AXI – **A**dvanced e**X**tensible **I**nterface

AXI4 (pronounced ack-see four) is a subset of the AMBA (Advanced Microcontroller Bus Architecture) specification developed by ARM. AXI4 was developed as a way for ARM to address communication issues between digital systems which were rapidly increasing in complexity. Due to its speed, error handling protocols, and ease of use, AXI4 has become an industry standard for computing systems across a variety of domains.

While AXI5, the latest generation of the interface, has been released, AXI4 remains the standard used across industries. For this reason, we will be focusing on AXI4, hereafter referred to as AXI.

The AXI specifications defines two types of users, Manager and Subordinate (previously Master and Slave). The characteristic which differentiates which user you are is whether you *initiate* transactions. An example would be a microcontroller and its memory. In this case the microcontroller is the Manager as it *initiates* all transactions between itself and the memory. Even though the memory sends data to the microcontroller, it is considered a Subordinate because it can only respond to requests which are sent by the microcontroller. In this way, we can classify all systems which use the AXI protocol. It is important to note that a device can be the Manager in relation to some systems while simultaneously being Subordinate to other Managers.

AXI comes in two flavors, bus transactions (AXI Full/AXI Lite) and streaming transactions (AXI Stream). Bus transactions employ the typical address based read/write commands while streaming transactions have a sender and a receiver. An example of a bus transaction model would be the aforementioned microcontroller and its memory. When initiating a transaction, the Manager specifies a target address, whether the transaction is a read or a write, and the respective data if it is a write transaction. Conversely, an example of a streaming transaction would be data being generated by a video camera being sent to a signal processing block for conditioning. The transaction has a distinct sender (the camera) and a unique receiver (the DSP block) and data is continuously being sent in one direction.

Bus Transaction Protocol – AXI Full / AXI Lite

Most systems we deal with are bus transaction systems, therefore we will spend some time developing an understanding of how this protocol works. As mentioned before, there are two subsets of the AXI bus transaction model – AXI Full and AXI Lite. As the name suggests, AXI Lite is a pared down version of AXI Full. AXI Full is a fully functional read/write protocol used for high-speed high-performance communication and is capable of transferring large blocks of data with a variety of security, addressing, and read/write mode options. This protocol is beyond the scope of this course and so we will focus on AXI Lite.

AXI Lite is a powerful and reliable method of communication between digital devices while remaining relatively easy to understand. We will examine in detail how this protocol works and by the end of this tutorial you will be able to develop your own AXI Manager and AXI Subordinate modules in HDL. Note that the naming convention for AXI systems is very important, and in some cases, it is strictly enforced. A full list of signal names and their meaning is provided in the appendix.

Handshaking – Ready/Valid

The foundation of any realistic digital communication protocol is a ready/valid handshake. This is a ubiquitous concept and should be internalized as soon as possible to establish a solid grasp of digital communications. The underlying principle is as follows: For two systems to communicate effectively, sender (source) must know when the receiver (destination) is ready to receive the data, and the receiver must know that the data being sent is valid. The real-world analog can be demonstrated with this simple example.

Suppose two friends (John and Sally) are having a conversation. John wants to say something important to Sally, but before he tells her, he must be sure she is paying attention. She visually cues him that she is *ready to receive information* by looking at him (the *ready* signal). John then proceeds to tell Sally the message while looking directly at her, indicating that the message is meant for her (the *valid* signal). Afterwards both parties are confident that the message was sent and received successfully, indicating a successful transaction.

Consider whether this transaction would have been successful if either Sally or John had not given their respective ready/valid signal. In this analogy, had Sally been speaking on the phone or reading a book, then John would have to wait until she was paying attention before speaking, otherwise he couldn’t guarantee that the message was successfully received. Likewise, suppose Sally was looking at John, ready for information, but John was speaking with somebody else behind him. Sally would have to wait until John turned around and addressed her directly before she could be sure that the information she was hearing was meant for her.

This concept, known as handshaking, establishes the basis for the AXI protocol. In all cases, the sender must assert a valid signal indicating that whatever data is being sent is good data while the receiver must likewise assert a ready signal, indicating that it is ready to receive data. The key takeaway is that a transaction is only considered to be successful when the *ready* and the *valid* signal are asserted in the **same** clock cycle. It is very important to remember that the ready and *valid* signals are **NOT** specific to Managers or Subordinates, but instead to sender (source) and receiver (destination). So far our model looks like this.



Read Transactions

Because the AXI Lite protocol is a bus transaction model, it has to be able to perform data reads and data writes. We will examine how reads are accomplished first. For a Manager to successfully request data from a Subordinate, it must first supply the address of the data it is interested in. This is the *ARADDR* signal. As mentioned above, for the Subordinate to know that the address being supplied is correct, the Manager must assert an associated valid signal, *ARVALID*. In order for the Manager to know that the Subordinate is ready to receive the address, it must wait until the Subordinate asserts the ready signal, *ARREADY*. These three signals, *ARADDR, ARVALID,* and *ARREADY,* comprise all necessary signals to guarantee that the read address has been transmitted.

Furthermore, the AXI specification supplies one more signal to facilitate secure transactions, *ARPROT*. This signal indicates who has permission to view the contents of the transaction. While this signal is included in the AXI protocol, we can safely ignore it for now. So far, our read address interface has four signals and will be referred to as the ***Read Address Channel***.



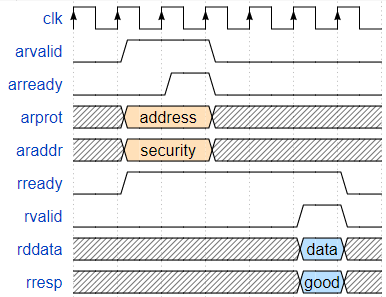
Now that the address has been successfully supplied, we must deal with the data that the Subordinate will send to the Manager. Similar to the ***Read Address Channel***, we will need a ***Read Data Channel*** with its own set of handshaking signals. First and foremost we must have a data signal, *RDATA,* which the Subordinate will use to send the requested data. In keeping with our handshaking protocol, this must be accompanied with a valid signal, *RVALID*. Similarly the Manager must supply its own ready, *RREADY*. Lastly, the Subordinate supplies a signal indicating whether the read transaction was successful or not. For example, if the address specified does not exist, or the Manager does not have the necessary permissions the Subordinate could tell the Manager using this last signal *RRESP*. With these four signals, we complete the ***Read Data Channel*** and our interface now looks like this.



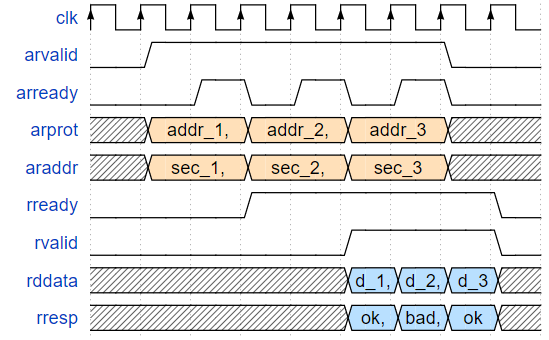
Before moving on to the rest of the channels, we will take a moment to discuss the timing related to this protocol. As mentioned before, the transaction can only take place if **both** the *ready* and *valid* flag are asserted in the same clock cycle. In order to facilitate this transaction and to avoid deadlocks, there are some golden rules that must be adhered to for this to work. The channel specific dependencies are publicly available online, but for the scope of this introduction we state the following:

* The *valid* signal CANNOT be dependent on its corresponding *ready* signal. Meaning that it cannot wait until it sees a *ready* signal from the receiver before asserting itself
* The *ready* signal MAY be dependent on its corresponding *valid* signal. Meaning that the designer may choose to allow the *ready* signal to wait until a *valid* signal is seen before asserting itself.

With that out of the way, can look at an example read transaction.



In the timing diagram above, we see two handshaking procedures taking place, one for the address and one for the actual data. The important thing to note is that the transaction occurs in the clock cycle when both the *ready* and *valid* signals are asserted. Immediately after, the handshake occurs, both the sender and receiver de-assert their signals if they have no further transactions to make. The following timing diagram is equally valid and is used to further demonstrate this idea.



In the timing diagram above, we see that even though a transaction has successfully taken place, the *arvalid* signal remains asserted because the Manager is requesting more read transactions. Note that the data associated with each *valid* signal must be available and, on the line, as soon as *valid* goes high.

Write Transactions

Now that we have established the fundamentals of the AXI protocol, we can expand our knowledge to the write transaction. In the same way that the read transaction requires a *ready/valid* handshake for both the read address and the read data, the write transaction will need two independent channels as well; **Write Address Channel** and **Write Data Channel**.

Each channel is analogous to its counterpart in the read request. For the **Write Address Channel** we have the following signals, *AWVALID, AWREADY, AWADDR,* and *AWPROT*, with each signal performing the exact same addressing functionality as the **Read Address Channels**.

For the **Write Data Channel**, we have one key difference that I would like to highlight. Instead of having a response signal associated with the write data (like we have *RRESP* for the read data), we replace it with a masking signal called *WSTRB****.*** This stands for write strobe, and it acts as a byte mask for the write data coming in. Because the data bus is measured in bytes, we can specify which bytes hold valid data to be written in.

For example, if we want to write the 16-bit data word xABXX (1010 1011 xxxx xxxx), where the last two bytes do not hold valid data, we would set the *WSTRB*signal to xC (1100). From this example we can derive two things. Firstly, each bit in the *WSTRB* signal corresponds to a byte in the *WDATA* where a 1 indicates valid and a 0 indicates invalid data, and secondly, the length of *WSTRB* will always be DATA\_BITS/8. This is a useful feature for more complex interfaces, but for the sake of simplicity we will design our interface to ignore it. Our Interface now looks like this.



There is one more piece of the puzzle figure out before we have a fully developed AXI interface. By replacing the write response signal with the *WSTRB* signal, the Manager has no way of knowing whether the write transaction was successful or not. In order to address this, AXI provides one more channel, called the **Write Response Channel**. This channel is contains a write response signal and its own set of *ready/valid* signals, resulting in three signals total. Adding it to the interface, we now have the full and final interface.



Which we can further abbreviate to the following.



Next, we provide a table of all the signals and their attributes and functions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Signal Name** | **Bit-Width** | **Channel** | **Function** | **Direction** |
| ACLK | 1 | N/A | AXI clock | N/A |
| ARESETN | 1 | N/A | Active low AXI reset | N/A |
| ARVALID | 1 | Read Address | Indicates valid read address on the line | M -> S |
| ARREADY | 1 | Read Address | Indicates subordinate is ready for read address | S -> M |
| ARADDR | Log2(WL)+2 | Read Address | Byte aligned read address (2 LSBs set to zero) | M -> S |
| ARPROT | 3 | Read Address | Indicates security/privilege level of read | M -> S |
| RVALID | 1 | Read Data | Indicates valid read data on the line | S -> M |
| RREADY | 1 | Read Data | Indicates manager is ready for read data | M -> S |
| RDATA | WL | Read Data | Read data | S -> M |
| RRESP | 2 | Read Data | Indicates status of read transaction | S -> M |
| AWVALID | 1 | Write Address | Indicates valid write address on the line | M -> S |
| AWREADY | 1 | Write Address | Indicates subordinate is ready for write address | S -> M |
| AWADDR | Log2(WL)+2 | Write Address | Byte aligned write address (2 LSBs set to zero) | M -> S |
| AWPROT | 3 | Write Address | Indicates security/privilege level of write | M -> S |
| WVALID | 1 | Write Data | Indicates valid write data on the line | M -> S |
| WREADY | 1 | Write Data | Indicates manager is ready for write data | S -> M |
| WDATA | WL | Write Data | Write data | M -> S |
| WSTRB | WL/8 | Write Data | Byte mask for write data | M -> S |
| BVALID | 1 | Write Response | Indicates valid write response on the line | S -> M |
| BREADY | 1 | Write Response | Indicates manager is ready for write response | M -> S |
| BRESP | 2 | Write Response | Indicates status of write transaction | S -> M |