SoC Bus and Interconnect Protocols

Simon Southwell

September 2022

Preface

This document brings together two articles written in September 2022 and published on LinkedIn, that is a look at Bus and interconnect protocols used processor based in SoC environments. The main focus is on AMBA bus protocols, as these are ubiquitous in the industry, with a review of some other bus protocols to finish.

Simon Southwell
Cambridge, UK
September 2022

Contents

Part 1: Busses (APB and AHB)

Introduction

Not so long ago I was in a meeting with a representative of a supplier (FPGA, tools, and support services), along with a manager and another engineer. In the discussion we were looking at features of both Intel (which we were using) and Xilinx FPGA products when it was mentioned that Xilinx uses AXI bus for interconnect. (Intel products do too, but one may use the Intel/Altera Avalon protocol in its place.) At this point the other engineer launched into, what I can only describe as, a rant about how complicated AXI was and how difficult it was to use compared to Avalon, as if it had been deliberately done this way to foil would-be engineering users. This notwithstanding that I was using AXI in our designs to have access to cache coherent data from the FPGA logic as this was not available using Avalon (see my article on caches regarding coherency issues).

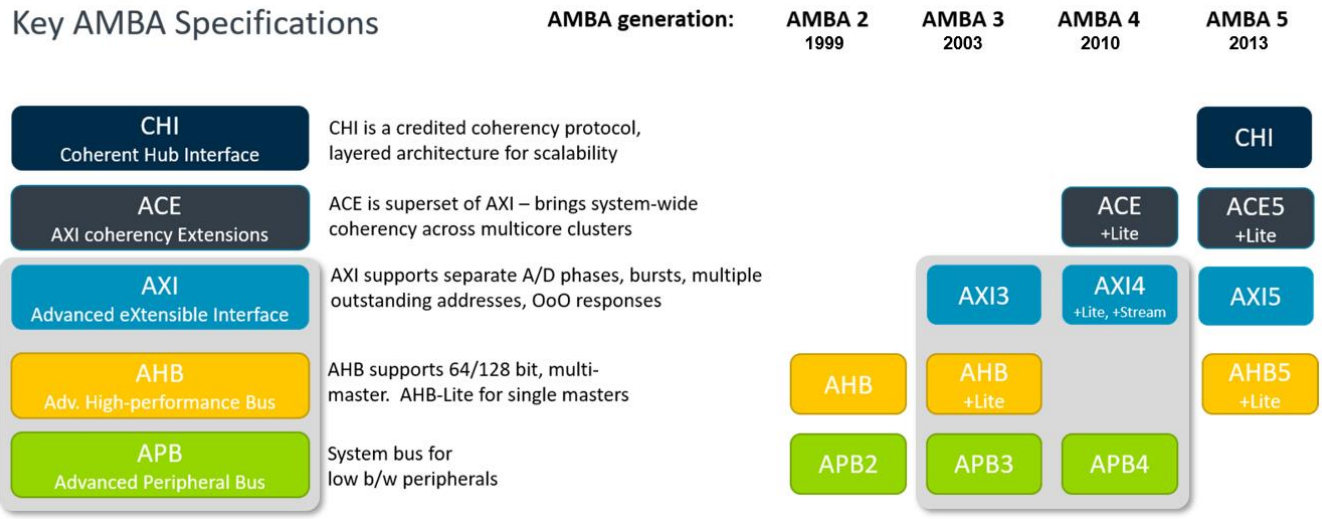
I'm sure the rep was not responsible for the AXI protocol, or in a position to do much about it. That said, I want, in this document, to dispel this misconception that AXI, or other bus protocols, are deliberately obtuse. The good people at ARM (just a few klicks from where I'm writing), I believe, have done a fine job of creating bus protocols that meet the requirements of processor based embedded systems (and beyond), with features added only where necessary to create an efficient system. Not only that, they have carefully allowed certain features to be optionally used so that only the complexity actually required need be considered. I have been working with ARM based system for many years now (decades, even), and have been grateful that certain features were an option I could use (such as the cache coherent accesses I mentioned before).

If one comes to a specification that is the result of decades of development and refinement without any prior knowledge, then this can be overwhelming. However, breaking a specification down into its individual features, which ones are required and which optional, and understanding what problem each feature is trying to solve, then the problem becomes more manageable. In this document, then, I want to use the ARM AMBA bus protocols as a case study as these are the most likely encountered and have all the features I want to talk about. So, we will start with APB for peripherals before moving to AHB for higher speed (both protocols which I have used in many SoC designs). Then we will move to AXI in part 2 of the document and will see that there has been an evolutionary path from these other protocols,

building to this interconnect specification. To conclude, I will review some alternative protocols along with the ARM coherent hub interface (CHI) next generation interconnect specification. By building up in steps, I hope to make the journey to AXI comprehensible.

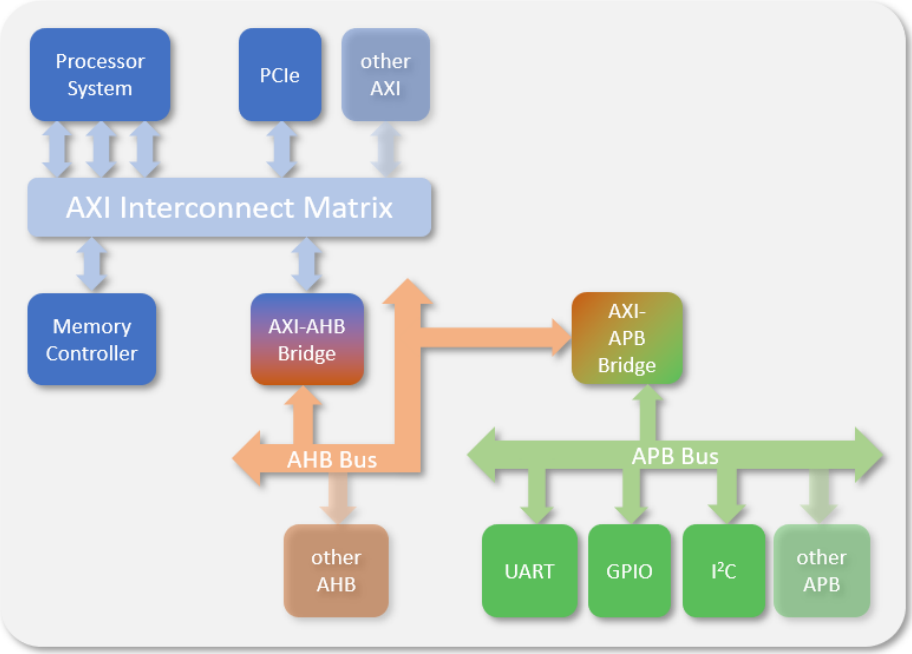
AMBA Overview

The diagram below shows the different protocols for the AMBA (Advanced Microcontroller Bus Architecture) suite of interconnects and their revisions.



In this document we will concentrate on AMBA versions 3 and 4, though version 5 will be mentioned, and we will stick to APB, AHB and AXI, but CHI features will be reviewed. The Advanced Peripheral Bus (APB) is meant for low power, low bandwidth peripherals. For example, an I^2C controller where the serial protocol is low bandwidth compared to the main SoC system, so the interface to the system bus does not need to be high bandwidth. The Advanced High-performance Bus (AHB) is a true bus protocol and meant for high bandwidth interconnect and peripherals. For example, a 100Mbps ethernet controller. Before AXI, AHB was the mainstay of a system bus in an ARM based system. To meet the needs of higher performance systems, with multiple cores and parallel high speed data transfer requirements the Advanced Extensible Interface (AXI) protocol was developed. This is an interconnect rather than a true bus, and devices are connected through an interconnect matrix—a small switching network, if you will.

Systems may have a mixture of these protocols, including AHB for reuse of legacy components with this interface, and these are usually arranged in a hierarchy, with bridges connecting different busses. A typical structure of an SoC based around AMBA bus protocols might look something like the following diagram:



The diagram shows the main system bus is AXI, with its interconnect fabric. A bridge connects one of the AXI ports to an AHB bus, though multiple busses might be supported. On the AHB bus, another bridge connects an APB bus, though it might have been connected directly to an AXI interconnect port. In general, though, there would be a hierarchy, with lower bandwidth and latency tolerant peripherals further away from the main system bus, and highspeed, low latency peripherals nearer the main bus.

I want to build up to AXI in steps from the simplest protocols, so that it is clearer what each added functionality the more complex protocols bring, and the problems they are trying to solve, so that I'm not jumping straight to a list of all the AXI signals with no context. So, let's start with APB.

APB

APB is the simplest of the AMBA protocols, and the one I've probably added to IP I was developing the most. By its very nature, it does not support advanced features meant for high-speed efficient data transfer, as it is assumed peripherals requiring such features do not use APB. Therefore, it only supports single word transfers, and there is no multiple word 'burst' transfers. It also does not support 'split transactions', that is a request for data is completed, but the data is sent at some future time. A whole transfer is completed before another can be started, so there are no 'overlapping' transfers, and it can be big- or little-endian (the default). There is optional support for byte writes (when the bus width is wider than a byte), for some

protection modes and user defined attributes. The use of optional features we shall see is common in the protocols, some of which were added in later specifications. This is useful for simplifying interface design for IP that does not, or cannot, use these features, or interfacing to peripherals of an older specification, even if the manager interface (ARM uses the terms manager and subordinate) has the optional features.

Signals

So, the APB signals are defined as mandatory or optional. The table below shows the mandatory signals.

signal	size	description
PCLK	1	Clock signal, rising edge timing
PRESETn	1	Reset signal (usually sys. reset)
PSEL*x*	1 (per peripheral)	Select
PADDR	ADDR_WIDTH up to 32 bits	Byte address
PENABLE	1	Enable. High 2nd cycle onwards
PWRITE	1	Write access when high
PWDATA	DATA_WIDTH: 8, 16 or 32 bits	Write data
PREADY	1	Ready (APB3)
PRDATA	DATA_WIDTH; 8, 16 or 32 bits	Read data
PSLVERR	1	Transfer error. Optional on O/Ps

The clock (PCLK) and reset (PRESETn) are global signals, and all the peripherals on the bus will connect to the same sources. The reset is often connected to the system reset, but may be different if separate APB resetting required, or if clock domains are different. Each peripheral on the bus will have an individual select line (PSEL), and some address decode logic will generate a mutually exclusive select when a peripheral is being accessed.

The other signals are the data transfer signals and transfer control. This is for single word, memory mapped transfers, so there is a write-not-read strobe (PWRITE) with an address (PADDR), and then separate read and write data busses (PRDATA and PWDATA). The width of the address and data busses is variable, defined with a

parameter as shown in the table. The address width is up to 32 bits, whilst the data width can be 8, 16, or 32-bits.

The PENABLE signal, along with PREADY, is used to 'sequence' the transfer, as we will see shortly. The final mandatory signal is PSLVERR. This is a legacy misnomer, as ARM refers to manager and subordinate for requesters and completers. This flags any error in the transfer (read or write) by the subordinate. This signal is mandatory for a manager device but is optional on a subordinate. If a subordinate does not have a PSLVERR output, then the manager signals is tied low.

Of the optional signals, these have been introduced in AMBA 4 and 5. These signals are shown in the table below:

signal	size	description
PPROT	3	Protection (norm, priv, secure; APB4)
PSTRB	USER_DATA_WIDTH/8	Write strobe (byte enables; APB4)
PWAKEUP	1	Wake up (peripheral active, APB5)
PAUSER	USER_REQ_WIDTH	User request attribute (APB5)
PWUSER	USER_DATA_WIDTH	User write data attribute (APB5)
PRUSER	USER_DATA_WIDTH	User read data attribute (APB5)
PBUSER	USER_RESP_WIDTH	User response attribute (APB5)

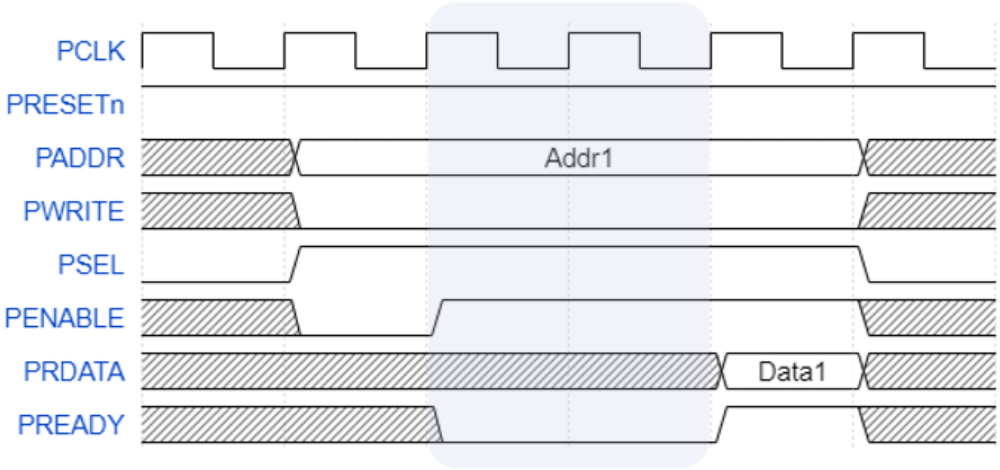
The 3-bit PPROT signal, generated by the manager indicates the protection privilege level—normal, privileged, or secure—if the peripheral can make distinctions between these levels. The PSTRB signal is effectively byte enables for the write data bus. This can only be present if DATA_WIDTH is greater than 8, where each bit corresponds to a write enable for the byte lane on PWDATA.

The other signals (in red) were all introduced in AMBA 5, so we will just briefly discuss them here. PWAKEUP is generated by a manager for a subordinate that supports different power states to indicate that activity is on the bus. It is asynchronous to PCLK. The user signals are all undefined and for any additional custom signalling that a design might wish to have. Each has its own (bounded) parameter to define its width.

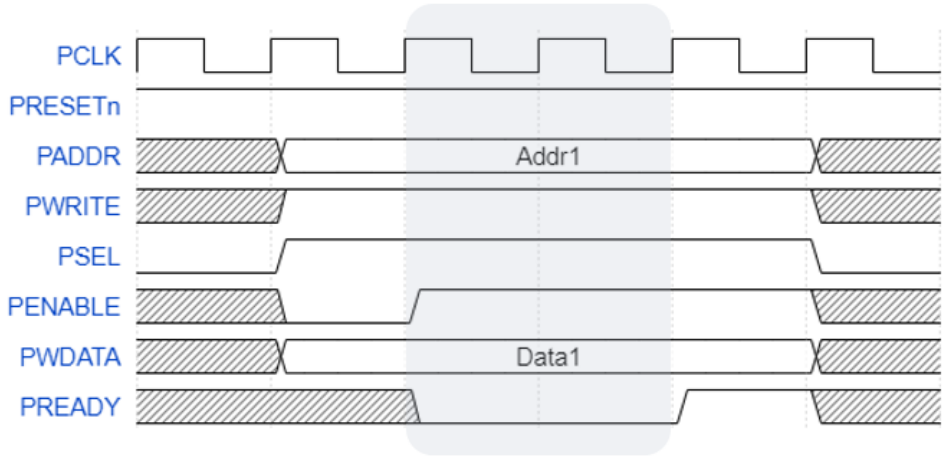
Transfers

APB transfers are two-phase, with the ability to insert wait states by the subordinate, using the PENABLE and the PREADY signals. At the start of a transfer, a peripheral's PSEL line is asserted, and PENABLE is set low by the manger to indicate the first phase. The address is set, with PWRITE either high or low to indicate a write or a read. The optional PPROT signal is also set in phase 1. The PSEL, PADDR, and PWRITE will remain unchanged for the rest of the transfer (as is PPROT if present). If a write, then PWDATA is set, and also remains unchanged and, if PSTRB signalling supported, this would also be set in phase 1 and held.

In the cycle after PENABLE is low, PENABLE will go high to indicate the second phase. It is in the second phase that a peripheral can insert wait state. During the first phase, the subordinate's PREADY signal is a don't care, but will go low if wait state are to be asserted (for instance if there is no space to accept a write, or it takes multiple cycles to fetch read data). It doesn't have to set PREADY low if it can complete the transfer immediately. When PENABLE is high, and the PREADY is also high, then phase 2 completes and the transfer is finished. The use of PSEL and PENABLE means that a transfer to the same peripheral can begin in the next cycle. In that case, PSEL remains high, but PENABLE is deasserted to indicate a new phase 1. The diagram below shows a read transfer with wait states (shaded).



The subordinate inserts two wait states by de-asserting PREADY for 2 cycles when PENABLE goes high. It then asserts PREADY, along with the read data (PRDATA), to complete the transfer. Not shown in the diagram, but if the peripheral had a PSLVERR signal, this would be asserted (if an error occurred) when PREADY high, and PRDATA would be a don't care. A write transfer, with two wait states, is shown below:



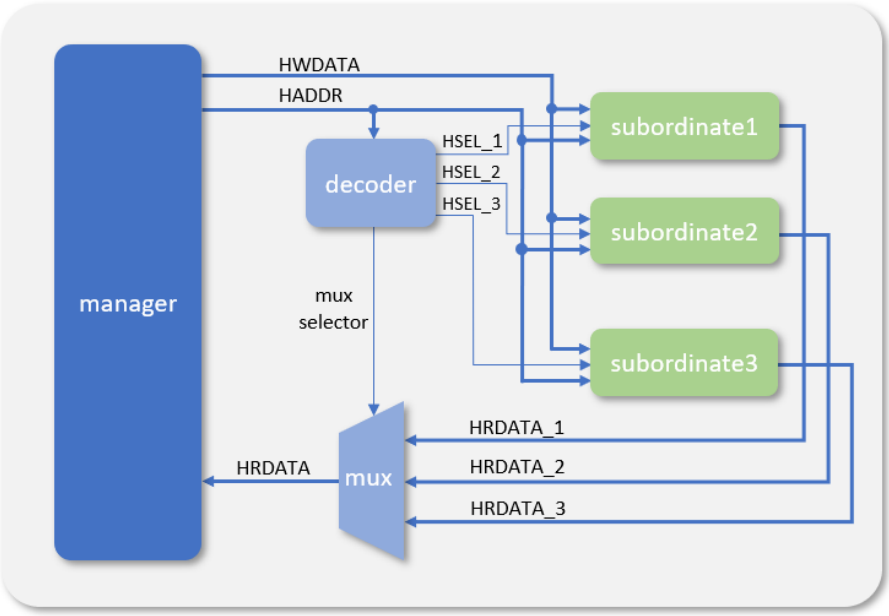
The signalling and timings are almost identical to a read, but `PWRITE` is high instead of low, and `PWDATA` is set in phase 1, and remains unchanged (`PRDATA` from the subordinate is a don't care). If `PSTRB` byte enables are supported, this is asserted in phase 1 and held for the whole transfer, along with the optional `PPROT` if present.

This, then, is all that's needed for an APB interface. A simple memory mapped word transfer interface with some configurability, optional signalling, and a straight-forward two-phase protocol with simple flow control.

AHB

The Advanced High-performance Bus has many of the characteristics of the APB bus, with some signalling that's very similar. Now, though, the protocol must be able to move bulk data efficiently and, being nearer the processor system, must support features that the processor system might need. This includes things like support for locked, secured, or exclusive transfers. For even better efficiency, transfers can overlap, though the basic transfers are much like for APB, and the general structure very similar. The diagram below shows the basic transfer architecture, and could equally apply to APB (with different signal names):

Simplified AHB configuration



Signals

So, let's start with what is the same on AHB as for APB. The table below shows the signals:

signal	size	source	description
HCLK	1	global	Clock signal, rising edge timing (cf. PCLK)
HRESETn	1	global	Reset signal (usually system reset. cf. PRESETn)
HSELx	1	global	Select (cf. PSELx)
HADDR	ADDR_WIDTH	manager	Byte address (cf. PADDR)
HWRITE	1	manager	Transfer direction (cf. PWRITE)
HWDATA	DATA_WIDTH	manager	Write data (c.f. PADDR)
HRDATA	DATA_WIDTH	subordinate	Read data (cf. PRDATA)
HREADY	1	subordinate	Ready signal (cf. PREADY)
HRESP	1	subordinate	Response (error status: cf. PSLVERR)
HPROT	HPROT_WIDTH	manager	Protection type (optional)
HWSTRB	DATA_WIDTH/8	manager	Write strobes (cf. PSTRB). AMBA 5 only.

As shown in the table, these 'common' signals map almost one-to-one with APB signals. The exception is that there is no equivalent of a PENABLE signal, and

sequencing the transfer is done a little differently to cater for overlapping transfers (as we shall see). Like in APB, the protection signal (HPROT) is optional, and HWSTRB isn't introduced until AMBA 5 (hence red in the table).

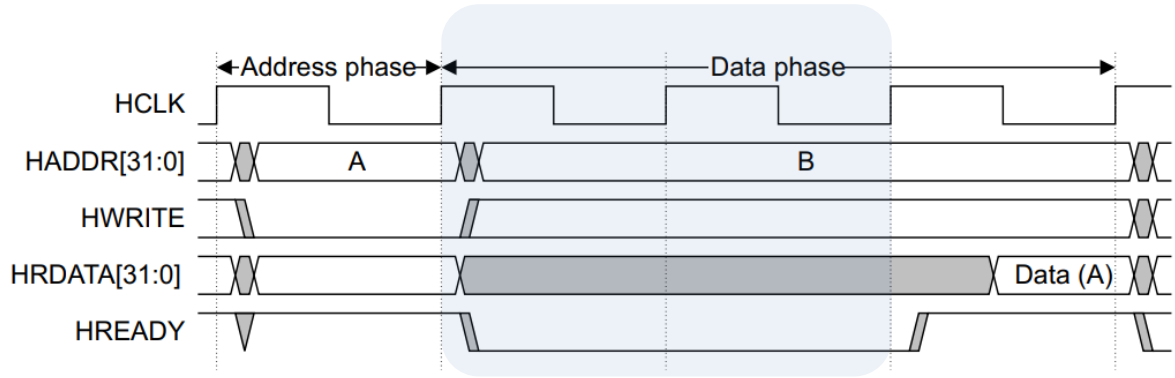
New signals are introduced over APB to determine the amount of data to be sent, and the nature of the order of the data. In addition, a transfer can be 'locked'—i.e., indivisible, to support atomic operations. AMBA 5 introduces some extra signals for secure and exclusive operations, as well as a manager ID, for managing access form multiple managers on a single bus. These optional signals are shown in the table below.

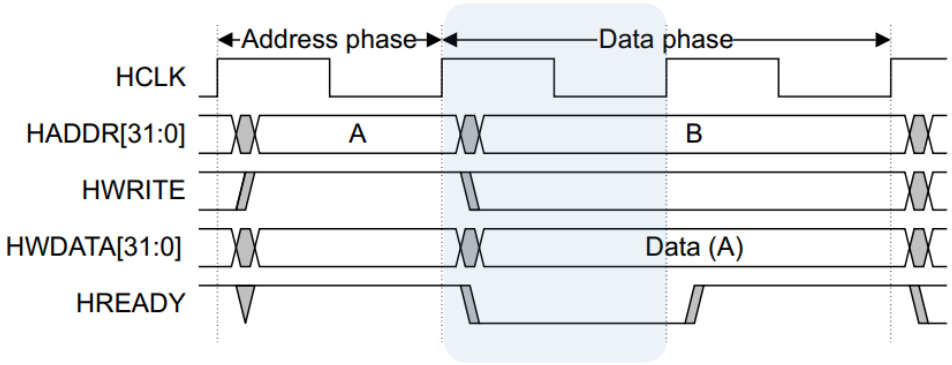
signal	size	source	description
HBURST	HBURST_WIDTH	manager	burst type (optional)
HSIZE	3	manager	Size of transfer
HTRANS	2	manager	Transfer type
HMASTLOCK	1	manager	Transfer part of locked sequence
HNONSEC	1	manager	Secure/non-secure transfer
HEXCL	1	manager	Exclusive access transfer
HMASTER	HMASTER_WIDTH	manager	Manager identifier (optional)
HEXOKAY	1	subordinate	Exclusive transfer okay

The first two signals (HBURST and HSIZE) are associated with the transfer of data and HMASTLOCK for locked transfers, used for things like semaphores. The two-bit HTRANS signal is what replaces PENABLE and sequences the transfers on the bus and the select line takes a less critical role, which we shall look at shortly. The other signals, in red, are for AMBA 5 and no more will be said about these here.

Transfers

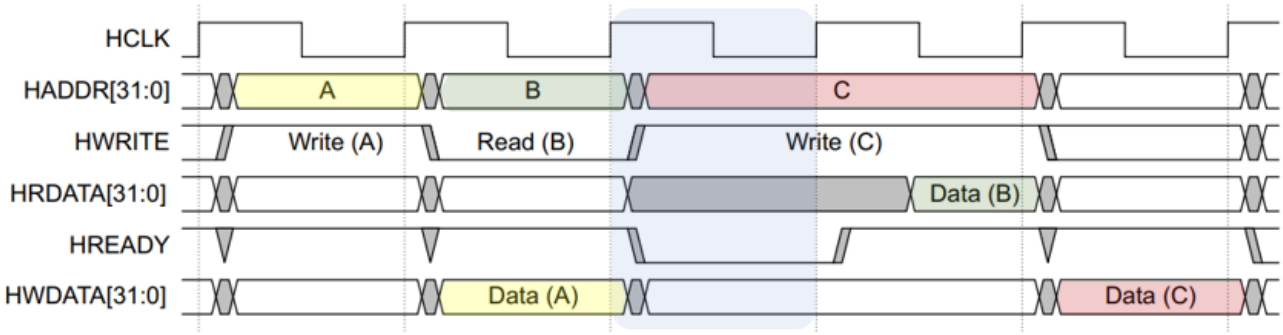
At its simplest, then, AHB can transfer single words much like APB. The diagrams below show the timings for single word accesses, firstly for a read, followed by the timing for a write:





For clarity, the HSEL signal is implied for the whole of the transactions, and for single cycle access HTRANS is set at a value of NONSEQ in the first phase, and then IDLE from then on, assuming no further transactions (see below). HADDR is only valid in the first phase, as shown, as are HWRITE and other manager control signals listed in the tables, when a single word transfer. Wait states are inserted by the subordinate using HREADY, just as for APB and PREADY. Not shown, but HRESP (like PSLVERR) is asserted at the end of the transaction if an error occurred.

The AHB protocol would be no better than APB if this was all that was possible. The first facility it has is that transfers can overlap. Since the address and general control signalling is valid in the first phase, then when data is transferred in the second phase, a new address and control may be asserted so that all cycles are used for data transfer, assuming no wait state inserted by the subordinate. The diagram below shows some overlapping transfer (including wait states):



Here we see a write to address A in the first cycle, the data is transferred in the second cycle, as HREADY was asserted. In that second cycle, a new read is instigated to address B. The subordinate de-asserts HREADY in the next cycle, so this is a wait state, but it's asserted in the following cycle, and the read data is returned. Another write, to address C, was instigated in the cycle after address B, but this is held whilst the wait state is active, and into the cycle with data for B returned. The data for address C is written in the cycle after this. Without the wait state, data would be transferred in consecutive cycles after address A is asserted, giving 100% efficiency.

The next feature above APB that AHB has available is that multiple words can be transferred on a single address 'command'—a burst. Until now, we have looked at a single word transfer, largely ignoring the `HTRANS`, `HBURST`, and `HSIZE` signals, which were more or less fixed (`HTRANS` was `NONSEQ`/`IDLE`, mirroring, somewhat, `PENABLE`).

The `HTRANS` signal, as mentioned before, sequences the transfers. The two-bit signal has four possible values:

- **`IDLE`**: indicates no transfer from the Manager. A subordinate must still return an OK response on `HRESP` (low) if selected with `HSEL`.
- **`BUSY`**: allows a *Manager* to insert wait states in the middle of a burst (which APB managers can't)
- **`NONSEQ`**: indicates the first transfer of a burst (or the only transfer of a single word transfer—effectively a burst of 1).
- **`SEQ`**: indicates subsequent transfers in a burst

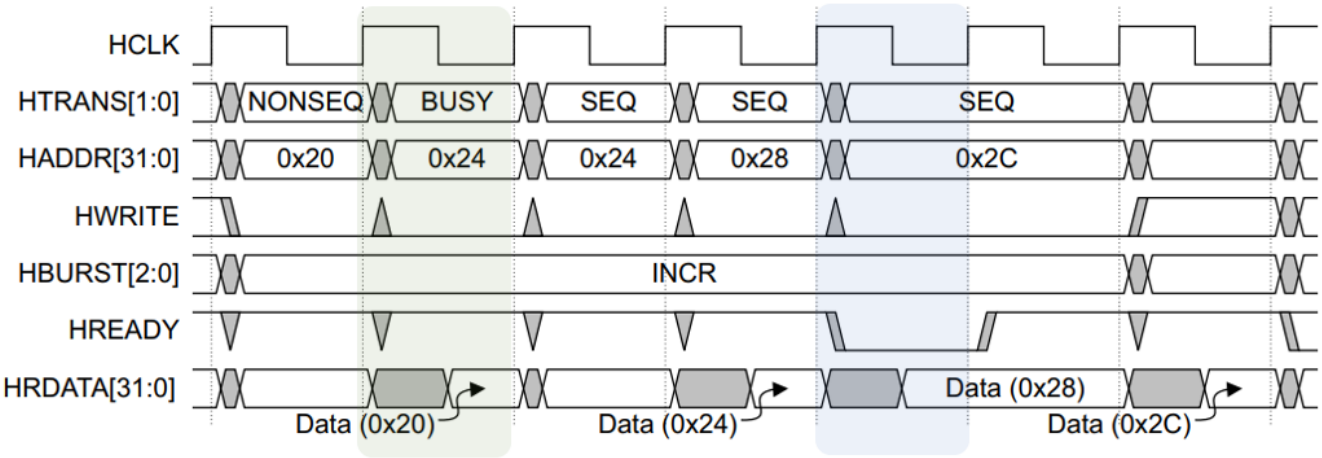
The number of words in a burst is determined by the `HBURST` signal. The simplest we have seen already, a single transfer. This is the default setting and, remembering that `HBURST` is optional, is what happens when this signal is not present. The other types of transfer are for an incrementing burst of undefined length, or fixed size burst of 4, 8, or 16 words, either incrementing or wrapping.

The table below shows the encodings for the burst types:

HBURST	size	description
3'b000	single	Single transfer burst
3'b001	incr	Incrementing burst of undefined length
3'b010	wrap4	4-beat wrapping burst
3'b011	incr4	4-beat incrementing burst
3'b100	wrap8	8-beat wrapping burst
3'b101	incr8	8-beat incrementing burst
3'b110	wrap16	16-beat wrapping burst
3'b111	incr16	16-beat incrementing burst

The incrementing types are fairly self-explanatory, though a restriction of an undefined length burst is that it must not cross a 1Kbyte boundary. For all the incrementing types, the address must be aligned with the width of the data word. For example, if a 32-bit wide data bus is defined, the address bottom 3 bits must be 0. At each data transfer, the address increments the appropriate number of bytes.

For the wrapping burst types, the addresses still increment by default, but will wrap at the address boundary determined by the wrap length. For instance, if a wrap4 type is selected, and the first address is 0x34, the address sequence is `0x34`, `0x38`, `0x3c`, `0x30`. The address signal `HADDR` follows these incrementing or wrapping address value—it is not implied after the command and the subordinate need not work out the address sequence (though it could). The diagram below shows a four-word incrementing burst read transfer:



It is assumed the HSEL is active in the first cycle, and for the entire transfer. Also not shown, but for every word transfer an HRESP is required. The transaction is started with HTRANS as NONSEQ and an address of 0x20. The transfer is a read since HWRITE is 0, and HBURST indicates an incrementing burst of undefined length. In the second cycle, data is returned from the subordinate, but also HTRANS is now BUSY, inserting a wait state from the manager. The transfer is resumed in the next cycle with HTRANS at SEQ, indicating subsequent data transfers in the burst. This proceeds for addresses 0x24 and 0x28, then a subordinate wait state is inserted with HREADY being raised on the 0x2c address, which is held until the next cycle. The data for 0x2c is returned in the last cycle.

What happens next depends on what the signals are for the last cycle. If HSEL was de-asserted, then the transfer has finished. If HSEL remains asserted, but HTRANS is IDLE, then no new commands have been issued. With HSEL asserted and an HTRANS value of NONSEQ, then a new burst (or single word transfer) is started, with any new start address required. Only if an HTRANS of SEQ would the current burst continue (possibly after IDLEs). Note that, for a fixed burst size, this can be terminated early with a new NONSEQ command before all the data transferred.

As stated previously, the width of the data busses is fixed by a parameter DATA_WIDTH. Although it is recommended that this is 32-bits, it can be set as little as 8 bits (1 byte), or as much as 1024 bits (128 bytes). Data less than the width of the bus may still be transferred if HSIZE is set to be less than this width. The table below shows the encodings for HSIZE.

HSIZE[2]	HSIZE[1]	HSIZE[0]	size (bits)	description
0	0	0	8	byte
0	0	1	16	halfword
0	1	0	32	word
0	1	1	64	double word
1	0	0	128	4-word line
1	0	1	256	8-word line
1	1	0	512	16-word line
1	1	1	1024	32-word line

The timing for HSIZE is that it is set in the first phase and held for all the address phases of the transfer. This signal, along with HADDR and HBURST determine which lanes on the bus are active. For example, on a 32-bit bus, if an incrementing burst, HSIZE indicates a half-word transfer (001b) and the address lower 2 bits are 10b, then the upper two lanes are active. In addition, for writes, the HWSTRB (introduced in AMBA 5) can modify 'active' lanes to not be written, so that non-consecutive bytes are updated. This is called sparse writing.

The last AHB signal to discuss is HPROT. This is similar to the PPROT signal we saw for APB, but with some slight differences. For AMBA 3 this is a 4-bit signal but, by AMBA 5 this is extended to 7 bits. The first three bits are (largely) the same as AMBA 3, but the rest extend the control over cache accesses and whether transfers can be modified, when transiting through the bus structure. AHB, then, adds some control over cache coherent accesses with AMBA 5 extending this control. (For the importance of cache coherent accesses, see my article on caches.) Here we confine ourselves to looking at AMBA 3, and the table below shows the HPROT encodings:

HPROT[3] cacheable	HPROT[2] bufferable	HPROT[1] privileged	HPROT[0] data/opcode	Description
-	-	-	0	Opcode fetch
-	-	-	1	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Non-bufferable
-	1	-	-	Bufferable
0	-	-	-	Non-cacheable
1	-	-	-	Cacheable

The timing for HPROT is that it is set in phase 1 and held for the rest of the address command cycles for the transfer. This concludes the review of the AHB protocol.

Conclusions

In first part we have looked at two of the AMBA bus protocols, starting with the simplest (APB) and then looked at a higher performance bus (AHB) where additional features are added to cope with the needs of moving data within a processor base system more efficiently using burst and overlapping transfers, and addressing the

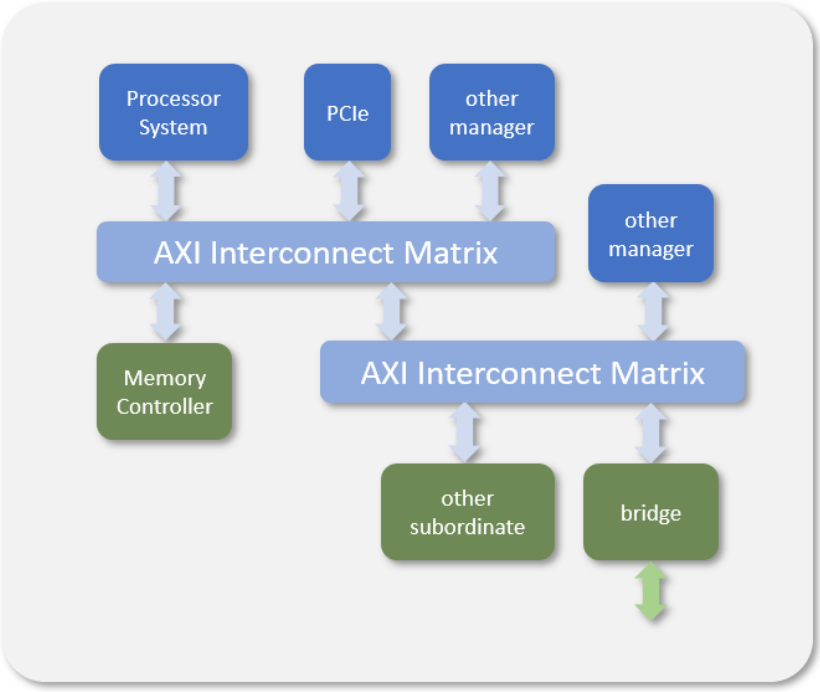
needs of such systems, such as requiring cache coherent accesses. The idea is to show a heritage from APB to AHB, with many common features.

As systems have evolved, with muti-core designs becoming common, AHB evolved to provide features for these, and AMBA 5 adds some of this over AMBA 3 specifications. However, the limitations of a bus-oriented system start to show, the more cores (or other manager components) that are required on the bus. Therefore, a new architecture based on a switching fabric is required, where multiple transfers from multiple managers is handled in parallel. The AXI bus was developed to do just this. As we shall see, this still has features that will be familiar from APB and AHB, but the architecture is somewhat different. We shall look at this, along with a review of alternative protocols and of AMBA protocols beyond AXI in the next part.

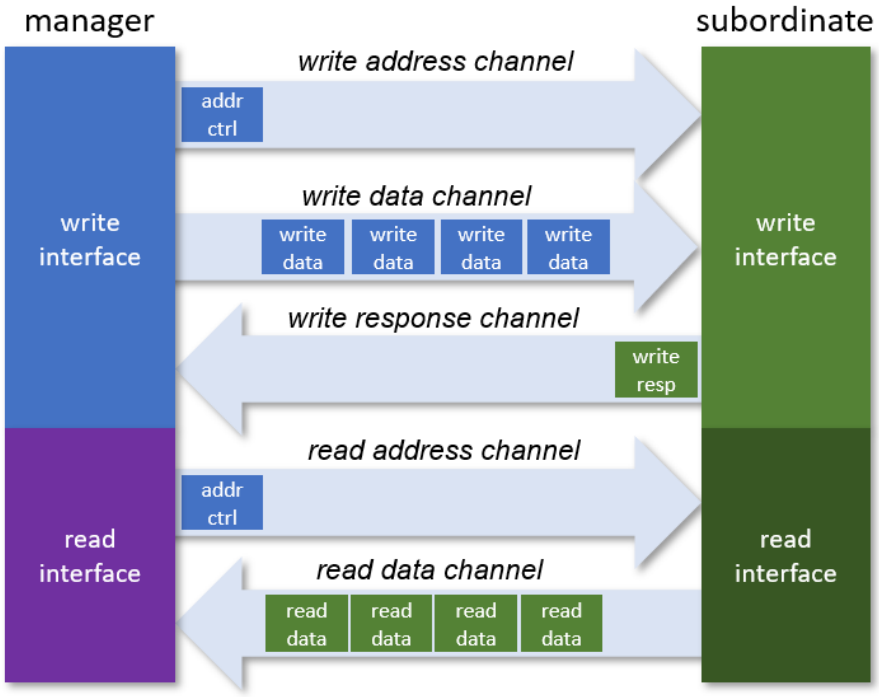
Part 2: Interconnect (AXI)

Introduction

In the first part we covered APB and AHB bus protocols. The defining common characteristic was that there were both true busses, with a single manager instigating transactions over the bus at any one time. The main differences between APB and AHB were some added features to AHB to allow overlapped and burst transfers. The assumption has been that there is only a single manager. AHB can, in fact, support multi-manager operation via an interconnect component that provides arbitration and routing signals from different managers to the appropriate subordinates. In the Advanced Extensible Interface (AXI) protocol, this use of an interconnect protocol now becomes the mode of operation, and the 'bus' itself is now a single point-to-point interface. Connection between managers and subordinates is now done via interconnect matrices that can connection multiple managers to different subordinates, with data transfer occurring in parallel. An individual matrix will have a finite set of ports for connecting managers, and a similarly finite set for connecting subordinates. If mode ports are required, other matric components can be connected as subordinates. The diagram blow shows a simplified arrangement of an AXI based system, with a second interconnect matrix component.



Another difference between AXI and its predecessors is that, instead of a single set of signals with outputs for sending commands and write data and inputs returning read data and responses, these functions are now split into five separates independent interfaces, or 'channels'. Three of these are for writes, with a write address channel, a write data channel, and a write response channel. The first two are instigated by the manager, and the last by the subordinate. The two remaining channels are for a read address channel, from the manager, and a read data channel, from the subordinate. The read response is combined with the read data channel as this is flowing in the same direction. The diagram below summarises this situation.



These channels, at the signal level, are truly independent but, of course, have to be consistent at the higher level. For example, a manager might send multiple write commands over the write address channel (if the subordinate can accept these) and then, at some future point, send the data for the outstanding writes. It (usually) has to send the write data in the order that was dictated by the address commands, but there is no timing coupling between the interfaces. (Out-of-order completions can be supported.) Indeed, if I read the specifications correctly, one might send all the data first (again, assuming the subordinate has space to accept this) and then the write commands. (I've not actually tried this, but I believe this is valid behaviour.)

What it does allow is (like for AHB) overlapping accesses, with addresses sent before the end of a previous burst, to allow full usage of the data bus bandwidth. We will be looking at bus signal timings shortly.

For the rest of this second part, we will concentrate on the AXI-3 specification, though we will be making reference to AXI-4 and even AXI-5. This is to keep the length of the document manageable, and the aim is to get familiar with AXI in general (hopefully to the point where implementing such an interface does not seem too daunting) and be able to discover more features in the newer specifications oneself if needed for a design's particular functionality.

Common Channel Characteristics

Each of the five channels has some common characteristics, which I will discuss here so that they can be taken as read when looking at the individual interfaces.

Like for AHB and APB, AXI has global clock and reset signals `ACLK` and `ARESETn`, with the reset being active low. These act in the same way as for the other interfaces.

Each of the channels have a handshaking method which, in my opinion, is simpler than for APB and AHB. All channels have a two-signal handshake, with a *x*`VALID` signal from the source, and an *x*`READY` signal from the destination. Only when both signals are high is a command or data word transferred. This makes inserting wait states from either and of the link possible without additional signalling (see `HTRANS`' `BUSY` state for AHB in part 1).

Write Channels

The three write channels (address, data, and response) as we shall see, have some common features from the protocols of the first part of the document.

Write Address Channel Signals

The write address channel signals are summarised in the table below. This has some colour coding to help identify the required versus optional signaling. Signals in black represent signals required at both manager and subordinate. The green signals are optional for a manager, but are required by a subordinate, but they may be tied off with the default values, as indicated in the table. Red signals are required by the manager but not by a subordinate and may be left unconnected. The rest of the signals in grey are optional for both manager and subordinate.

signal	bits	default	description
`AWID`	`ID_W_WIDTH`	0	ID tag for write transaction
`AWADDR`	`ADDR_WIDTH`	-	Address of first write transfer

AWLEN	8	0 (length of 1)	Length (number of transfers) - 1
AWSIZE	3	data bus width	Size (in bytes) of each transfer
AWBURST	2	INCR	Burst type
AWLOCK	2 (1 for AXI4)	normal access	Indicates a locked transaction
AWCACHE	4	4'b0000	Indicates caching (and other) properties
AWPROT	3	-	Protection attributes
AWQOS	4	4'b0000	AXI4. Quality of service ID
AWREGION	4	4'b0000	AXI4. Region indicator
AWUSER	USER_REQ_WIDTH	0	AXI4. User defined extension
AWVALID	1	-	Indicates write address signals valid
AWREADY	1	-	Returned ready signal from subordinate

Before we look at what these individual signals do, let's look at what minimal interfacing this channel requires. There must be three active signals—an address and the two handshake signals mentioned previously. The green signals can be tied off on a subordinate, and the red signal left unconnected on the manager. The default values for the green signals means that all transfers will be from a single manager and be a full single word transfer. This is the minimum functionality of the AHB, and I would argue, easier to implement with AXI handshaking. The belies the criticism of AXI complexity levelled by the engineer I mentioned in the introduction to the first part of the document.

All the signals for the AXI write channel are prefixed with `AW`, such as `AWADDR`. As this is the write address channel this is the only fully required signal (with handshaking implied and common to all interfaces). Like for AHB, it is configurable in width and gives the address of the *first* transfer. Since the data is transferred over its on channel, the address does not change for each data transfer and the subordinate must internally increment the address appropriately for a burst.

The `AWSIZE` signal is basically the same as for AHB, defining the number of bytes that will be in each individual data transfer, from 1 to 128. The length of a burst is defined with `AWLEN`, unlike AHB where this was undefined, or from a limited set of fixed sizes, depending on the transfer type. This length can be, for AXI-3, anything from 1 up to 16 words, though in AXI-4 incrementing transfers can be up to 256 words. In all cases, a burst must not be issued that would cross a 4Kbyte boundary (cf. 1Kbyte

restriction of AHB). Note that AWLEN is actually the transfer length minus 1, so that 0 equals a 1 word transfer up to 255 for a 256-word transfer.

The AWBURST signal is not quite the same as for HBURST of AHB. It does have an incrementing mode (INCR) and a wrapping mode (WRAP) with the AWLEN dictating the wrapping characteristics, just as for AHB. In addition, there is a FIXED mode, where each data transfer is to the same address. This might be useful for pushing onto a queue at some fixed location in the memory map.

AWPROT defined the access's protection mode and is similar to AHB's HPROT but drops a bit for distinguishing between opcode and data transfers. The AWLOCK signal is 2-bits for AXI-3 and specifies NORMAL, EXCLUSIVE, and LOCKED accesses. AXI-4 actually simplifies this signal to a single bit for just NORMAL and EXCLUSIVE. The AWCACHE signals define the accesses cache attributes. In AHB these were part of the HPROT signal and the AWCACHE has many of these attributes. It defines whether the access is bufferable, modifiable (i.e., attributes changed in transit), non-cached/cached access, cache read and/or write cache allocation and write-thru/write-back.

The AWID signal is an identification tag for the write transactions. It is optional for a manager and can be tied off for a subordinate to the default value. When used, it identifies which manager port was used for the transaction for routing back read data and responses, which will have a matching ID. For a single manager it is not required.

The other signals—AWQOS, AWREGION, and AWUSER—are introduced in AXI-4. The first is related to quality-of-service (i.e., minimum bandwidth/latency targets), the second indicates an address region identifier allowing a single interface to have multiple logical interfaces on a single physical interface, and the last which is a user defined signal. More details can be found in the AXI specifications (listed at the end of the document).

We shall look at timing in another section but, in general, all of the manager driven signals are all set in a single cycle, when AWVALID set. If AWREADY is high then the command is transferred, else the signals are held during the wait states until AWREADY does go high.

Write Data Channel Signals

The write data channel interface is the data counterpart to the write address channel and has the same handshaking method. All the write data channel signals are

prefixed with W, and the handshaking signals are thus WVALID and WREADY. The table below summarises the write channel signal, with the same colour coding as for the write address channel.

signal	bits	default	description
WID	ID_W_WIDTH	-	AXI3 only. ID tag of write transaction
WDATA	DATA_WIDTH	-	Write data
WSTRB	DATA_WIDTH/8	all ones	Write strobes for each byte lane
WLAST	1	-	Indicator of last data transfer
WUSER	USER_DATA_WIDTH	-	not AXI3. User extension attribute
WVALID	1	-	Indicates write data signals valid
WREADY	1	-	Returned ready signal from subordinate

As for the write address channel, only three signals need to be active for a valid interface, with the two handshaking signals and WDATA. The WLAST signal is required for a manager and is set on the last data transfer word of a burst (or the only transfer for a single word transfer). However, it need not be connected to a subordinate if there is no input. The WSTRB signal (cf. HWSTRB of AHB-5) and is the byte enables for the data bus, if wider that 8-bits. It is optional for a manager and can be tied off in a subordinate as all ones for all lanes always active.

The optional WID signal is a write identification tag, meant to identify the manager port that the data came from for data ordering purposes. This is defined for AXI-3 only and the specification suggests not to use it and use the AWID signal of the write address channel instead. Living with regret is just part of being an engineer. The WUSER signal was introduced in AXI-4 for user defined signalling and is optional.

Write Response Channel Signals

Since the write address and data channels are both from manager to subordinate a separate response channel is needed. The signals are defined in the table below, with the usual colour coding. All write response channel signals are prefixed with B (not sure why).

signal	bits	default	description
BID	ID_W_WIDTH	-	Identification tag for write response
BRESP	2	OKAY	Write response
BUSER	USER_RESP_WIDTH	all zeros	not AXI3.
BVALID	1	-	Indicates write response signals valid
BREADY	1	-	Returned ready signal from manager

The only absolutely required signals here are the two handshaking signals. The BRESP signal indicates the response status but, if a subordinate does not generate errors, this signal need not be implemented, and the default response is OKAY. The valid values for this two-bit signal are:

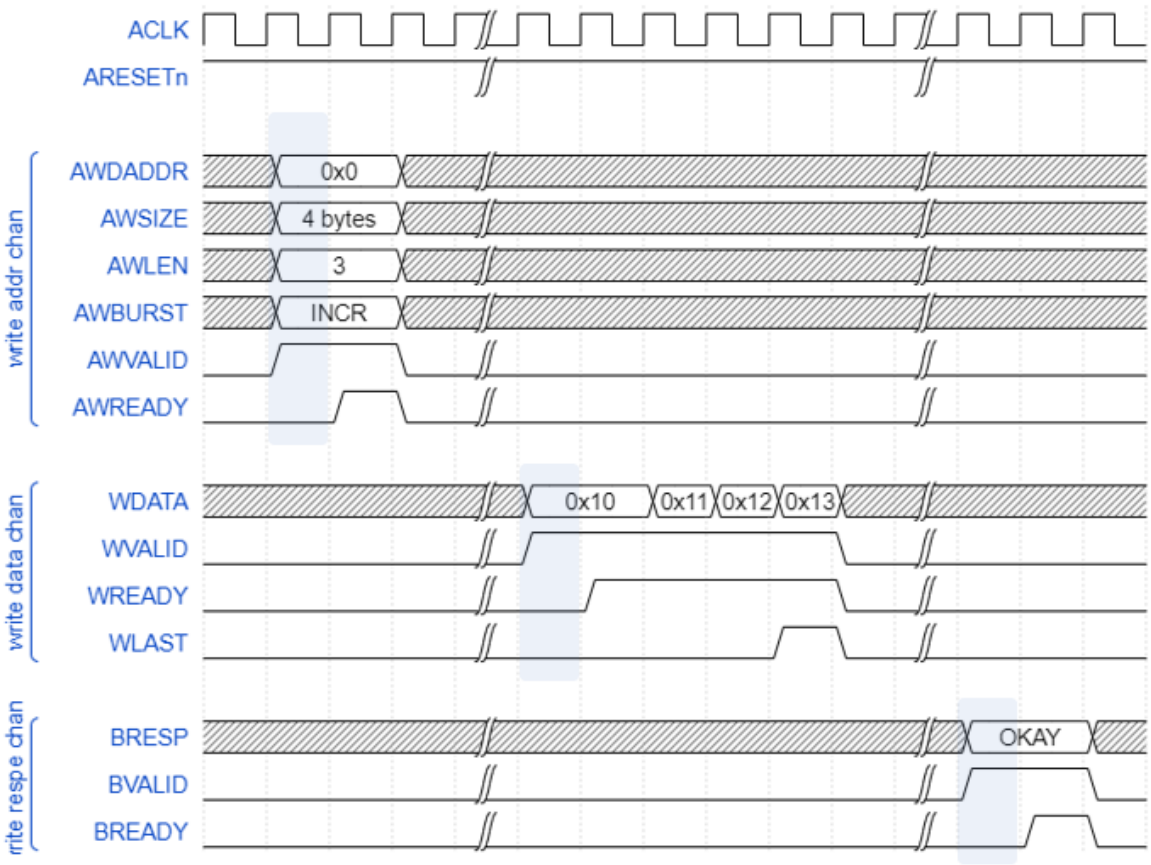
- 00b : OKAY (no error)
- 01b : EXOKAY (exclusive access with no error)
- 10b : SLVERR (unsuccessful transfer)
- 11b : DECERR (decoder error)

Note, the top bit of BRESP acts like PSLVERR of APB or HRESP of AHB, the bottom bit sub-divides this for exclusive access OK (cf. HEXOKAY of AHB-5) when the response is good, or a normal or decoder error when not good.

BID is a write identification that is required on a subordinate but is optional for a manager. When AWID is used on a write command, the BID will match the data returned for that ID. The BUSER signal, introduced in AXI-4, is completely optional and is for user defined signalling.

AXI Write Transaction Timing

Now we have defined all the signals for the write channels, let's look at what a write transaction looks like for the signal timing. The diagram below shows a write transfer for a 4-word incrementing burst transfer, where the data width is a byte.



The timing diagram shows the write address channel at the top. A write is issued for address `0x0`, for a burst of length 4 (remember `AWLEN` is length – 1), with `AWBURST` showing incrementing addresses. Since the `AWREADY` is low for the first cycle that `AWVALID` is high, a wait state is inserted, and the values are held for a cycle.

Some cycles later, on the write data channel, the data appears with the setting of `WVALID`, and 4 transfers take place. The first transfer had a wait state inserted by the subordinate with `WREADY` low. The subordinate's optional signal `WLAST` is low for all the transfers until the final one. Note that `WVALID` is shown active for the whole transfer, but it need not be. This may be deasserted at any point and then reasserted for the next transfer.

To complete the whole transfer, sometime after the write data is transferred, the write response channel is activated with `BVALID` asserted (driven by the subordinate), which waits for `BREADY` assertion (by the manager) and, if present, the response is given on `BRESP`.

For all the other signals not shown, whether optional, or for different specifications, the timings are just the same, being asserted when the appropriate *x*`VALID` signal is set.

Having looked at the write channels, we can turn our attention to the read channels. These have so much in common with the write channels that we can run through this in fairly short order.

Read Channels

The two read channels (address and data) are very similar to the write channels, with signalling combined for read data and response.

AXI Read Address Channel Signalling

The read address channel signals all have the prefix AR, but in every other respect they are the same as for the write address channel. The table below shows the signals with the usual colour coding:

signal	bits	default	description
ARID	ID_R_WIDTH	0	ID tag for read transaction
ARADDR	ADDR_WIDTH	-	Address of first read transfer
ARLEN	8	0 (length of 1)	Length (number of transfers) - 1
ARSIZE	3	data bus width	Size (in bytes) of each transfer
ARBURST	2	INCR	Burst type
ARLOCK	2 (1 for AXI4)	normal access	Indicates a locked transaction
ARCACHE	4	4'b0000	Indicates caching (and other) properties
ARPROT	3	-	Protection attributes
ARQOS	4	4'b0000	AXI4. Quality of service ID
ARREGION	4	4'b0000	AXI4. Region indicator
ARUSER	USER_REQ_WIDTH	0	AXI4. User defined extension
ARVALID	1	-	Indicates write address signals valid
ARREADY	1	-	Returned ready signal from subordinate

Suffice it to say we need not run through all these signals in detail and the section for the write address channel applies to these except for the signal name prefix.

AXI Read Data Channel Signalling

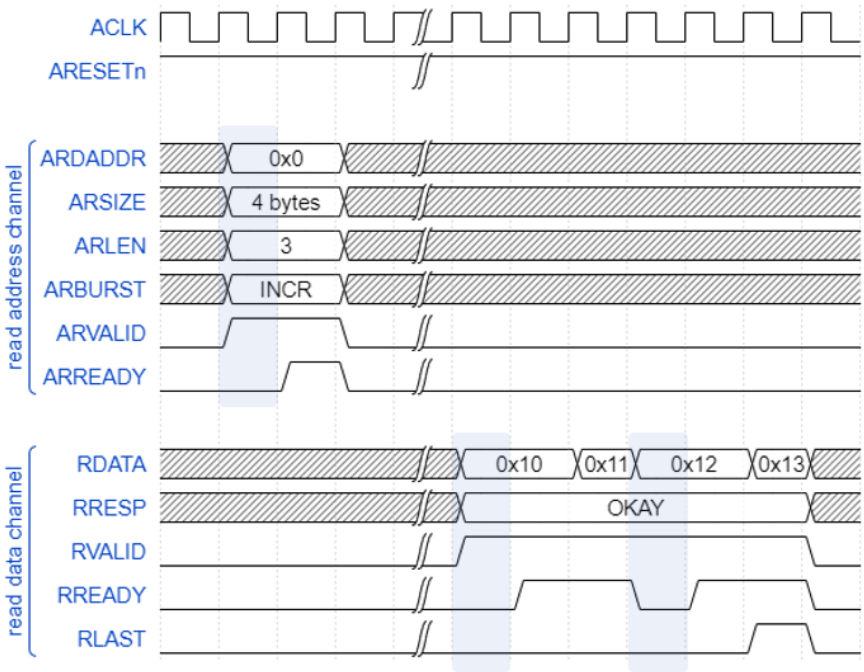
The read data channel combines both the data and response signalling, and all the signals are prefixed with R. The table below summarises these signals.

signal	bits	default	description
RID	ID_R_WIDTH	-	ID tag of read transaction
RDATA	DATA_WIDTH	-	Read data
RRESP	2	OKAY	Read response
RLAST	1	-	Indicator of last data transfer
RUSER	USER_DATA_WIDTH	-	not AXI3. User extension attribute
RVALID	1	-	Indicates read data signals valid
RREADY	1	-	Returned ready signal from manager

As ever, only the handshaking signals and the data signal are required at both ends. The RID (optional for the manager) is equivalent to the BID/WID signals and is AXI-3 only, and RLAST is required for the subordinate. The remaining signals are optional and have the same functionality as for the write data channel, except in the opposite direction (driven by the subordinate). So, we can now go straight into looking at the timings for a read transaction.

AXI Read Transaction Timing

The diagram below shows a write transfer for a 4-word incrementing burst transfer, where the data width is a byte.



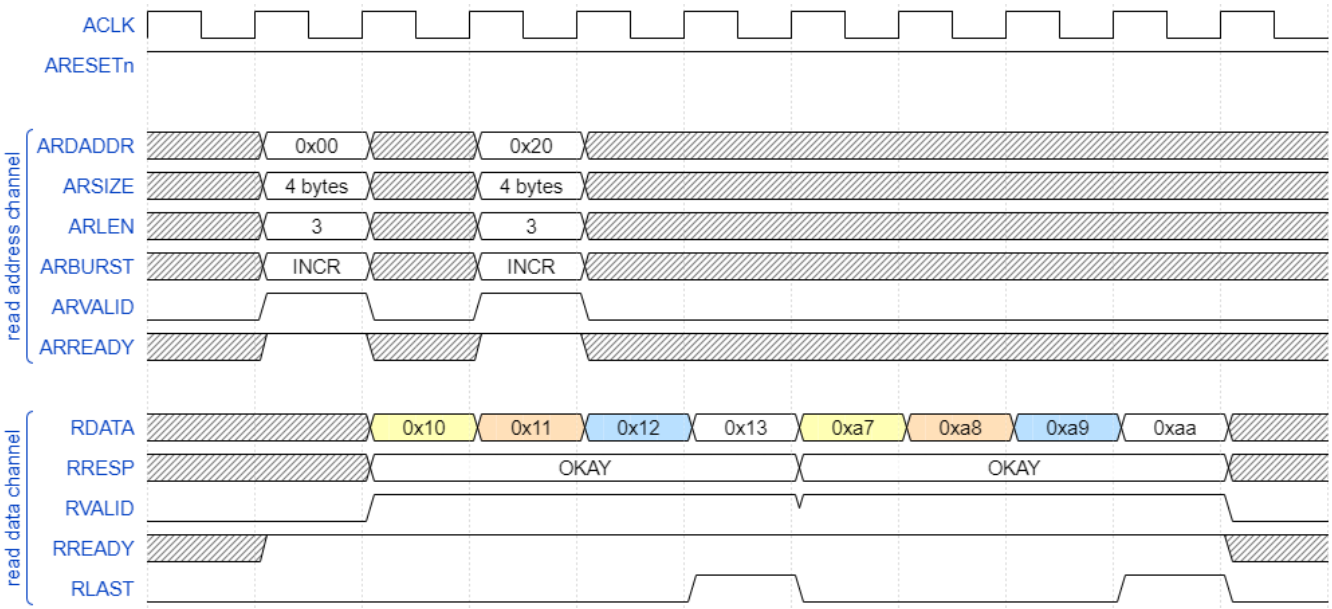
The timing diagram shows the read address channel at the top. A read is issued for address 0x0, for a burst of length 4, with ARBURST defining incrementing addresses. Since the ARREADY is low for the first cycle that ARVALID is high, a wait state is inserted, and the values are held for a cycle. Some cycles later the read data is returned on RDATA when RVALID is asserted. Two wait states are inserted (by the manager) during the data transfer. The read response is also returned on RRESP. Note that this is separately valid for each data transaction, so that while some values may have a response of OKAY, others may not. The optional RLAST is asserted only with the last data transaction.

The other signals not shown have the same timings as the control signals that are shown, if they are required.

AXI Multiple Outstanding Transactions

As mentioned before, transfers can overlap with new addresses valid before a transaction has completed, and data returned before a response for a previous transaction has been issued. This allows the bus to be able to run at 100% efficiency.

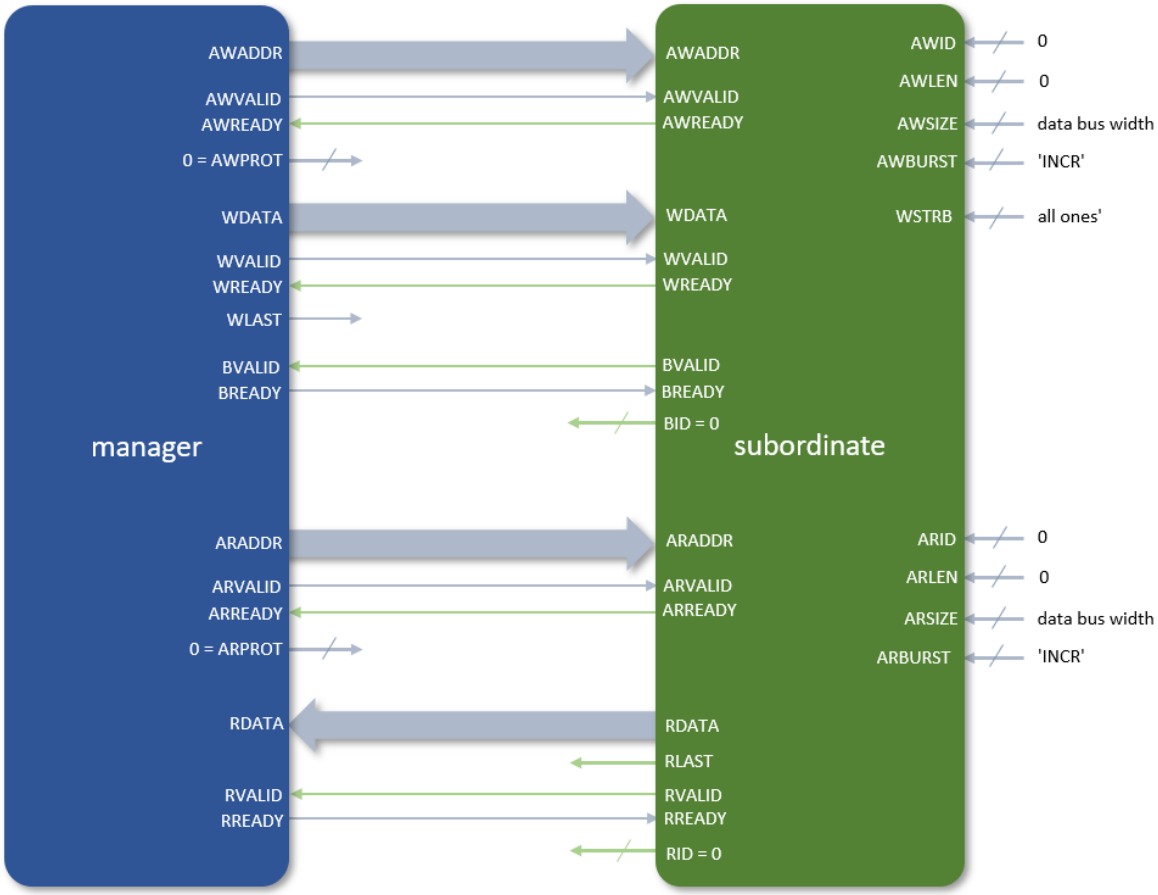
The diagram below shows two overlapping read transactions for four-word bursts



As can be seen, a read is issued, and data starts to return in the following cycles. Before the end of the burst, a new read address is issued for a 4-word burst. When the last data word is transferred for the first read, the data for the second read begins immediately.

Implementing AXI Interfaces

We have now been through all five channels of the AXI interface, with their timings. Hopefully, by now, you have seen that the protocol is not so complicated and has many advantages. We have seen that it has many advanced features to support such things as cache coherency and atomic operations, and also seen that many of these features are optional. The diagram below summarises a *minimum* manager subordinate interface connection:



For minimum compliance just the address and data busses (both read and write) are connected, each with the valid/ready handshake signals. The only additional connected signalling is the write response handshake signals. All other signals are only those required at one or the other end and can be left unattached or tied off to defaults, as shown. Even the manager `BREADY` signals could be tied high internally to the manager.

For the address and data channels, as a minimum, each end could be a short FIFO, with the driving end setting *x*`VALID` when 'not empty' and popping the value when *x*`READY` is also high. At the receiving end, *x*`READY` is simply set on 'not full', and data

is pushed into the queue when *x*VALID also high. This can be improved with registering outputs and early full or empty statuses, or not using queues for lower latency, but you get the idea. As this is the same for all interfaces, this logic need only be designed once and used for all address and data channels. Then, internally, any logic timing and implementation can be used as needed for the design, but one now can interface to an AXI port, either as a manager or a subordinate, and transfer data.

This only gives a minimalist interface for single word transfers but, from this base, additional functionality can be added as need for burst transfers, cache coherency, QoS, and whatever else is required.

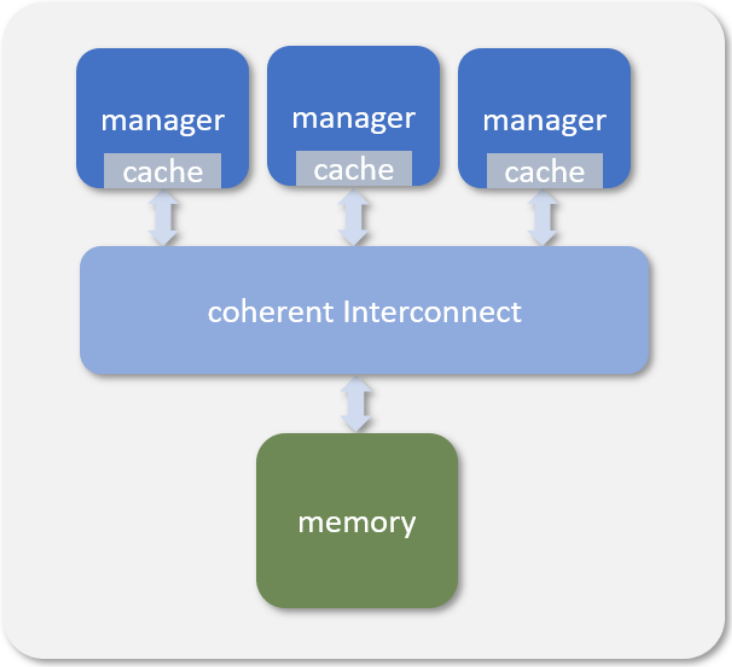
Other Protocols

Beyond AXI (ACE and CHI)

Back in the first part of the document, the AMBA busses were summarised in the first diagram and a couple of protocols were shown beyond AXI. These are the AXI Coherency Extensions (ACE) and the Coherency Hub Interface (CHI). As you can see from the names, cache coherency is front and centre of the protocol features. As systems have more and more processors, cache coherency is a genuine concern for efficient and accurate data sharing between cores (discussed in my article on caches).

The ACE protocol extends AXI and adds hardware-coherent cache support. That is, cached memory can coherently be shared across components without the need for software cache maintenance. This is done system wide, and regions can be specified for coherent memory. Each cache line now has five states associated with it, and what happens when that cached location is accessed is determined by that state. These states are UniqueClean, UniqueDirty, SharedClean, SharedDirty, and Invalid. For invalid and dirty/clean definitions, see my article on caches. The unique status is that a cache line is only valid in one cache, whereas shared implies it might be valid in multiple caches.

Cached data associated with a particular manager can be accessed from another manager, even when the first manager is active on the interconnect. The diagram below, shows multiple managers, each with their own cache, with a coherent interconnect, allowing (depending on cache line state) for one manager to access data in another's cache.



The ACE specification modifies the AXI address channels for cache access and barriers and extends signalling on BRESP. In addition, three new channels are added for 'snooping': snoop address, snoop response and snoop data. In this context, snooping is accessing a cache to see if an address is active within it and retrieving data from there.

The CHI protocol is trying to solve basically the same problems, but for systems that are far more distributed, with many more cores, in a scalable way. It has the same five states for cache lines as for ACE but moves away from busses and interconnects as such and moves to a message-based access model. It can be configured in multiple ways as a network: mesh, ring, crossbar etc. Indeed, the high-performance computer systems designed at Quadrics, when I was there, were just such message-based coherent systems using a crossbar network. Since then, on-chip systems with large numbers of processors have been developed, with all the same problems associated with them, and CHI aims to solve these in a similar manner.

The CHI protocol ensures coherency by allowing only one copy of data to exist for a location when a store occurs. Each manager can get a copy for its own local cache after the store. Within the interconnect, a 'home node' receives all the requests and manages and coordinates the snooping caching etc. The interconnect block may optionally have its own cache.

Intel Avalon

The Avalon bus specification from Intel (formerly Altera) aims to provide the same general functions as the AMBA specifications, with memory mapped interfaces, burst transfers, streaming interfaces and more. Much of the basic signalling for the memory mapped interface is similar to AXI, with notable differences being separate read and write strobes and burst lengths being from 0 to 4095 words and specified as *n* rather than *n*-1. Indeed, the interfaces are so similar that I have, in the past, constructed a converter from Avalon burst read interface (used internally to the design) to AXI manager interface (for interfacing to a memory controller) using just simple combinatorial logic.

The memory mapped interface has similar response errors and support for locked transaction. What is notably missing is any support for cache coherency, QoS, or protection. It does, though, specify an interrupt interface.

In addition to the memory mapped and interrupt interfaces, the Avalon specification defines two streaming interfaces; that is, queue-based data transfers. The first is signal flow controlled whilst the second is credit controlled (cf. data link layer of PCIe). A last 'conduit' interface is defined, but this is really a means to bundle an arbitrary set of user defined signals into a single unit—useful when using Intel's platform designer to construct logic block hierarchy and auto-generate the interconnect logic.

Wishbone

I want to mention the Wishbone bus, which is an open-source specification with much the same aims as the other specifications. It can be configured for a shared bus (like AHB), a pipeline or a crossbar switch system. The signalling is very limited compared to the other specifications and mainly associated with data transfer. There is no higher-level signalling for cache coherency, QoS, locked transfers etc.

Its main advantage is that is a free open-source specifications. My understanding, though, is that AMBA has no licence or royalty associated with using the specifications (unlike the ARM processor architectures), so this advantage is limited.

Conclusions

We have, over the two parts of the document, looked at some of the AMBA specifications as case studies for the bus and interconnect typically used in SoC embedded systems, starting with a simple word transfer protocol (APB), through a

higher performance burst bus (AHB), to an interconnect base protocol, AXI. All of these have a common heritage, but new features are added (often optionally) to solve particular problems of efficient data transfer within a system. We have seen that cache coherency solutions start to dominate, with ACE and CHI specifications beyond AXI, in order to meet the needs of large multi-core distributed systems, using techniques that were the mainstay only of supercomputers not so long ago.

AMBA protocols are by far the most commonly used (in my experience), but other protocols such as Avalon are used within their own domain, and open-source specifications exist for freedom from any tie-in to corporate control.

The aim of this document has been to show how, at the basic level, these interfaces are not complicated to understand or implement, despite the perception of the engineer mentioned in the introduction to the first part of the document. The layering of features, we have shown, allows, at its simplest, a straight-forward implementation of an AXI interface (either manager or subordinate) using just well understood components, such as FIFOs. So, do you still think AXI is too complicated a protocol to use?

Getting Hold of the Specifications

For those wishing to dive deeper into the protocols, the list below gives links to all the specifications mentioned in this document.

- APB
- AHB
- AXI/ACE
- CHI
- Avalon
- Wishbone