure 1-4 shows a read burst of four transfers. In this example, the master drives the address, and the slave accepts it one cycle later.

Note

The master also drives a set of control signals showing the length and type of the burst, but these signals are omitted from the figure for clarity.

After the address appears on the address bus, the data transfer occurs on the read channel. The slave keeps the **VALID** signal LOW until the read data is available. For the final data transfer of the burst, the slave asserts the **RLAST** signal to show that the last data item is being transferred.

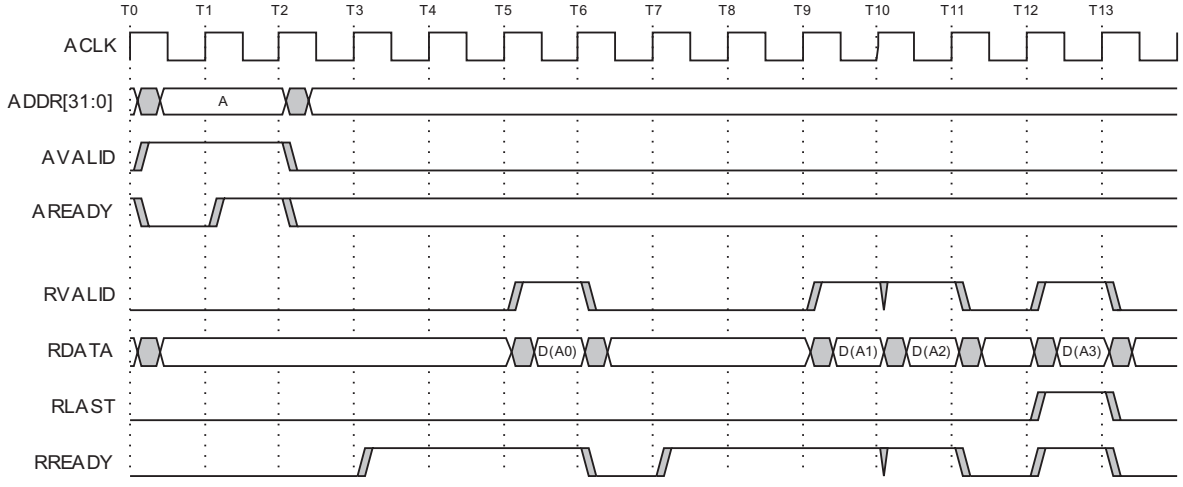


Figure 1-4 Read burst

Figure 1-5 on page 1-8 shows how a master can drive another burst address after the slave accepts the first address. This enables a slave to begin processing data for the second burst in parallel with the completion of the first burst.

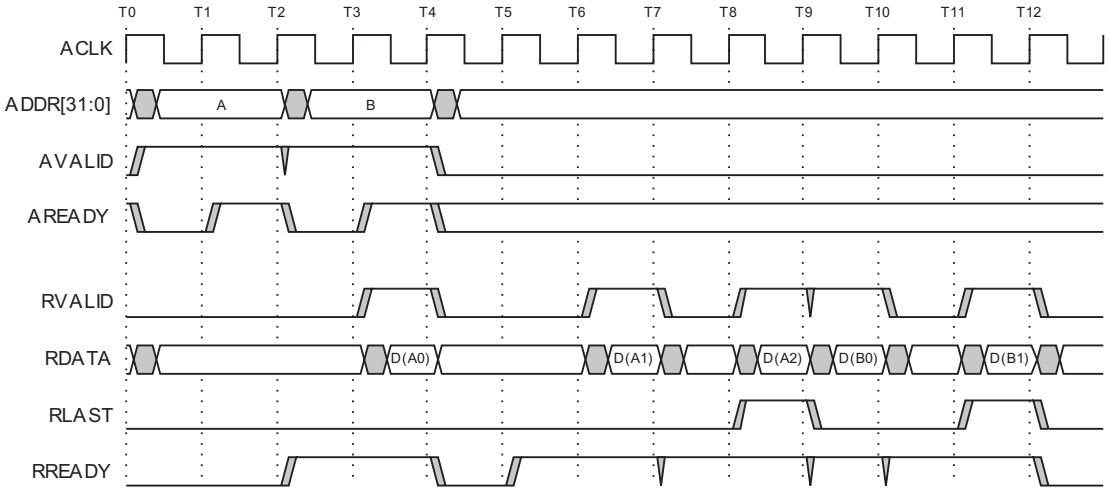


Figure 1-5 Overlapping read bursts

Figure 1-6 on page 1-9 shows a write transaction. The process starts when the master sends an address and control information on the address channel. The master then sends each item of write data over the write channel. When the master sends the last data item, the **WLAST** signal goes HIGH. When the slave has accepted all the data items, it drives a write response back to the master to indicate that the write transaction is complete.

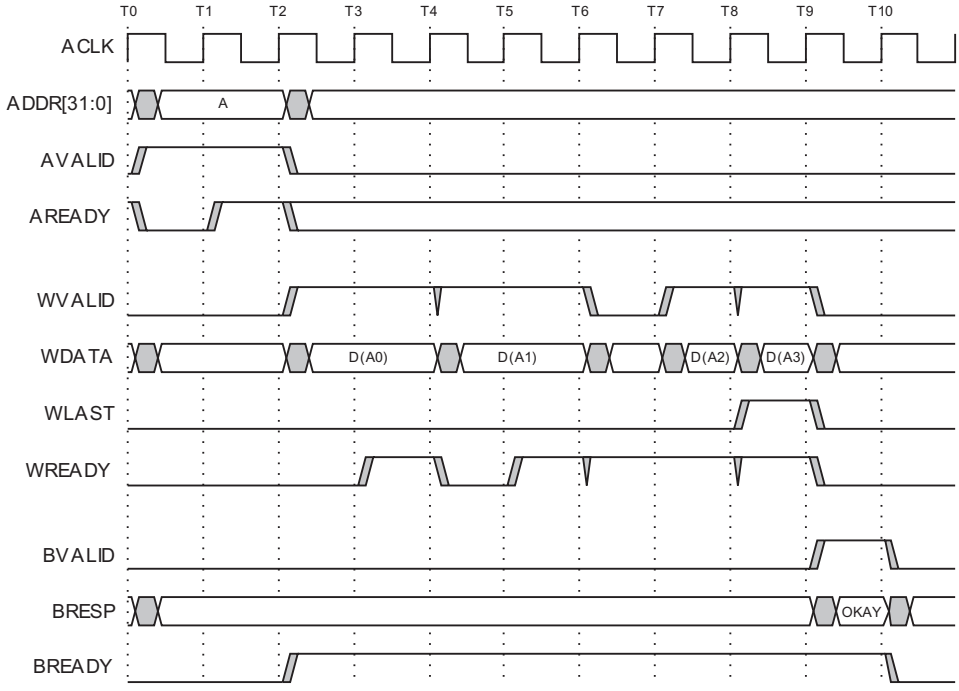


Figure 1-6 Write burst

1.3.1 Transaction ordering

The AXI protocol enables out-of-order transaction completion. The AXI protocol gives an ID tag to every transaction across the interface. The protocol requires that transactions with the same ID tag are completed in order, but transactions with different ID tags can be completed out of order.

Out-of-order transactions can improve system performance in two ways:

- The interconnect can enable transactions with fast-responding slaves to complete in advance of earlier transactions with slower slaves.
- Complex slaves can return read data out of order. For example, a data item for a later access might be available from an internal buffer before the data for an earlier access is available.

If a master requires that all transactions are completed in the same order that they are issued, then they must all have the same ID tag. If, however, a master does not require in-order transaction completion, it can supply the transactions with different ID tags, enabling them to be completed in any order.

In a multimaster system, the interconnect is responsible for appending additional information to the ID tag to ensure that ID tags from all masters are unique. The ID tag is similar to a master number, but with the extension that each master can implement multiple virtual masters within the same port by supplying an ID tag to indicate the virtual master number.

Although complex devices can make use of the out-of-order facility, simple devices are not required to use it. Simple masters can issue every transaction with the same ID tag, and simple slaves can respond to every transaction in order, irrespective of the ID tag.

1.4 Additional features

The AXI also supports the following additional features:

Burst types

The AXI supports three different burst types that are suitable for:

- normal memory accesses
- wrapping cache line bursts
- streaming data to peripheral FIFO locations.

See Chapter 4 *Addressing Options*.

System cache support

The cache-support signal of the AXI enables a master to provide to a system-level cache the bufferable, cacheable, and allocate attributes of a transaction.

See *Cache support* on page 5-2.

Protection unit support

To enable both privileged and secure accesses, the AXI provides three levels of protection unit support.

See *Protection unit support* on page 5-4.

Atomic operations

The AXI defines mechanisms for both exclusive and locked accesses.

See Chapter 6 *Atomic Accesses*.

Error support

The AXI provides error support for both address decode errors and slave-generated errors.

See Chapter 7 *Response Signaling*.

Unaligned address

To enhance the performance of the initial accesses within a burst, the AXI supports unaligned burst start addresses.

See Chapter 10 *Unaligned Transfers*.

Chapter 2
Signal Descriptions

This chapter defines the AXI signals. Although bus width and transaction ID width are implementation-specific, the tables in this chapter show a 32-bit data bus, a four-bit write data strobe, and four-bit ID fields. This chapter contains the following sections:

2.1 Global signals

Table 2-1 lists the global AXI signals.

Table 2-1 Global signals

Signal	Source	Description
ACLK	Clock source	Global clock signal. All signals are sampled on the rising edge of the global clock.
ARESETn	Reset source	Global reset signal. This signal is active LOW.

2.2 Address channel signals

Table 2-2 lists the AXI address channel signals.

Table 2-2 Address channel signals

Signal	Source	Description
AVALID	Master	Address valid. This signal indicates that valid address and control information are available: 1 = address and control information available 0 = address and control information not available. The address and control information remain stable until the address acknowledge signal, **AREADY**, goes HIGH.
ADDR[31:0]	Master	Address. The address bus gives the address of the first transfer in a burst. The associated control signals are used to determine the addresses of the remaining transfers in the burst.
AWRITE	Master	Burst direction. This signal indicates the transfer direction of a burst: 1 = write transfer 0 = read transfer.
ALEN[3:0]	Master	Burst length. This signal indicates the number of transfers in a burst. See Table 4-1 on page 4-3.
ASIZE[2:0]	Master	Burst size. This signal indicates the size of each transfer in a burst. See Table 4-2 on page 4-4. For write transfers, byte lane strobes indicate which byte lanes to update in memory.
ABURST[1:0]	Master	Burst type. This signal indicates a fixed, incrementing, or wrapping burst. See Table 4-3 on page 4-5. The AXI uses **ABURST[1:0]** and **ASIZE[2:0]** to calculate the address for successive transfers in the burst.
ALOCK[1:0]	Master	Lock type. This signal indicates a normal, exclusive, or locked transaction. See Table 6-1 on page 6-2.
ACACHE[3:0]	Master	Cache type. This signal indicates the bufferable, cacheable, write-through, write-back, and allocate attributes of the transaction. See Table 5-1 on page 5-2.
APROT[2:0]	Master	Protection level. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access. See Table 5-2 on page 5-4.
AID[3:0]	Master	Address ID. This signal is the ID tag of the read or write transaction.
AREADY	Slave	Address ready. This signal indicates that the slave is ready to accept an address: 1 = slave ready 0 = slave not ready.

2.3 Read channel signals

Table 2-3 lists the AXI read channel signals.

Table 2-3 Read channel signals

Signal	Source	Description
RVALID	Slave	Read valid. This signal indicates that valid read data is available: 1 = read data available 0 = read data not available.
RLAST	Slave	Read last. This signal indicates the last transfer in a read burst.
RDATA[31:0]	Slave	Read data. The read data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
RRESP[1:0]	Slave	Read response. This signal indicates the status of the read transfer. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
RID[3:0]	Slave	Read ID tag. This signal is the ID tag of the read transfer. The **RID** value must match the **AID** value of the read transaction to which the slave is responding.
RREADY	Master	Read ready. This signal indicates that the master can accept the read data and response information: 1= master ready 0 = master not ready.

2.4 Write channel signals

Table 2-4 lists the AXI write channel signals.

Table 2-4 Write channel signals

Signal	Source	Description
WVALID	Master	Write valid. This signal indicates that valid write data and strobes are available: 1 = write data and strobes available 0 = write data and strobes not available.
WLAST	Master	Write last. This signal indicates the last transfer in a write burst.
WDATA[31:0]	Master	Write data. The write data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
WSTRB[3:0]	Master	Write strobes. This signal indicates which byte lanes to update in memory. There is one write strobe for each eight bits of the write data bus. Therefore, **WSTRB[n]** corresponds to **WDATA[(8 × n) + 7:(8 × n)]**.
WID[3:0]	Master	Write ID tag. This signal is the ID tag of the write transfer. The **WID** value must match the **AID** value of the write transaction.
WREADY	Slave	Write ready. This signal indicates that the slave can accept the write data: 1 = slave ready 0 = slave not ready.

2.5 Write response channel signals

Table 2-5 lists the AXI write response channel signals.

Table 2-5 Write response channel signals

Signal	Source	Description
BVALID	Slave	Write response valid. This signal indicates that a valid write response is available: 1 = write response available 0 = write response not available.
BRESP[1:0]	Slave	Write response. This signal indicates the status of the write transaction. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
BID[3:0]	Slave	Response ID. The identification tag of the write response. The **BID** value must match the **AID** value of the write transaction to which the slave is responding.
BREADY	Master	Response ready. This signal indicates that the master can accept the response information. 1 = master ready 0 = master not ready.

2.6 Low-power interface signals

Table 2-6 lists the signals of the optional low-power interface.

Table 2-6 Low-power interface signals

Signal	Source	Description
CACTIVE	Peripheral device	Clock active. This signal indicates that the peripheral requires its clock signal: 1 = peripheral clock required 0 = peripheral clock not required.
CSYSREQ	Clock controller	System low-power request. This signal is a request from the system clock controller for the peripheral to enter a low-power state.
CSYSACK	Peripheral device	Low-power request acknowledgement. This signal is the acknowledgement from a peripheral of a system low-power request.

Chapter 3
Channel Handshake

This chapter describes the master/slave handshake process and outlines the relationships and default values of the **READY** and **VALID** handshake signals. It contains the following sections:

3.1 Handshake process

All four channels use the same **VALID/READY** master/slave handshake to transfer data and control information. This two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information moves. The source generates the **VALID** signal to indicate when the data or control information is available. The destination generates the **READY** signal to indicate that it accepts the data or control information. Transfer occurs only when both the **VALID** and **READY** signals are HIGH.

There must be no combinatorial paths between input and output signals on both master and slave interfaces.

Figure 3-1 to Figure 3-3 on page 3-3 show examples of the handshake sequence. In Figure 3-1, the source presents the data or control information and drives the **VALID** signal HIGH. The data or control information from the source remains stable until the destination drives the **READY** signal HIGH, indicating that it accepts the data or control information. The arrow shows when the transfer occurs.

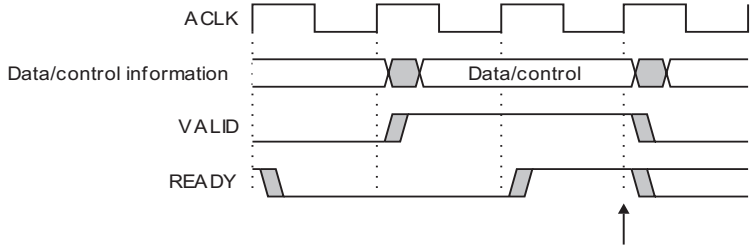


Figure 3-1 VALID before READY handshake

In Figure 3-2 on page 3-3, the destination drives **READY** HIGH before the data or control information is valid. This indicates that the destination can accept the data or control information in a single cycle as soon as it becomes valid. The arrow shows when the transfer occurs.

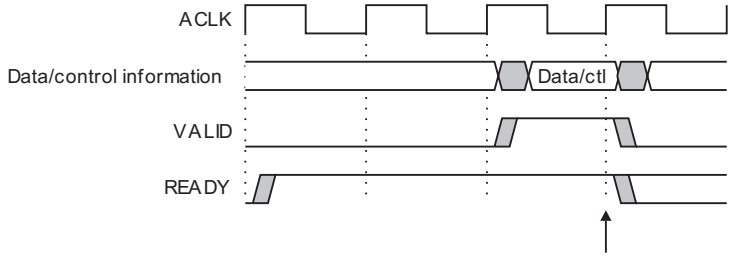


Figure 3-2 READY before VALID handshake

In Figure 3-3, both the source and destination happen to indicate in the same cycle that they can transfer the data or control information. In this case the transfer occurs immediately. The arrow shows when the transfer occurs.

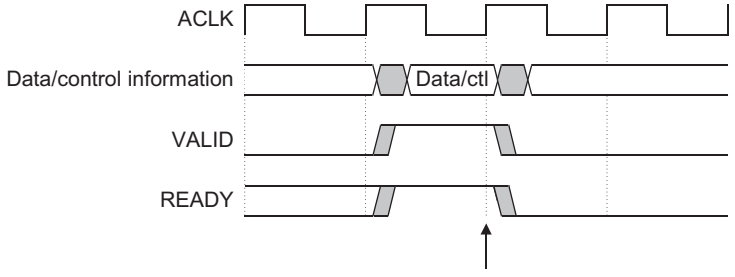


Figure 3-3 VALID with READY handshake

3.1.1 Address channel

The master can assert the **AVALID** signal only when it drives valid address and control information. **AVALID** must remain asserted until the slave accepts the address and control information and asserts the **AREADY** signal.

The default value of **AREADY** can be either HIGH or LOW. The recommended default value is HIGH, although if **AREADY** is HIGH, the slave must be able to accept any valid address that is presented to it.

A default **AREADY** value of LOW is possible but not recommended, because it implies that the transfer takes at least two cycles, one to assert **AVALID** and another to assert **AREADY**.

3.1.2 Read channel

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

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

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

3.1.3 Write channel

During a write burst, the master can assert the **WVALID** signal only when it drives valid write data. **WVALID** must remain asserted until the slave accepts the write data and asserts the **WREADY** signal.

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

The master must assert the **WLAST** signal when it drives the final write transfer in the burst.

When **WVALID** is LOW, the **WSTRB[3:0]** signals can take any value, although it is recommended that they are either driven LOW or held at their previous value.

3.1.4 Write response channel

The slave can assert the **BVALID** signal only when it drives a valid write response. **BVALID** must remain asserted until the master accepts the write response and asserts **BREADY**.

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

3.2 Relationships between the channels

The relationship between the address, read, write, and write response channels is flexible.

For example, the write data can appear at an interface before the address of the write data. This can occur when the address channel contains more register stages than the write channel. It is also possible for the write data to appear in the same cycle as the address.

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

Two relationships that must be maintained are:

- read data must always follow the address to which the data relates
- a write response must always follow the last write transfer in the write transaction to which the write response relates.

3.3 Dependencies between channel handshake signals

To prevent a deadlock situation, you must observe the dependencies that exist between the handshake signals.

In any transaction:

- the **VALID** signal of one AXI component must not be dependent on the **READY** signal of the other component in the transaction
- the **READY** signal can wait for assertion of the **VALID** signal.

A deadlock can occur if a **VALID** signal waits for a **READY** signal. For example, if a master waits for **AREADY** before asserting **WVALID**, and the slave is also waiting for **WVALID** before asserting **AREADY**, a deadlock condition results.

Figure 3-4 and Figure 3-5 on page 3-7 show the handshake signal dependencies. The single-headed arrows point to signals that can be asserted before or after the previous signal is asserted. Double-headed arrows point to signals that must be asserted only after assertion of the previous signal.

Figure 3-4 shows that, in a read transaction:

- the master must not wait for the slave to assert **AREADY** before asserting **AVALID**
- the slave must not assert **RVALID** before the master asserts **AVALID**
- the slave must not assert **RVALID** without first asserting **AREADY**
- the slave must not wait for the master to assert **RREADY** before asserting **RVALID**.

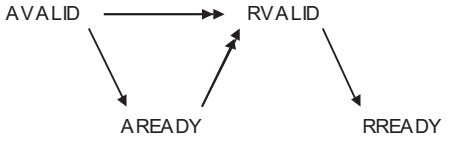


Figure 3-4 Read transaction handshake dependencies

Figure 3-5 on page 3-7 shows that, in a write transaction:

- the master must not wait for the slave to assert **AREADY** or **WREADY** before asserting **AVALID** or **WVALID**
- the slave can wait for **AVALID** or **WVALID** before asserting **AREADY**

- the slave can wait for **AVALID** or **WVALID** before asserting **WREADY**
- the slave must not assert **BVALID** before the master asserts **WVALID**
- the slave must not assert **BVALID** before asserting **WREADY**.

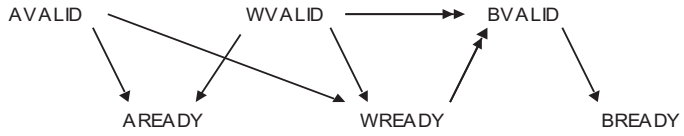


Figure 3-5 Write transaction handshake dependencies

Note

Although an AXI component can wait for **VALID** to be asserted before asserting **READY**, operation can be more efficient when the default value of **READY** is HIGH.

Chapter 4
Addressing Options

This chapter describes AXI burst types and how to calculate addresses and byte lanes for transfers within a burst. It contains the following sections:

4.1 About addressing options

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

To prevent bursts from crossing boundaries between slaves and to limit the size of the address incrementer required within a slave, bursts must not cross 4Kbyte boundaries.

4.2 Burst length

The **ALEN[3:0]** signal specifies the number of data transfers that occur within each burst. As Table 4-1 shows, each burst can be 1-16 transfers long.

Table 4-1 ALEN[3:0] encoding

ALEN[3:0]	Number of data transfers
b0000	1
b0001	2
b0010	3
. . .	
b1100	14
b1110	15
b1111	16

For wrapping bursts, the length of the burst must be 2, 4, 8, or 16 transfers.

Every transaction must have the number of transfers specified by **ALEN[3:0]**. No component can terminate a burst early to reduce the number of data transfers. During a write burst, the master can disable further writing by deasserting all the write strobes, but it must complete the remaining transfers in the burst. During a read burst, the master can discard further read data, but it must complete the remaining transfers in the burst.

Caution

Discarding read data that is not required can result in lost data when accessing a read-sensitive device such as a FIFO. A master must never access such a device using a burst length longer than required.

4.3 Burst size

Table 4-2 shows how the **ASIZE[2:0]** signal specifies the maximum number of data bytes to transfer in each beat, or data transfer, within a burst.

Table 4-2 ASIZE[2:0] encoding

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

The AXI determines from the transfer address which byte lanes of the data bus to use for each transfer.

For incrementing or wrapping bursts with transfer sizes narrower than the data bus, data transfers are on different byte lanes for each beat of the burst. The address of a fixed burst remains constant, and every transfer uses the same byte lanes.

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

4.4 Burst type

The AXI defines three burst types:

- fixed bursts for FIFO-type accesses
- incrementing bursts for normal sequential memory accesses
- wrapping bursts for cache line accesses.

Table 4-3 shows how the **ABURST[1:0]** signal selects the burst type.

Table 4-3 ABURST[1:0] encoding

ABURST[1:0]	Burst type	Description
b00	FIXED	Fixed-address burst
b01	INCR	Incrementing-address burst
b10	WRAP	Incrementing-address burst that wraps to a lower address at the wrap boundary
b11	-	Reserved

4.4.1 Fixed burst

In a fixed burst, the address remains the same for every transfer in the burst. This burst type is for repeated accesses to the same location, such as when loading or emptying a peripheral FIFO.

4.4.2 Incrementing burst

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

4.4.3 Wrapping burst

A wrapping burst is similar to an incrementing burst, in that the address for each transfer in the burst is an increment of the previous transfer address. However, in a wrapping burst the address wraps around to a lower address when a wrap boundary is reached. The wrap boundary is the size of each transfer in the burst multiplied by the total number of transfers in the burst.

Two restrictions apply to wrapping bursts:

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

4.5 Burst address

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

`Start_Address`	The start address issued by the master.
`Number_Bytes`	The maximum number of bytes in each data transfer.
`Data_Bus_Bytes`	The number of byte lanes in the data bus.
`Aligned_Address`	The aligned version of the start address.
`Burst_Length`	The total number of data transfers within a burst.
`Address_N`	The address of transfer N within a burst. N is an integer from 2-16.
`Wrap_Boundary`	The lowest address within a wrapping burst.
`Lower_Byte_Lane`	The byte lane of the lowest addressed byte of a transfer.
`Upper_Byte_Lane`	The byte lane of the highest addressed byte of a transfer.
`INT(x)`	The rounded-down integer value of x.

Use these equations to determine addresses of transfers within a burst:

- `Start_Address` = **ADDR**
- `Number_Bytes` = $2^{\mathbf{ASIZE}}$
- `Burst_Length` = **ALEN** + 1
- `Aligned_Address` = (`INT(Start_Address / Number_Bytes)`) × `Number_Bytes`.

Use this equation to determine the address of the first transfer in a burst:

- `Address_1` = `Start_Address`.

Use this equation to determine the address of any transfer after the first transfer in a burst:

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

For wrapping bursts, the `Wrap_Boundary` variable is extended to account for the wrapping boundary:

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

If `Address_N` = `Wrap_Boundary` + (`Number_Bytes` × `Burst_Length`), use this equation:

- `Address_N` = `Wrap_Boundary`.

Use these equations to determine which byte lanes to use for a the first transfer in a burst:

- Lower_Byte_Lane = Start_Address
- Upper_Byte_Lane = Aligned_Address + Number_Bytes – 1.

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

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

Data is transferred on:

- DATA[(8 × Upper_Byte_Lane) + 7 : (8 × Lower_Byte_Lane)].

Chapter 5
Additional Control Information

This chapter describes AXI support for system-level caches and protection units. It contains the following sections:

5.1 Cache support

The **ACACHE[3:0]** signal supports system-level caches by providing the bufferable, cacheable, and allocate attributes of the transaction:

Bufferable bit, ACACHE[0]

When the bufferable bit is HIGH, the interconnect or any other component can delay the write for an arbitrary number of cycles.

Cacheable bit, ACACHE[1]

When the cacheable bit is HIGH, the transaction at the destination does not have to match the characteristics of the original transaction.

For write transactions, a number of different writes can be merged together.

For read transactions, the read data can be prefetched or fetched only once for multiple read transactions.

To determine whether or not to cache a transaction, use this bit in conjunction with the read allocate bit and the write allocate bit.

Read allocate bit, ACACHE[2]

When the read allocate bit is HIGH, it indicates that a read access that misses can be allocated in the cache. The read allocate bit must not be HIGH if the cacheable bit is LOW.

Write allocate bit, ACACHE[3]

When the write allocate bit is HIGH, it indicates that a write access that misses can be allocated in the cache. The write allocate bit cannot be HIGH if the cacheable bit is LOW.

Table 5-1 shows the encoding of the **ACACHE[3:0]** signal.

Table 5-1 ACACHE[3:0] encoding

ACACHE[3:0]	Transaction attributes
b0000	Noncacheable and nonbufferable
b0001	Bufferable only
b0010	Cacheable, but do not allocate
b0011	Cacheable and bufferable, but do not allocate
b0100 and b0101	Reserved

Table 5-1 ACACHE[3:0] encoding (continued)

ACACHE[3:0]	Transaction attributes
b0110	Cacheable write-through, allocate on reads only
b0111	Cacheable write-back, allocate on reads only
b1000 and b1001	Reserved
b1010	Cacheable write-through, allocate on writes only
b1011	Cacheable write-back, allocate on writes only
b1100 and b1101	Reserved
b1110	Cacheable write-through, allocate on both reads and writes
b1111	Cacheable write-back, allocate on both reads and writes

In write transactions, the **ACACHE[3:0]** signal can determine which device provides the write response. If the **ACACHE[3:0]** signal for a write transaction specifies bufferable, then a buffer component, bridge, or system-level cache can provide the write response. If the **ACACHE[3:0]** signal specifies nonbufferable, then the write response must come from the final destination of the transaction.

The AXI does not determine the mechanism by which buffered or cached data reaches its destination. For example, a system-level cache might have a controller to manage cleaning, flushing, and invalidating cache entries. Another example is the write buffer, which might have control logic to drain the buffer if it receives a nonbufferable write with a matching transaction ID.

5.2 Protection unit support

To support complex system designs, it is often necessary for both the interconnect and other devices in the system to provide protection against illegal transactions. The **APROT[2:0]** signal gives three levels of access protection:

Privileged or normal, APROT[0]

Privileged access typically gives a greater level of system access than normal operation access. Setting **APROT[0]** selects privileged access.

Nonsecure or secure, APROT[1]

Secure access gives a higher level of privileged access. Clearing **APROT[1]** selects secure access.

Instruction or data, APROT[2]

APROT[2] indicates whether a transaction is an instruction access or a data access. Setting **APROT[2]** selects an instruction access.

Note

A transaction might contain a mix of instruction accesses and data accesses. For transactions that might contain both instruction and data accesses, it is recommended that the **APROT[2]** bit is cleared as for a data access.

Table 5-2 summarizes the encoding of the **APROT[2:0]** signal.

Table 5-2 APROT[2:0] encoding

APROT[2:0]	Protection level
[0]	1 = privileged access 0 = normal access
[1]	1 = nonsecure access 0 = secure access
[2]	1 = instruction access 0 = data access

Chapter 6
Atomic Accesses

This chapter describes how the AXI implements exclusive access and locked access mechanisms. It contains the following sections:

6.1 About atomic accesses

To enable the implementation of atomic access primitives, the **ALOCK[1:0]** signal provides exclusive access and locked access. Table 6-1 shows the encoding of the **ALOCK[1:0]** signal.

Table 6-1 ALOCK[1:0] encoding

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

6.2 Exclusive access

The exclusive access mechanism enables the implementation of semaphore type operations without requiring the bus to remain locked to a particular master for the duration of the operation. The advantage of exclusive access is that semaphore type operations do not impact either the critical bus access latency or the maximum achievable bandwidth.

The **ALOCK[1:0]** signal selects exclusive access, and the **RRESP[1:0]** and **BRESP[1:0]** signals (see Table 7-1 on page 7-2) indicate the success or failure of the exclusive access.

The slave must have additional logic to support exclusive access. The AXI provides a fail-safe mechanism to indicate when a master attempts an exclusive access to a slave that does not support it.

6.2.1 Exclusive access process

This is the basic process for an exclusive access:

1. A master performs an exclusive read from an address location.
2. The slave returns the EXOKAY response, indicating that it recorded the address and **AID** value of the master access.
3. At some later time, the master attempts to complete the exclusive access by performing an exclusive write to the same address location.
4. The slave signals the exclusive write of the master as:
   - Successful, if no other master wrote to that location between the read and write accesses. The exclusive write updates the address location, and the slave returns the EXOKAY response.
   - Failed, if another master wrote to that location between the exclusive read and exclusive write. The exclusive write attempt does not update the address location, and the slave returns the OKAY response.

Note

A master might not complete the write portion of an exclusive operation. The exclusive access monitoring hardware is expected to monitor only one address per **AID**. Therefore, if a master does not complete the write portion of an exclusive operation, a subsequent exclusive read changes the address that is being monitored for exclusivity.

6.2.2 Exclusive access from the perspective of the master

A master starts an exclusive operation by performing an exclusive read. This usually returns the EXOKAY response from the slave, indicating that the slave recorded the address to be monitored.

Note

If the master attempts an exclusive read from a slave that does not support exclusive accesses, the slave returns the OKAY response instead of the EXOKAY response. The master can treat this as an error condition indicating that the exclusive access is not supported. It is recommended that the master not perform the write portion of this exclusive operation.

At some time after the exclusive read, the master tries an exclusive write to the same location. If the location has not changed since the exclusive read, the exclusive write operation succeeds. The slave returns the EXOKAY response, and the exclusive write updates the memory location.

If the address location has changed since the exclusive read, the exclusive write attempt fails, and the slave returns the OKAY response instead of the EXOKAY response. The exclusive write attempt does not update the memory location.

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

6.2.3 Exclusive access from the perspective of the slave

A slave that does not support exclusive access must return the OKAY response for both normal and exclusive accesses.

A slave that supports exclusive access must have monitor hardware. It is recommended that such a slave has a monitor unit for each exclusive-capable master ID that can access it. A single-ported slave can have a standard exclusive access monitor external to the slave, but multiported slaves might require internal monitoring.

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

When an exclusive write occurs with a given **AID** value, the monitor checks to see if that address is being monitored for exclusivity. If it is, then this implies that no write has occurred to that location, and the exclusive write proceeds, completing the exclusive access. The slave returns the EXOKAY response to the master.

If the address is no longer being monitored at the time of an exclusive write, this implies one of the following:

- the location has been updated since the exclusive read
- the monitor has been reset to another location.

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

6.2.4 Exclusive access restrictions

The following restrictions apply to exclusive accesses:

- **ALEN[3:0]** must be b0000, selecting a burst length of one transfer.
- The address for the exclusive read and the exclusive write must be identical.
- The **AID** value for the exclusive read and the exclusive write must be identical.
- Except for **AWRITE**, the control signals for the exclusive read and the exclusive write must be identical.
- The value of the **ACACHE[3:0]** signal must guarantee that the slave that is monitoring the exclusive access sees the transaction. For example, an exclusive access being monitored by a slave must not have an **ACACHE[3:0]** value that indicates that the transaction is cacheable.

Failure to observe these restriction causes Unpredictable behavior.

The value of the **ASIZE[2:0]** field defines the minimum number of bytes to be monitored in an exclusive transaction. It is acceptable to monitor up to the maximum transfer size of 128 bytes. However, monitoring a large number of bytes can cause unsuccessful exclusive accesses by updating neighboring bytes.

Using narrow transfers for exclusive accesses helps to ensure that they can pass across an interconnect without width conversion problems. It is recommended that the size of exclusive accesses does not exceed 32 bits.

6.2.5 Slaves that do not support exclusive access

The response signals, **BRESP[1:0]** and **RRESP[1:0]**, include an OKAY response for successful normal accesses and an EXOKAY response for successful exclusive accesses. This means that a slave that does not support exclusive accesses can provide an OKAY response to indicate the failure of an exclusive access.

Note

An exclusive write to a slave that does not support exclusive access always updates the memory location.

An exclusive write to a slave that supports exclusive access updates the memory location only if the exclusive write is successful.

6.3 Locked access

When a locked transaction occurs, the interconnect must ensure that no other master can access that slave region until the master doing the locked transaction does an unlocked transaction. The arbiter in the interconnect is responsible for enforcing this restriction.

To remove a lock, the master must follow the locked transaction with an unlocked transaction.

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

Note

Locked transactions can have an impact on the interconnect performance. It is recommended that locked transactions are used only to support legacy devices.

The following restrictions are recommended but not mandatory:

- keep all locked transaction sequences within the same 4Kbyte address region
- limit locked transaction sequences to two transactions.

Chapter 7
Response Signaling

This chapter describes the four slave responses in AXI read and write transactions. It contains the following sections:

7.1 About response signaling

An AXI slave uses the **RRESP[1:0]** and **BRESP[1:0]** signals to send status information about a transaction back to the master. In a read transaction, the read channel conveys both the read data and the completion response information from the slave. In a write transaction, the write response channel conveys the completion response information from the slave.

The AXI responses are:

- OKAY
- EXOKAY
- SLVERR
- DECERR.

Table 7-1 shows the encoding of the **RRESP[1:0]** and **BRESP[1:0]** signals.

Table 7-1 RRESP[1:0] and BRESP[1:0] encoding

RRESP[1:0] BRESP[1:0]	Response	Meaning
b00	OKAY	Normal access success or exclusive access failure
b01	EXOKAY	Exclusive access success
b10	SLVERR	Slave detects error condition
b11	DECERR	No slave at the transaction address

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

The number of burst transfers specified by **ALEN[3:0]** must be performed, even if an error is reported. For example, if a master requests a read of eight transfers, but the slave detects an error condition, the slave must perform eight transfers, each with an error response.

This protocol places restrictions on masters that can issue multiple outstanding addresses and that must also support precise error signaling. Such masters must be able to handle an error response for an earlier transfer while later transfers are already underway.

7.2 Response types

This section describes the four AXI response types:

- *Normal access success*
- *Exclusive access*
- *Slave error*
- *Decode error*.

7.2.1 Normal access success

The OKAY response indicates:

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

OKAY is the response for most transactions.

7.2.2 Exclusive access

The EXOKAY response indicates the success of an exclusive access. Chapter 6 *Atomic Accesses* describes this response in detail.

7.2.3 Slave error

The SLVERR response indicates an unsuccessful transaction. Examples of slave error conditions are:

- FIFO/buffer overrun or underrun condition
- unsupported transfer size attempted
- write access attempted to read-only location
- timeout condition in the slave
- access attempted to an address where no registers are present
- access attempted to a disabled or powered-down function.

To simplify system monitoring and debugging, it is recommended that error responses are used only for error conditions and not for signaling normal, expected events.

7.2.4 Decode error

In a system without a fully-decoded address map, there can be addresses at which there are no slaves to respond to a transaction. In such a system, the interconnect must provide a suitable error response to flag the access as illegal and also to prevent the system from locking up by trying to access a nonexistent slave.

When the interconnect cannot successfully decode a slave access, it effectively routes the access to a default slave, and the default slave returns the DECERR response.

An implementation option is to have the default slave also record the details of decode errors for later determination of how the errors occurred. In this way, the default slave can significantly simplify the debugging process.

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

Chapter 8
Ordering Model

This chapter describes how transaction ID tags enable the AXI to issue multiple outstanding addresses and process transactions out of order. It contains the following sections:

8.1 About the ordering model

The AXI enables out-of-order transaction completion and the issuing of multiple outstanding addresses. These features enable the implementation of a high-performance interconnect, maximizing data throughput and system efficiency.

The ID signals, **AID**, **WID**, **RID**, and **BID**, support out-of-order transactions by enabling each port to act as multiple ordered ports. All transactions with a given ID must be ordered, but there is no restriction on the ordering of transactions with different IDs. The four transaction IDs are:

AID The ID tag for a transaction address. The master transfers the **AID** with the transaction address.

WID The ID tag for a write transaction. Along with the write data, the master transfers a **WID** to match the **AID** of the corresponding address.

RID The ID tag for a read transaction. The slave transfers an **RID** to match the **AID** of the transaction to which it is responding.

BID The ID tag for the write response. The slave transfers a **BID** to match the **AID** and **WID** of the transaction to which it is responding.

Note

There is no requirement for slaves and masters to use these advanced features. Simple masters and slaves can process one transaction at a time in the order they are issued.

The ability to issue multiple outstanding addresses means that masters can issue transaction addresses without waiting for earlier transactions to complete. Because it enables parallel processing of transactions, this feature can improve system performance.

The ability to complete transactions out of order means that transactions to faster memory regions can complete without waiting for earlier transactions to slower memory regions. Because it reduces the effect of transaction latency, this feature can also improve system performance.

Note

The reordering of transactions is always with respect to other transactions. There is no facility for the reordering of data transfers within a burst. The address and control signals that define the burst control the order of transfers within the burst.

8.2 Transaction ID fields

The AXI protocol provides an address ID field, **AID**, to enable a master to issue a number of separate transactions, each of which must be returned in order.

A master can use the **AID** field of a transaction to provide additional information about the ordering requirements of the master. The rules governing the ordering of transactions are as follows:

- Transactions from different masters have no ordering restrictions. They can complete in any order.
- Transactions from the same master, but with different **AID** values, have no ordering restrictions. They can complete in any order.
- The data for a sequence of write transactions with the same **AID** value must complete in the same order in which the master issued the addresses.
- The data for a sequence of read transactions with the same **AID** value must be returned as follows:
  - When the reads are from the same slave, the slave must ensure that the read data returns in the same order in which the master issued the addresses.
  - When the reads are from different slaves, the interconnect must ensure that the read data returns in the same order in which the master issued the addresses.
- Read and write transactions with the same **AID** value can complete in any order, but the result must be the same as if they were executed in the same order in which they were issued.

8.3 Address ordering

The interconnect must preserve the order in which addresses with the same **AID** value are issued from a master so that they arrive in the same order at the slave. This ensures that slaves can process according to the transaction ordering rules in *Transaction ID fields* on page 8-3.

An interconnect almost always presents addresses to a slave in the same order in which the master issued them. However, when an interconnect provides multiple address paths from a single master to a single slave, special attention is required to ensure that address ordering is maintained.

8.4 Read ordering

At a master interface, read data from read transactions with the same **AID** value must arrive in the same order in which the master issued the addresses. Data from read transactions with different **AID** values can return in any order.

A slave must return read data from a sequence of read transactions with the same **AID** value in the same order in which it received the addresses. In a sequence of read transactions with different **AID** values, the slave can return the read data in a different order than that in which the transactions arrived.

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

The interconnect must ensure that a sequence of read transactions with the same **AID** value from different slaves complete in order.

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

8.5 Normal write ordering

If a slave does not support write data interleaving (see *Write data interleaving* on page 8-7), the master must issue the data of write transactions in the same order in which it issues the transaction addresses.

A slave usually receives write data in the same order that it received the addresses. If the interconnect combines write transactions from different masters to one slave, it must ensure that it combines the write data in address order.

These restrictions apply even if the write transactions have different **AID** values.

8.6 Write data interleaving

Write data interleaving enables a slave interface to accept interleaved write data with different **AID** values. The slave declares a write data interleaving depth that indicates if the interface can accept interleaved write data from sources with different **AID** values. The write data interleaving depth is statically configured. By default, the write data interleaving depth of any interface is one.

The write data interleaving depth is the number of different addresses that are currently pending in the slave interface for which write data can be supplied. For example, a slave with a write data interleaving depth of two that has four different addresses pending can accept data for either of the first two pending addresses.

AXI cannot interleave write data of transactions that have the same **AID** value.

The order in which a slave receives the first data item of each transaction must be the same as the order in which it receives the addresses for the transactions.

Write data interleaving can prevent stalling when the interconnect combines multiple streams of write data destined for the same slave. The interconnect might combine one write data stream from a slow source and another write data stream from a fast source. By interleaving the two write data streams, the interconnect can improve system performance.

Note

If two write transactions with different **AID** values access the same or overlapping address locations, the processing order is not defined. A higher-level protocol must ensure the correct order of transaction processing.

A master interface that is capable of generating write data with only one **WID** value generates all write data in the same order in which it issues the addresses. However, a master interface can interleave write data with different **WID** values if the slave interface has a write data interleaving depth greater than one.

For most masters that can internally control the generation of the write data, write data interleaving is not necessary. Such a master can generate the write data in the same order in which it generates the addresses. However, a master interface that is passing write data from multiple sources with different speeds can interleave the sources to make maximum use of the interconnect.

To avoid a deadlock situation, a slave interface must have a write reorder depth greater than one only if it can continuously accept interleaved write data. The slave interface must never stall the acceptance of write data in an attempt to change the order of the write data.

8.7 Read and write interaction

Read and write transactions can complete in any order. The only restriction is that, at the slave, the results of the transactions must be the same as if they were completed in the same order as they were issued.

For memory regions this means that read and write transactions can complete in any order unless the transactions are to the same, or overlapping, address regions. If transactions are to the same address range, the slave might force the order of completion if it must complete an earlier transaction to correctly complete a later one.

For peripheral devices, the effect of the transactions must be the same as if the transactions were completed in order. Peripheral devices are not expected to attempt to reorder transactions.

The interconnect does not force any relationship between read and write transactions. If a master requires a particular ordering between two transactions, then it must issue the address of the second transaction after the first transaction is complete. This usually occurs when the two transactions are with two different slaves.

8.8 Examples of transaction reordering

This section gives a examples of transaction reordering.

8.8.1 Read-write-read reorder options

The first example is a master that issues a read-write-read sequence, and all the transactions have the same **AID** value. The data phase of the transactions can complete in several different ways. Possible examples are:

- they can complete in the same order in which the transactions were issued
- the write can complete before the two reads complete.

Note

The first read cannot complete after the second read. This would violate the rules about reordering read transactions with the same **AID** value.

With the transactions issued in the order R1-W-R2, Table 8-1 shows the reordering options.

Table 8-1 Read-write-read reorder options

Order	Allowable	Description
R1-W-R2	Yes	Original ordering
R1-R2-W	-	Not possible if the slave requires the write data to complete R2
W-R1-R2	Yes	Slave must provide R1 with the value it held before the write transaction
W-R2-R1	No	Violates read ordering rules
R2-R1-W	No	Violates read ordering rules
R2-W-R1	No	Violates read ordering rules

The second entry in Table 8-1 shows the case in which the order of completion might be forced if the addresses of the transactions are the same or overlap. In the example, if the address of the write overlaps with the address of the read, the slave might be forced to complete the write to obtain the correct data for the read.

8.8.2 Write-read-write reorder options

Table 8-2 on page 8-10 shows the options for a write-read-write sequence, W1-R-W2, with all transactions having the same **AID** value.

Table 8-2 Write-read-write reorder options

Order	Allowable	Description
W1-R-W2	Yes	Original ordering
W1-W2-R	Yes	Slave must provide R with the value it held before the W2 write transaction
R-W1-W2	-	Not possible if slave requires the W1 data to complete R
R-W2-W1	No	Violates write ordering rules
W2-R-W1	No	Violates write ordering rules
W2-W1-R	No	Violates write ordering rules

8.9 Interconnect use of ID fields

When a master interface is connected to an interconnect, the interconnect appends additional bits to both the **AID** and **WID** fields that are unique to that master port. This has two effects:

- masters do not have to know what ID values are used by other masters, because the interconnect makes the ID values unique when it appends the master number to the field
- the width of the ID field at a slave interface is wider than the ID field at a master interface.

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

8.10 Recommended width of ID fields

To take advantage of the AXI out-of-order transaction capability, use the following recommendations:

- implement a transaction ID up to four bits in master components
- implement up to four additional bits of transaction ID for master port numbers in the interconnect
- implement eight bits of ID support in slave components.

Chapter 9
Data Buses

This chapter describes transfers of varying sizes on the AXI read and write data buses and how the AXI uses byte-invariant endianness to handle mixed-endian transfers. It contains the following sections:

9.1 About the data buses

The AXI has two independent data buses, one for read data and one for write data. Because these data buses have their own individual handshake signals, it is possible for data transfers to occur on both buses at the same time.

Every transfer generated by a master must be the same width as or narrower than the data bus for the transfer.

9.2 Write strobes

The write strobe signals, **WSTRB**, enable sparse data transfer on the write data bus. Each write strobe signal corresponds to one byte of the write data bus. When asserted, a write strobe indicates that the corresponding byte lane of the data bus contains valid information to be updated in memory.

There is one write strobe for each eight bits of the write data bus, so **WSTRB[n]** corresponds to **WDATA[(8 × n) + 7: (8 × n)]**. Figure 9-1 shows this relationship on a 64-bit data bus.

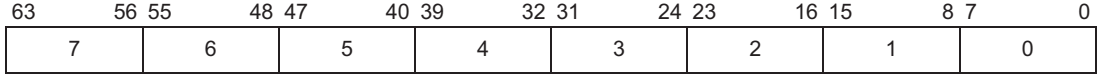


Figure 9-1 Byte lane mapping

A master must ensure that the write strobes are asserted only for byte lanes that can contain valid data as determined by the control information for the transaction.

9.3 Narrow transfers

When a master generates a transfer that is narrower than its data bus, the address and control information determine which byte lanes the transfer uses. In incrementing or wrapping bursts, different byte lanes transfer the data on each beat of the burst. In a fixed burst, the address remains constant, and the byte lanes that can be used also remain constant.

Figure 9-2 and Figure 9-3 give two examples of byte lanes use.

In Figure 9-2:

- the burst has five transfers
- the starting address is 0
- each transfer is eight bits
- the transfers are on a 32-bit bus.

Byte lane used				
			DATA[7:0]	1st transfer
		DATA[15:8]		2nd transfer
	DATA[23:16]			3rd transfer
DATA[31:24]				4th transfer
			DATA[7:0]	5th transfer

Figure 9-2 Narrow transfer example with eight-bit transfers

In Figure 9-3:

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

Byte lane used		
DATA[63:32]		1st transfer
	DATA[31:0]	2nd transfer
DATA[63:32]		3rd transfer

Figure 9-3 Narrow transfer example with 32-bit transfers

9.4 Byte invariance

To access mixed-endian data structures that reside in the same memory space, the AXI uses a byte-invariant endian scheme.

Byte-invariant endianness means that a byte transfer to a given address passes the eight bits of data on the same data bus wires to the same address location.

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

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

Figure 9-4 is an example of a data structure requiring byte-invariant access. It is possible that the header information, such as the source and destination identifiers, is in little-endian format, but the payload is a big-endian byte stream.

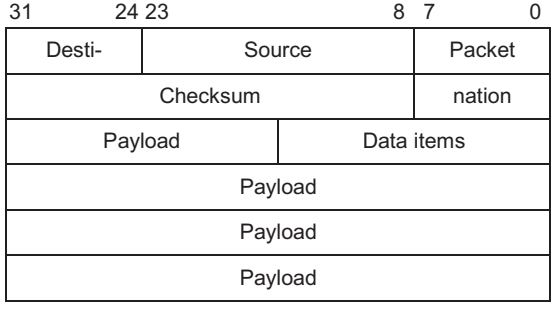


Figure 9-4 Example mixed-endian data structure

Byte invariance ensures that little-endian access to parts of the header information does not corrupt other big-endian data within the structure.

Chapter 10
Unaligned Transfers

This chapter describes how the AXI handles unaligned transfers. It contains the following sections:

10.1 About unaligned transfers

The AXI protocol uses burst-based addressing, which means that each transaction consists of a number of data transfers. Typically, each data transfer is aligned to the size of the transfer. For example, a 32-bit wide transfer is usually aligned to four-byte boundaries. However, there are times when it is desirable to begin a burst at an unaligned address.

For any burst that is made up of data transfers wider than one byte, it is possible that the first bytes that have to be accessed do not align with the natural data width boundary. For example, a 32-bit (four-byte) data packet that starts at a byte address of `0x1002` is not aligned to a 32-bit boundary.

The AXI protocol enables a master to use the low-order address lines to signal an unaligned start address for a burst. The information on the low-order address lines must be consistent with the information contained on the byte lane strobes.

Note

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

The master can also simply provide an aligned address and, in a write transaction, rely on the byte lane strobes to provide the information about which byte lanes the data is using.

10.2 Examples

Figure 10-1, Figure 10-2 on page 10-4, and Figure 10-3 on page 10-4 show examples of aligned and unaligned transfers on buses with different widths. Each row in the figures represents a transfer. The shaded cells indicate bytes that are not transferred, based on the address and control information.

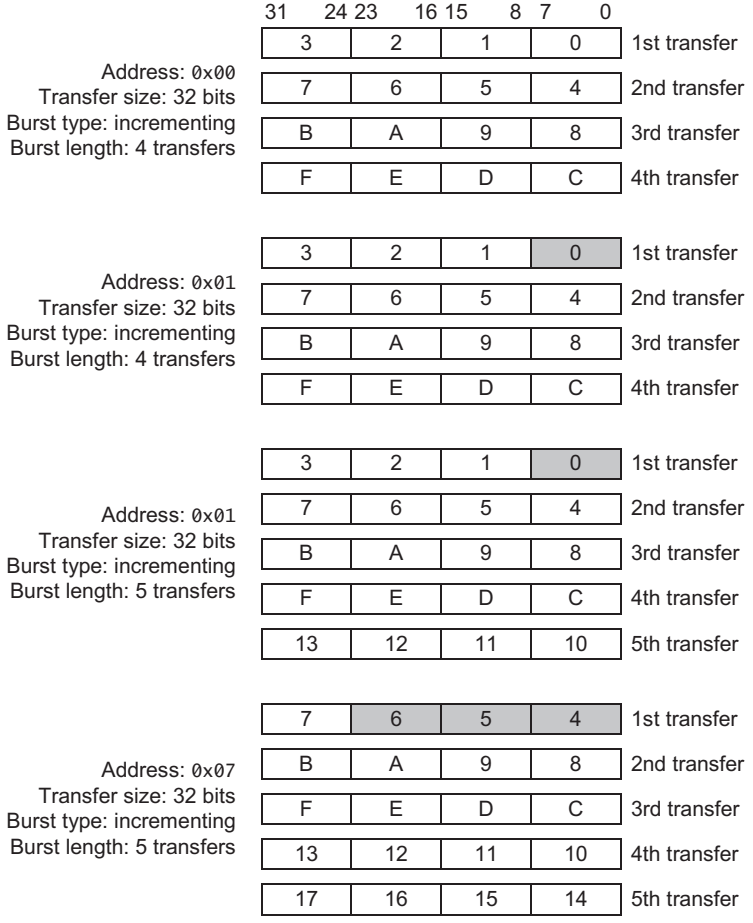


Figure 10-1 Aligned and unaligned word transfers on a 32-bit bus

Figure 10-2 on page 10-4 shows three bursts of 32-bit transfers on a 64-bit bus.

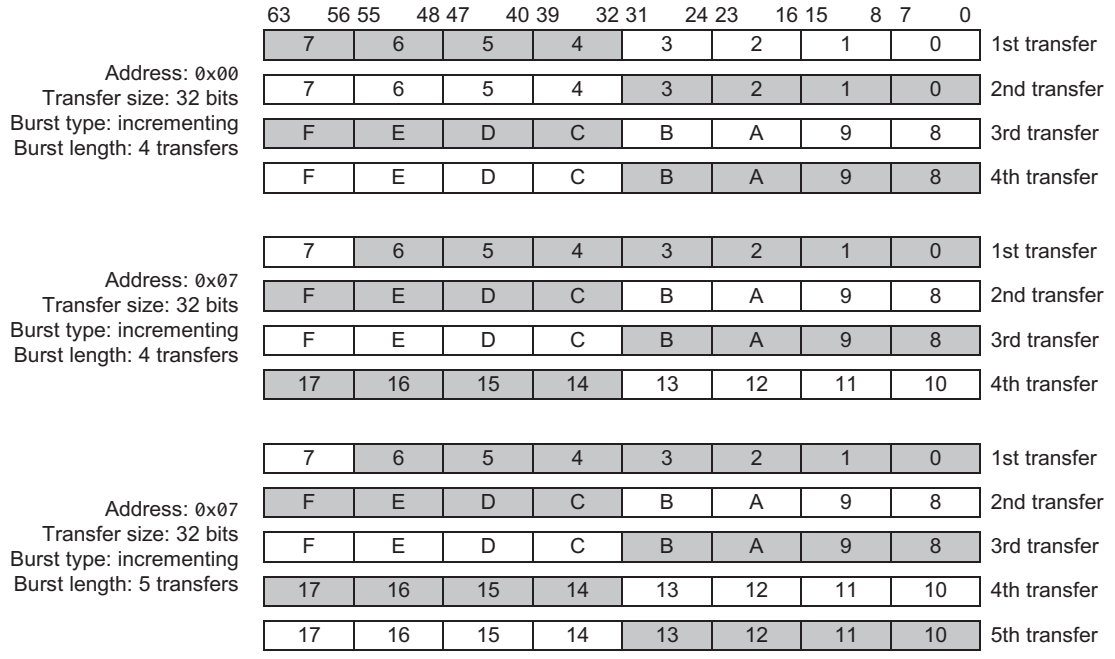


Figure 10-2 Aligned and unaligned word transfers on a 64-bit bus

Figure 10-3 shows a wrapping burst of 32-bit transfers on a 64-bit bus.

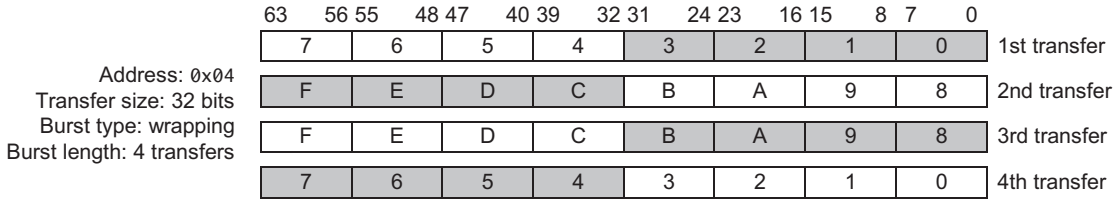


Figure 10-3 Aligned wrapping word transfers on a 64-bit bus

Chapter 11
Clock and Reset

This chapter describes the timing of the AXI clock and reset signals. It contains the following section:

- *Clock and reset requirements* on page 11-2.

11.1 Clock and reset requirements

This section gives the requirements for implementing the **ACLK** and **ARESETn** signals.

11.1.1 Clock

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

There must be no combinatorial paths between input and output signals on both master and slave interfaces.

11.1.2 Reset

The AXI protocol includes a single active LOW reset signal, **ARESETn**. The reset signal can be asserted asynchronously, but deassertion must be synchronous after the rising edge of **ACLK**.

During reset the following interface requirements apply:

- a master interface must drive **AVALID** and **WVALID** LOW
- a slave interface must drive **RVALID** and **BVALID** LOW.

All other signals can be driven to any value.

A master interface must begin driving **AVALID** or **WVALID** HIGH only at a rising **ACLK** edge after **ARESETn** is HIGH. Figure 11-1 shows the first point after reset that **AVALID** or **WVALID**, can be driven HIGH.

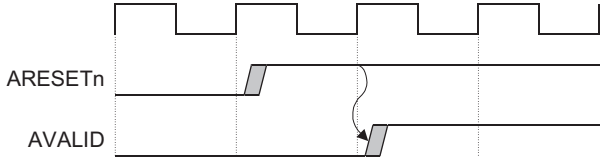


Figure 11-1 Exit from reset

Chapter 12
Low-power Interface

This chapter describes the AXI clock control interface during entry into and exit from a low-power state. It contains the following sections:

12.1 About the low-power interface

The low-power interface is an optional extension to the data transfer protocol that targets two different classes of peripherals:

- Peripherals that require a power-down sequence, and that can have their clocks turned off only after they enter a low-power state. These peripherals require an indication from a system clock controller to determine when to initiate the power-down sequence.
- Peripherals that have no power-down sequence, and that can independently indicate when it is acceptable to turn off their clocks.

12.2 Low-power clock control

The low-power clock control interface consists of the following signals:

- a signal from the peripheral indicating when its clocks can be enabled or disabled
- two handshake signals for the system clock controller to request exit or entry into a low-power state.

The primary signal in the clock control interface is **CACTIVE**. The peripheral uses this signal to indicate when it requires its clock to be enabled. The peripheral asserts **CACTIVE** to indicate that it requires the clock, and the system clock controller must enable the clock immediately. The peripheral deasserts **CACTIVE** to indicate that it does not require the clock. The system clock controller can then determine whether to enable or disable the peripheral clock.

A peripheral that can have its clock enabled or disabled at any time can drive **CACTIVE** LOW permanently. A peripheral that must have its clock always enabled must drive **CACTIVE** HIGH permanently.

This simple interface to the system clock controller is sufficient for some peripherals with no power-down or power-up sequence.

For a more complex peripheral with a power-down or power-up sequence, entry into a low-power state occurs only after a request from the system clock controller. The AXI provides a two-wire request/acknowledge handshake to support this request:

CSYSREQ To request that the peripheral enter a low-power state, the system clock controller drives the **CSYSREQ** signal LOW. During normal operation, **CSYSREQ** is HIGH.

CSYSACK The peripheral uses the **CSYSACK** signal to acknowledge both the low-power state request and the exit from the low-power state.

Figure 12-1 shows the relationship between **CSYSREQ** and **CSYSACK**.

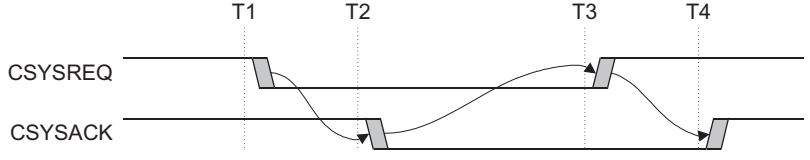


Figure 12-1 CSYSREQ and CSYSACK handshake

At the start of the sequence in Figure 12-1 on page 12-3, both **CSYSREQ** and **CSYSACK** are HIGH for normal clocked operation. At time T1, the system clock controller deasserts **CSYSREQ**, indicating a request to put the peripheral in a low-power state. The peripheral acknowledges the request at time T2 by deasserting **CSYSACK**. At T3, the system clock controller asserts **CSYSREQ** to indicate the exit from the low-power state, and the peripheral asserts **CSYSACK** at T4 to acknowledge the exit.

This relationship between **CSYSREQ** and **CSYSACK** is a requirement of the AXI protocol.

The peripheral can accept or deny the request for a low-power state from the system clock controller. The level of the **CACTIVE** signal when the peripheral acknowledges the request by deasserting **CSYSACK** indicates the acceptance or denial of the request.

12.2.1 Acceptance of low-power request

Figure 12-2 shows the sequence of events when a peripheral accepts a system low-power request.

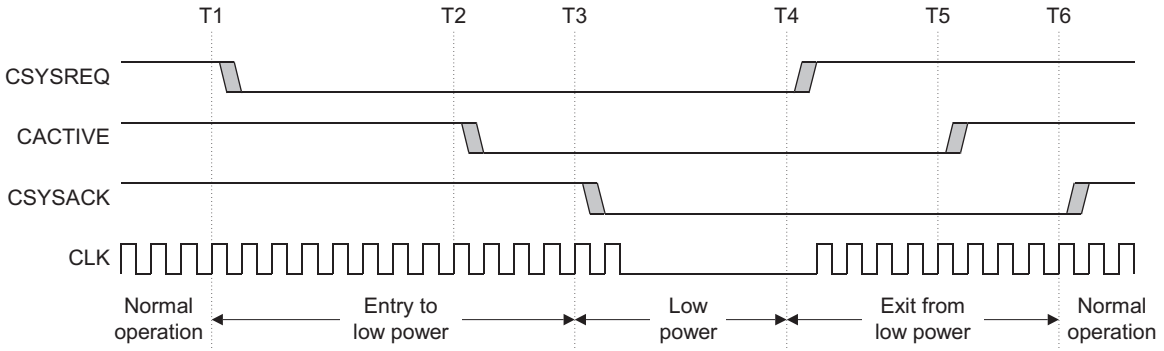


Figure 12-2 Acceptance of a low-power request

In Figure 12-2, the sequence begins at T1 when the system clock controller deasserts **CSYSREQ** to request that the peripheral enter a low power state. After the peripheral recognizes the request, it can then perform its power-down function and deassert **CACTIVE**. The peripheral then deasserts **CSYSACK** at T3 to complete the entry into the low-power state.

At T4, the system clock controller begins the low-power state exit sequence by asserting **CSYSREQ**. The peripheral then asserts **CACTIVE** at T5 and completes the exit sequence at T6 by asserting **CSYSACK**.

12.2.2 Denial of a low-power request

Figure 12-3 shows the sequence of events when a peripheral denies a system low-power request.

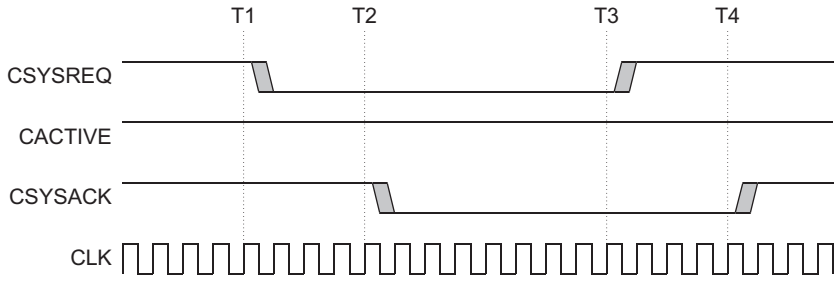


Figure 12-3 Denial of a low-power request

In Figure 12-3, the peripheral denies a low-power request by holding **CACTIVE** HIGH when it acknowledges the low-power request. After that point, the system clock controller must complete the low-power request handshake by asserting **CSYSREQ** before it can initiate another request.

12.2.3 Exiting a low-power state

Either the system clock controller or the peripheral can request to exit the low-power state and restore the clock. By definition, both **CACTIVE** and **CSYSREQ** are LOW during the low power state, and driving either of these signals HIGH initiates the exit sequence.

The system clock controller can initiate the exit from the low-power state by enabling the clock and driving **CSYSREQ** HIGH. The peripheral can then perform a power-up sequence in which it drives **CACTIVE** HIGH. Then it completes the exit by driving **CSYSACK** HIGH.

The peripheral can initiate the exit from a a low-power state by driving **CACTIVE** HIGH. The system clock controller must then immediately restore the clock. It must also drive **CSYSREQ** HIGH to continue the handshake sequence. The peripheral then completes the sequence by driving **CSYSACK** HIGH while exiting the low-power state. The peripheral can keep **CSYSACK** LOW for as many cycles as it requires to complete the exit sequence.

12.2.4 Clock control sequence summary

Figure 12-4 on page 12-6 shows the typical flow for entering and exiting a low-power state.

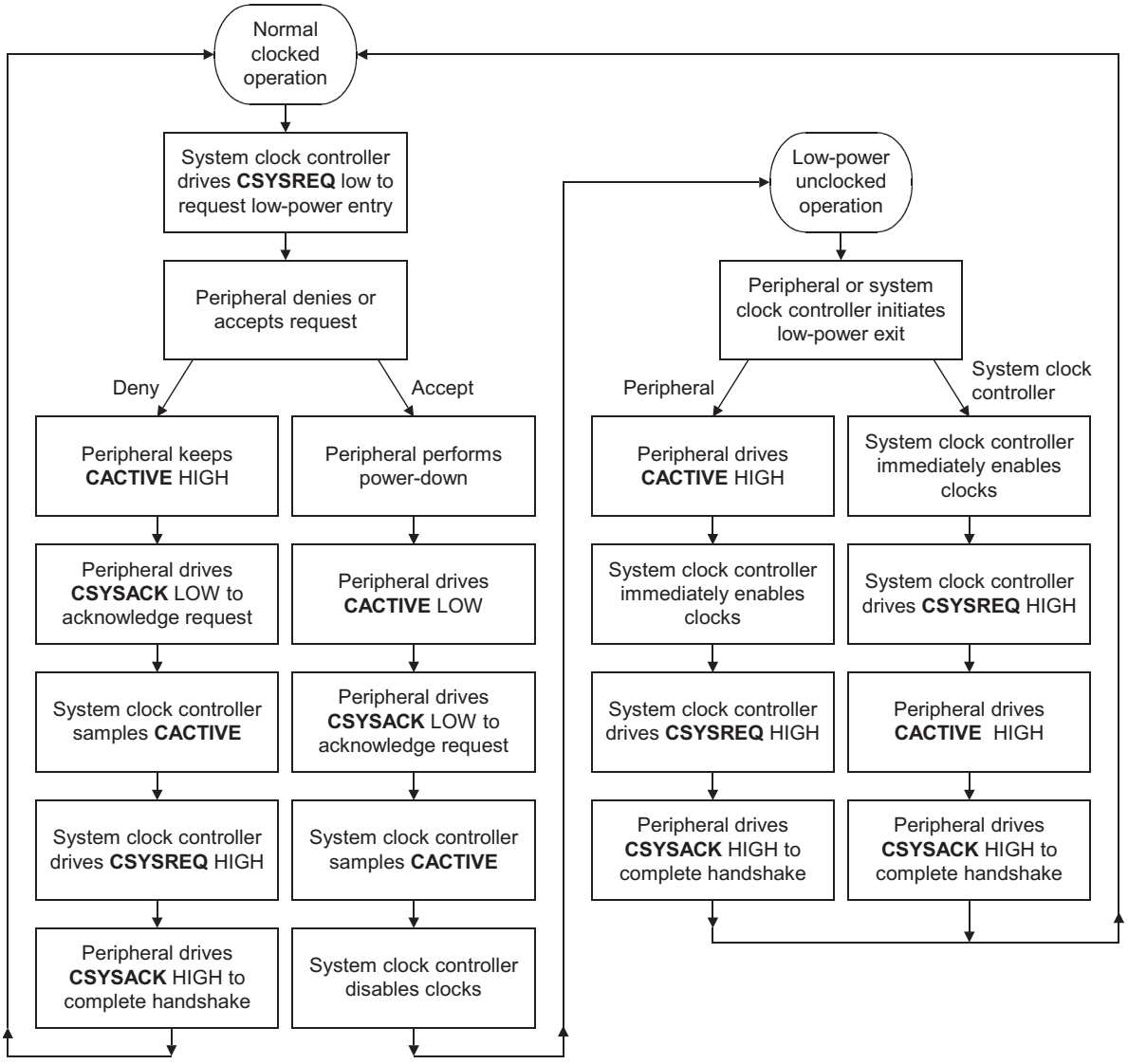


Figure 12-4 Low-power clock control sequence

12.2.5 Combining peripherals in a low-power domain

The system clock controller can combine a number of different peripherals within the same low-power clock domain. Then the clock domain can be treated in the same way as a single peripheral if the following rules are observed:

- The clock domain **CACTIVE** signal is the logical OR of all the **CACTIVE** signals within the clock domain. This means that the system clock controller can disable the clocks only when all peripherals indicate that they can be disabled.
- The system clock controller can use a single **CSYSREQ** signal that is routed to all peripherals within the clock domain.
- The clock domain **CSYSACK** signal is generated as follows:
  - the falling edge of **CSYSACK** occurs when the last falling edge from all of the peripherals occurs
  - the rising edge of **CSYSACK** occurs when the last rising edge from all of the peripherals occurs.

Index

The items in this index are listed in alphabetical order with references to page numbers.

C

D

E

F

H

I

L

M

N

O

P

