# Advanced eXtensible Interface

The **Advanced eXtensible Interface** (**AXI**), is an on-chip communication bus protocol developed by ARM. It is part of the Advanced Microcontroller Bus Architecture 3 (AXI3) and 4 (AXI4) specifications.[1]

AXI has been introduced in 2003 with the AMBA3 specification. In 2010, a new revision of AMBA, AMBA4, defined the AXI4, AXI4-Lite and AXI4-Stream protocol. AXI is royalty-free and its specification is freely available from ARM.

AMBA AXI specifies many optional signals, which can be included depending on the specific requirements of the design,[2] making AXI a versatile bus for numerous applications.

While the communication over an AXI bus is between a single initiator and a single target, the specification includes detailed description and signals to include N:M interconnects, able to extend the bus to topologies with more initiators and targets.[3]

AMBA AXI4, AXI4-Lite and AXI4-Stream have been adopted by Xilinx and many of its partners as main communication buses in their products.[4][5]

## Contents



## Thread IDs

Thread IDs allow a single initiator port to support multiple threads, where each thread has in-order access to the AXI address space, however each thread ID initiated from a single initiator port may complete out of order with respect to each other. For instance in the case where one thread ID is blocked by a slow peripheral, another thread ID may continue independently of the order of the first thread ID. Another example, one thread on a cpu may be assigned a thread ID for a particular initiator port memory access such as read addr1, write addr1, read

addr1, and this sequence will complete in order because each transaction has the same initiator port thread ID. Another thread running on the cpu may have another initiator port thread ID assigned to it, and its memory access will be in order as well but may be intermixed with the first thread IDs transactions.

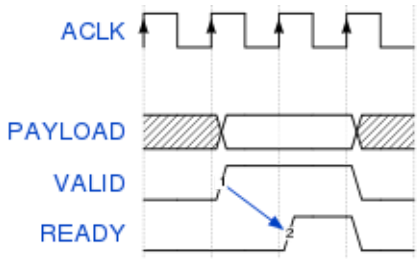
Thread IDs on an initiator port are not globally defined, thus an AXI switch with multiple initiator ports will internally prefix the initiator port index to the thread ID, and provide this concatenated thread ID to the target device, then on return of the transaction to its initiator port of origin, this thread ID prefix will be used to locate the initiator port and the prefix will be truncated. This is why the target port thread ID is wider in bits than the initiator port thread ID.

Axi-lite bus is an AXI bus that only supports a single ID thread per initiator. This bus is typically used for an end point that only needs to communicate with a single initiator device at a time, example, a simple peripheral such as a UART. In contrast, a CPU is capable of initiating transactions to multiple peripherals and address spaces at a time, and will support more than one thread ID on its AXI initiator ports and AXI target ports. This is why a CPU will typically support a full spec AXI bus. A typical example of a front side AXI switch would include a full specification AXI initiator connected to a CPU initiator, and several AXI-lite targets connected to the AXI switch from different peripheral devices.

(Additionally, the AXI-lite bus is restricted to only support transaction lengths of a single data word per transaction.)

## Handshake

AXI defines a basic handshake mechanism, composed by an xVALID and xREADY signal.[6] The xVALID signal is driven by the source to inform the destination entity that the payload on the channel is valid and can be read from that clock cycle onwards. Similarly, the xREADY signal is driven by the receiving entity to notify that it is prepared to receive data.



Basic handshake mechanism of the AMBA AXI protocol. In this example, the destination entity waits for a high VALID to assert its own READY.

When both the xVALID and xREADY signals are high in the same clock cycle, the data payload is considered "transferred" and the source can either provide a new data payload, by keeping high xVALID, or terminate the transmission, by de-asserting xVALID. An individual data transfer, so a clock cycle when both xVALID and xREADY are high, is called "beat".

Two main rules are defined for the control of these signals:

- A source must not wait for a high xREADY to assert xVALID.
- Once asserted, a source must keep a high xVALID until a handshake occurs.

Thanks to this handshake mechanism, both the source and the destination can control the flow of data, throttling the speed if needed.

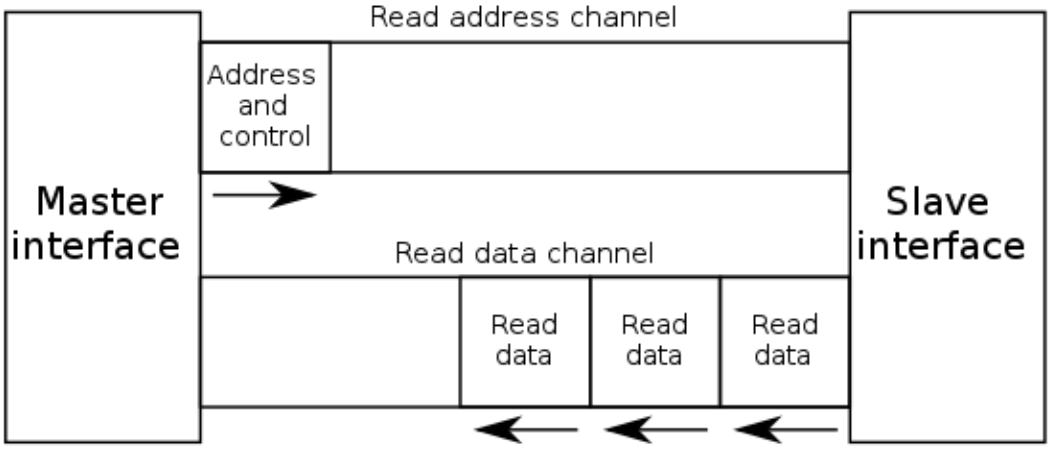
## Channels

In the AXI specification, five channels are described:[7]

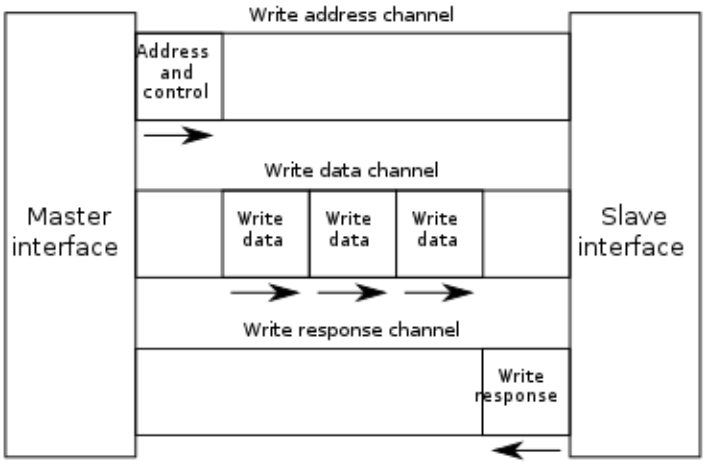
- Read Address channel (AR)
- Read Data channel (R)
- Write Address channel (AW)
- Write Data channel (W)

- Write Response channel (B)

Other than some basic ordering rules,[8] each channel is independent from each other and has its own couple of xVALID/xREADY handshake signals.[9]




AXI Read Address and Read Data channels.




AXI Write Address, Write Data and Write Response channels.

# AXI

## Signals

Signals of the Read and Write Address channels

| Signal description | Write Address channel | Read Address channel |
|---|---|---|
| Address ID, to identify multiple streams over a single channel | AWID | ARID |
| Address of the first beat of the burst | AWADDR | ARADDR |
| Number of beats inside the burst | AWLEN[nb 1] | ARLEN[nb 1] |
| Size of each beat | AWSIZE | ARSIZE |
| Type of the burst | AWBURST | ARBURST |
| Lock type, to provide atomic operations | AWLOCK[nb 1] | ARLOCK[nb 1] |
| Memory type, how the transaction has to progress through the system | AWCACHE | ARCACHE |
| Protection type: privilege, security level and data/instruction access | AWPROT | ARPROT |
| Quality of service of the transaction | AWQOS[nb 2] | ARQOS[nb 2] |
| Region identifier, to access multiple logical interfaces from a single physical one | AWREGION[nb 2] | ARREGION[nb 2] |
| User-defined data | AWUSER[nb 2] | ARUSER[nb 2] |
| xVALID handshake signal | AWVALID | ARVALID |
| xREADY handshake signal | AWREADY | ARREADY |

Signals of the Read and Write Data channels

| Signal description | Write Data channel | Read Data channel |
|---|---|---|
| Data ID, to identify multiple streams over a single channel | WID[nb 3] | RID |
| Read/Write data | WDATA | RDATA |
| Read response, to specify the status of the current RDATA signal | | RRESP |
| Byte strobe, to indicate which bytes of the WDATA signal are valid | WSTRB | |
| Last beat identifier | WLAST | RLAST |
| User-defined data | WUSER[nb 2] | RUSER[nb 2] |
| xVALID handshake signal | WVALID | RVALID |
| xREADY handshake signal | WREADY | RREADY |

Signals of the Write Response channel

| Signal description | Write Response channel |
|---|---|
| Write response ID, to identify multiple streams over a single channel | BID |
| Write response, to specify the status of the burst | BRESP |
| User-defined data | BUSER[nb 2] |
| xVALID handshake signal | BVALID |
| xREADY handshake signal | BREADY |

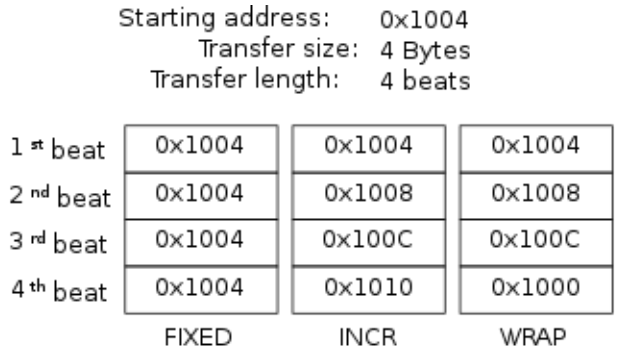
[10]

1. Different behavior between AXI3 and AXI4
2. Available only with AXI4
3. Available only with AXI3

## Bursts

AXI is a burst-based protocol,[11] meaning that there may be multiple data transfers (or beats) for a single request. This makes it useful in the cases where it is necessary to transfer large amount of data from or to a specific pattern of addresses. In AXI, bursts can be of three types, selected by the signals ARBURST (for reads) or AWBURST (for writes):[12]

- FIXED
- INCR
- WRAP



Example of FIXED, INCR and WRAP bursts

In FIXED bursts, each beat within the transfer has the same address. This is useful for repeated access at the same memory location, such as when reading or writing a FIFO.

$$Address = StartAddress$$

In INCR bursts, on the other hand, each beat has an address equal to the previous one plus the transfer size. This burst type is commonly used to read or write sequential memory areas.

$$Address_i = StartAddress + i \cdot TransferSize$$

WRAP bursts are similar to the INCR ones, as each transfer has an address equal to the previous one plus the transfer size. However, with WRAP bursts, if the address of the current beat reaches the "Higher Address boundary", it is reset to the "Wrap boundary":

$$Address_i = WrapBoundary + (StartAddress + i \cdot TransferSize) \bmod (BurstLength \cdot TransferSize)$$

with

$$WrapBoundary = \left\lfloor \frac{StartAddress}{NumberBytes \cdot BurstLength} \right\rfloor \cdot (NumberBytes \cdot BurstLength)$$

## Transactions

### Reads

To start a read transaction, the initiator has to provide on the Read address channel:

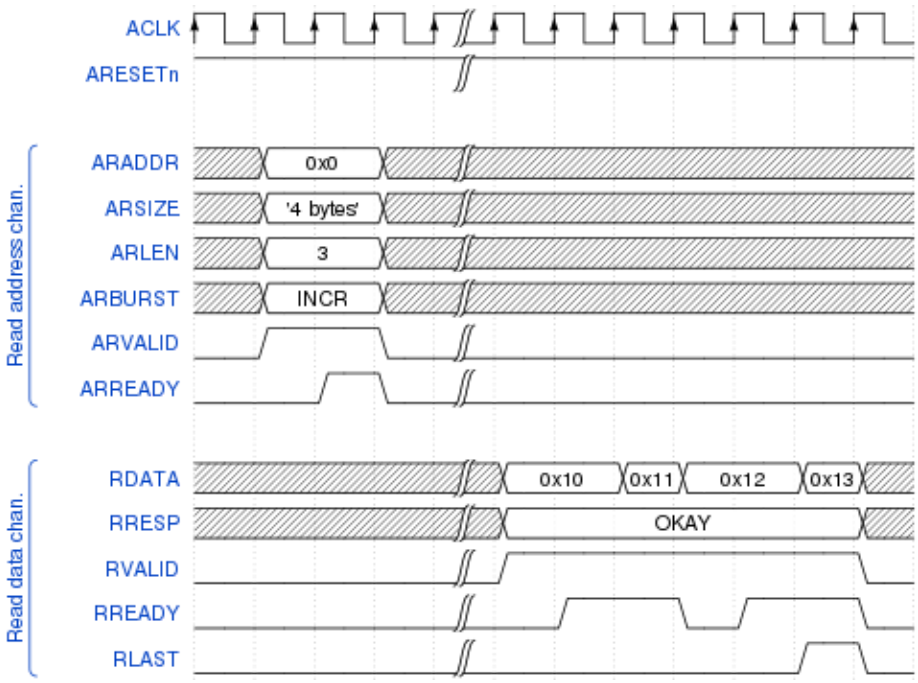
- the start address on ARADDR
- the burst type, either FIXED, INCR or WRAP, on ARBURST (if present)
- the burst length on ARLEN (if present).

Additionally, the other auxiliary signals, if present, are used to define more specific transfers.

After the usual ARVALID/ARREADY handshake, the target has to provide on the Read data channel:

- the data corresponding to the specified address(es) on RDATA
- the status of each beat on RRESP

plus any other optional signals. Each beat of the target's response is done with a RVALID/RREADY handshake and, on the last transfer, the target has to assert RLAST to inform that no more beats will follow without a new read request.




Example of an AXI read transaction. The initiator requests 4 beats (ARLEN + 1[13]) of 4 Bytes each starting from address 0x0 with INCR type. The target returns 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8 and 0x13 for address 0xc, all with the OKAY status. Only the most relevant signals are shown here.

### Writes

To start a write operation, the initiator has to provide both the address information and the data ones.

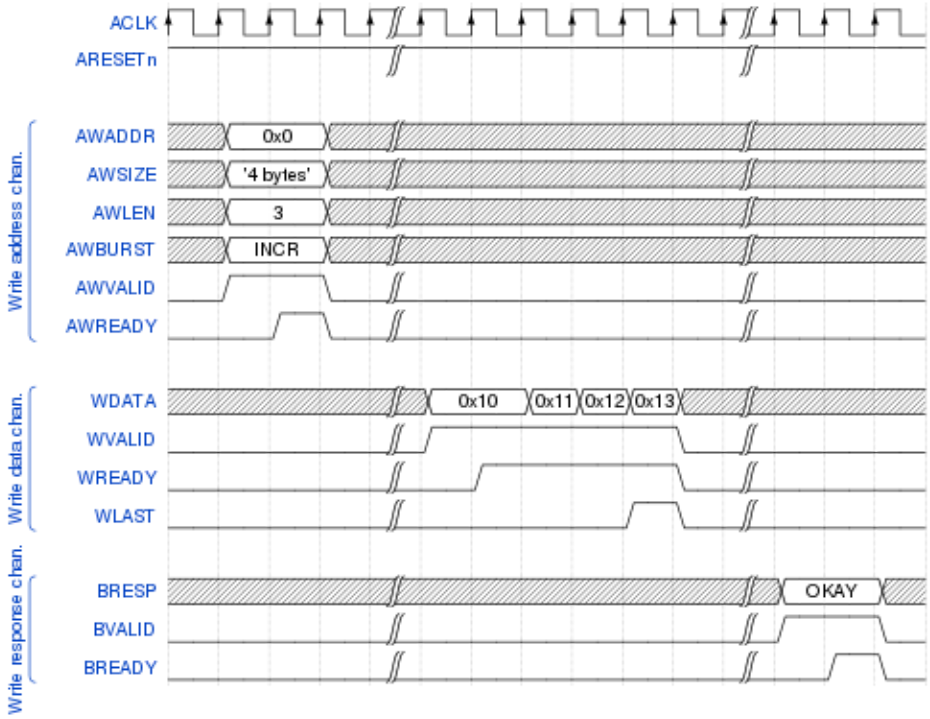
The address information are provided over the Write address channel, in a similar manner as a read operation:

- the start address has to be provided on AWADDR
- the burst type, either FIXED, INCR or WRAP, on AWBURST (if present)
- the burst length on AWLEN (if present)

and, if present, all the optional signals.

An initiator has also to provide the data related to the specified address(es) on the Write data channel:

- the data on WDATA
- the "strobe" bits on WSTRB (if present), which conditionally mark the individual WDATA bytes as "valid" or "invalid"




Example of an AXI write transaction. The initiator drives 4 beats (AWLEN + 1[13]) of 4 Bytes each starting from address 0x0 with INCR type, writing 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8 and 0x13 for address 0xc. The target returns 'OKAY' as write response for the whole transaction. Only the most relevant signals are shown here.

Like in the read path, on the last data word, WLAST has to be asserted by the initiator.

After the completion of both the transactions, the target has to send back to the initiator the status of the write over the Write response channel, by returning the result over the BRESP signal.

# AXI4-Lite

AXI4-Lite is a subset of the AXI4 protocol, providing a register-like structure with reduced features and complexity.[14] Notable differences are:

- all bursts are composed by 1 beat only
- all data accesses use the full data bus width, which can be either 32 or 64 bits

AXI4-Lite removes part of the AXI4 signals but follows the AXI4 specification for the remaining ones. Being a subset of AXI4, AXI4-Lite transactions are fully compatible with AXI4 devices, permitting the interoperability between AXI4-Lite initiators and AXI4 targets without additional conversion logic.[15]

## Signals

| Write address channel | Write data channel | Write response channel | Read address channel | Read data channel |
|---|---|---|---|---|
| AWVALID | WVALID | BVALID | ARVALID | RVALID |
| AWREADY | WREADY | BREADY | ARREADY | RREADY |
| AWADDR | WDATA | BRESP | ARADDR | RDATA |
| AWPROT | WSTRB | | ARPROT | RRESP |

[16]

# AXI-Stream

# See also

- Advanced Microcontroller Bus Architecture
- Wishbone (computer bus)
- Master/slave (technology)

# References


1. "AMBA | Documentation" (https://developer.arm.com/architectures/system-architectures/amba). Arm Holdings.
2. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 109–118. Retrieved 5 July 2019.
3. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 23–24. Retrieved 5 July 2019.
4. "AMBA AXI4 Interface Protocol" (https://www.xilinx.com/products/intellectual-property/axi.html). *www.xilinx.com*. Xilinx Inc.
5. "AXI4 IP" (https://www.xilinx.com/products/intellectual-property/axi/axi4_ip.html). *www.xilinx.com*. Xilinx Inc.

6. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 37–38. Retrieved 5 July 2019.
7. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 22–23. Retrieved 5 July 2019.
8. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 40. Retrieved 5 July 2019.
9. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 38. Retrieved 5 July 2019.
10. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 28–34. Retrieved 5 July 2019.
11. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 22. Retrieved 5 July 2019.
12. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 45–47. Retrieved 5 July 2019.
13. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 44. Retrieved 5 July 2019.
14. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. pp. 121–128. Retrieved 5 July 2019.
15. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 124. Retrieved 5 July 2019.
16. Arm Holdings. "AMBA AXI and ACE Protocol Specification" (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf) (PDF). *developer.arm.com*. p. 122. Retrieved 5 July 2019.

## External links

- AMBA webpage (https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications)
- AXI4 specification (https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf)
- ARM AXI introduction (https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface)
- Xilinx AXI introduction (https://www.xilinx.com/products/intellectual-property/axi.html)