CHAPTER

On-Chip Communication Architecture Standards

3

System-on-chip (SoC) designs typically have several different types of components such as processors, memories, custom hardware, peripherals, and external interface IP (intellectual property) blocks that need to communicate with each other. In SoC design houses, some of these components might be designed from scratch, while others are reused from previous designs or procured from external IP vendors. Each of these components has an interface to the outside world consisting of a set of pins that are responsible for sending/receiving addresses, data, and control information to/from other components. The choice of pins at the interface is governed by the particular bus protocol of the communication architecture. In order to seamlessly integrate all these components into an SoC design, it is necessary to have some kind of a standard interface definition for the components. Without a standard interface definition, the component interfaces will not be compatible with the bus architecture implementation, and consequently will not function correctly. In such a scenario, the components will require the design of logic wrappers at their interfaces (more details in Chapter 9) to correctly interface with the bus architecture being used. These logic wrappers, however, require additional area on the chip and can be time consuming to design and verify.

To speed up SoC integration and promote IP reuse over several designs, several bus-based communication architecture standards have emerged over the past several years. A communication architecture standard defines a specific data transfer protocol, which in turn decides the number and functionality of the pins at the interface of the components. Usually, bus-based communication architecture standards define the interface between components and the bus architecture, as well as the bus architecture that implements the data transfer protocol. Many of the bus architecture standards give designers a certain amount of freedom to implement the bus architecture in one of many ways. For instance, most bus-based communication architecture standards give designers freedom to select arbitration policies that are suitable for the specific SoC being designed. Socket-based bus interface standards on the other hand give absolute freedom to a designer, with respect to the choice and implementation of the bus architecture, since they only provide definitions for designing component interfaces. Any proprietary or standard bus architecture implementation can be selected, and components with socket-based interface definitions can be connected to it using logic adapters. This allows a great deal of flexibility for IP reuse. In this chapter, we present various standards used in SoCs with bus-based communication architectures. Section 3.1 details some commonly used bus-based communication architecture standards. Section 3.2 describes popular socket-based bus interface standards. Finally, we briefly discuss some of the standards for off-chip interconnects in Section 3.3.

3.1 STANDARD ON-CHIP BUS-BASED COMMUNICATION ARCHITECTURES

Since the early 1990s, several on-chip bus-based communication architecture standards have been proposed to handle the communication needs of emerging SoC designs. Some of the popular standards include ARM Microcontroller Bus Architecture (AMBA) versions 2.0 [1] and 3.0 [2], IBM CoreConnect [3], STMicroelectronics STBus [4], Sonics SMART Interconnect [5], OpenCores Wishbone [6], and Altera Avalon [7]. The next few sections describe these bus-based communication architecture standards in more detail. Since these standards are constantly evolving, the descriptions of these standards presented here are meant to serve as exemplars, to highlight the capabilities and features required from on-chip communication architectures for supporting diverse SoC application requirements.

3.1.1 AMBA 2.0

AMBA version 2.0 [1] is one of the most widely used on-chip communication standards today. The goal of this standard is to provide a flexible high performance bus architecture specification, that is technology independent, takes up minimal silicon area, and encourages IP reuse across designs. AMBA 2.0 defines three distinct bus standards:

1. *Advanced high performance bus (AHB)*, which is a high performance bus meant to connect high bandwidth, high clock frequency components such as microprocessors, DMA (Direct Memory Access) controllers, off-chip memory interfaces, and high bandwidth on-chip memory blocks.
2. *Advanced system bus (ASB)*, which is a light-weight alternative to the AHB bus, meant to connect high clock frequency components that do not need the advanced protocol features of AHB.
3. *Advanced peripheral bus (APB)*, is a low complexity bus optimized for low power operation, and meant for high latency, low bandwidth peripheral components such as timers, UARTs (universal asynchronous receivers/transmitters), user interface (e.g., keyboard) controllers, etc.

Figure 3.1 shows an example of a typical AMBA-based system, with the buses arranged in a simple hierarchical bus topology (see Chapter 2). The AMBA AHB (or ASB) bus typically acts as a backbone bus that provides a high bandwidth interface between the components involved in a majority of the transfers. The bridge component on the high performance bus is used to interface to the lower bandwidth APB bus, to which most of the low bandwidth peripherals are

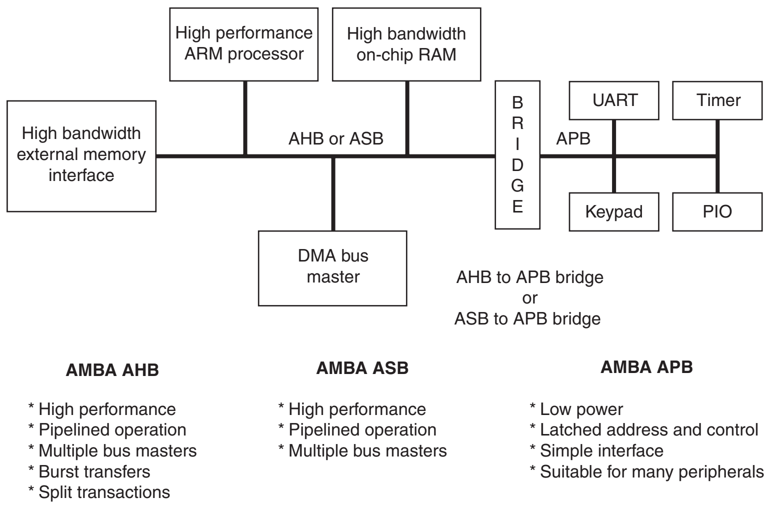

FIGURE 3.1

A typical AMBA 2.0 system [1]
Source: ARM Inc.

connected. Note that Fig. 3.1 shows just one of the topology arrangements possible with the AMBA 2.0 buses. Other topology configurations such as single shared bus, multi-layer AHB (or bus matrix) are also possible and will be discussed later in this section. Also note that the AMBA specification [1] is described at the architectural level granularity, and does not provide any information about the electrical characteristics of the bus, which are dependent on the selected manufacturing process technology. Even the timing specifications are only provided in terms of signal behavior at the cycle level—the exact timing requirements depend on the selected complementary metal-oxide semiconductor (CMOS) process technology and frequency of operation. The AMBA specification [1] defines all the signals, transfer modes, structural configuration, and other bus protocol details for the AHB, ASB, and APB buses. Since the specification recommends using the AHB over the ASB for all new designs, and the features of the ASB are simply a subset of the AHB features, we will not present details of the ASB bus. We now describe the specifications of the AHB and APB buses.

3.1.1.1 *Advanced High Performance Bus*

The AHB bus standard describes a high performance bus that supports advanced features for high bandwidth, low latency data transfers. AHB can be used to connect multiple master components, and supports high performance data transfer features such as pipelined operations, burst mode transfers, and split transactions. A single 32-bit address bus is used by the masters to transmit the addresses of the slaves required to complete the read or write data transfer requests. Since the AMBA specification restricts all its on-chip buses to a non-tri-state implementation, the AHB has separate data buses for reads and writes. The data buses have a minimum recommended width of 32 bits, but can have any values ranging through 8, 16, 32, 64, 128, 256, 512, or 1024 bits, depending on application bandwidth requirements,

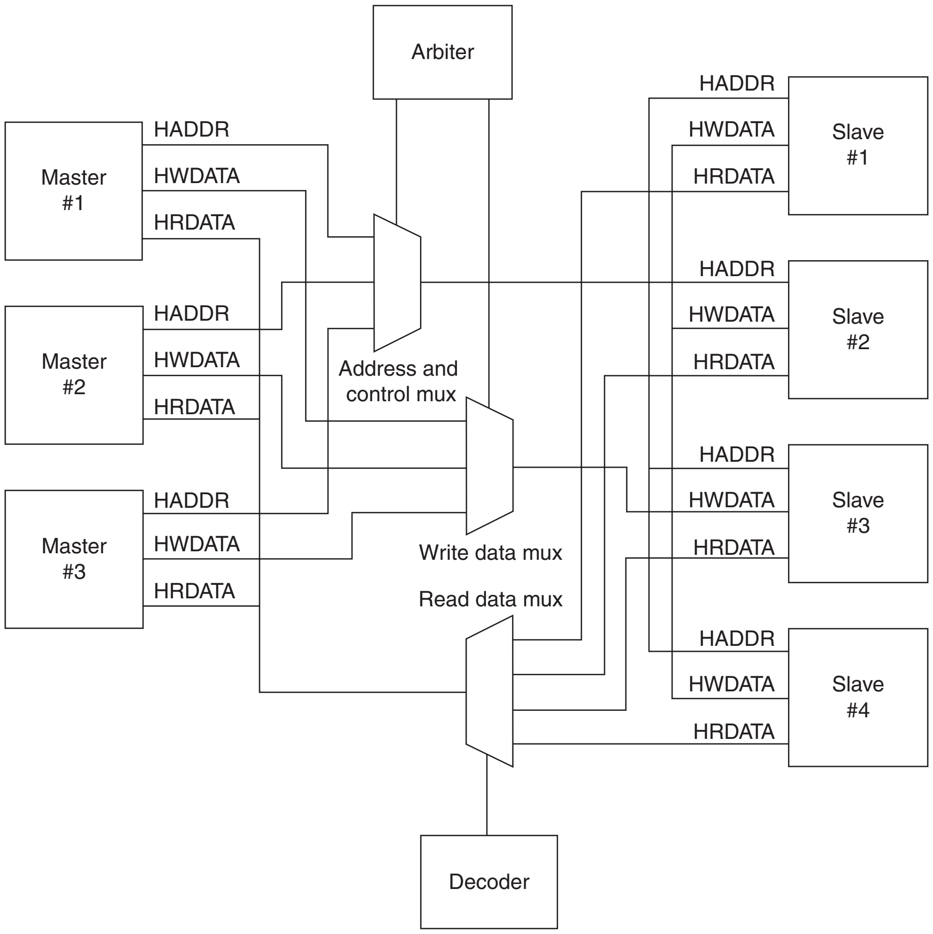

FIGURE 3.2

AHB multiplexer interconnection scheme [1]
Source: ARM Inc.

component interface pin constraints, and the bit width of words accessed from memory modules (e.g., embedded DRAM). Figure 3.2 shows the structure of a simple AHB shared bus with 3 masters and 4 slaves. As can be seen, the AHB makes use of a central multiplexer-based interconnection scheme, along with centralized arbiter and decoder modules to manage data transfers. The masters on the bus drive their address signals (*HADDR*) and control signals (not shown in the figure) whenever they want to perform a data transfer, and the arbiter determines which of the masters will have its address, control, and possibly write data (or *HWDATA*) signals routed (broadcast) to all the slaves on the bus. The decoder is used to decode the destination slave address, select the appropriate slave to receive the data transfer request, and route back response (and possible read data or *HRDATA*) signals to the masters.

Figure 3.3 shows a basic data transfer on the AHB bus. An AHB transfer consists of an address phase that lasts for a single cycle, and a data phase, that can require

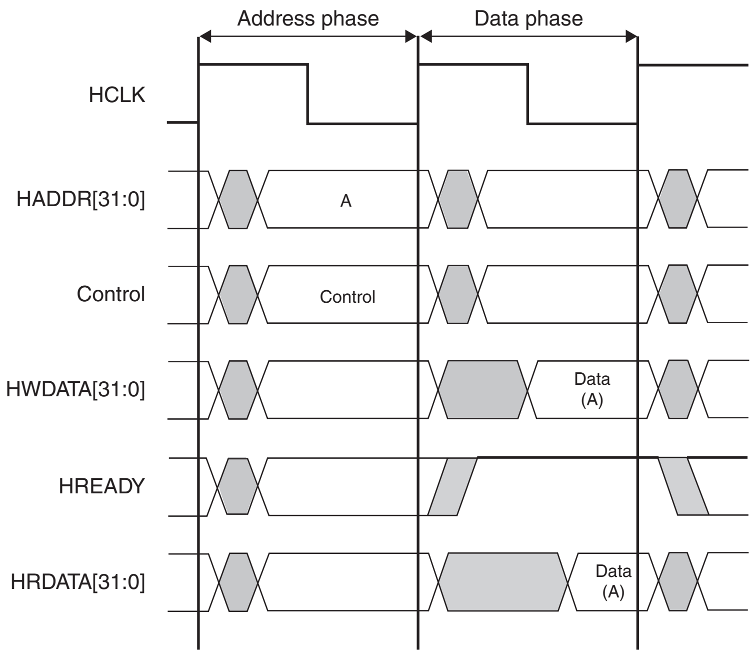

FIGURE 3.3

Basic data transfer on AHB bus [1]
Source: ARM Inc.

one or more cycles. In the figure, the master drives the address and control signals on the bus after the rising edge of the clock (*HCLK*) in the address phase. The slave samples (i.e., reads) the address and control signals at the next rising edge of the clock, which marks the beginning of the data phase. In the case of a read, the slave drives the appropriate data onto the read data bus (*HRDATA*) in this data phase cycle, followed by the master sampling the data off the bus on the third rising edge of the clock. Otherwise, in the case of a write, the master drives the data onto the write bus (*HWDATA*) in the data phase, followed by the slave sampling the data off the bus on the third rising edge of the clock. Note that Fig. 3.3 shows data on both the read and write buses for illustration purposes only (corresponding to a read or a write, respectively). In practice, the read data bus will be idle on a write, and the write data bus will be idle during a read.

It is possible that the slave can require more than one cycle to provide a response to the master, either because it needs one or more cycles to read the requested data, or to get in a state to write data. In such cases, the slave can introduce wait cycles by lowering the *HREADY* signal to indicate to the master that additional time is required to complete the transaction. Figure 3.4 shows the case for a read (or write) operation, where the slave inserts two wait cycles by lowering the *HREADY* signal for two cycles, before driving the read data onto the read data bus (or sampling the write data from the write data bus). Since the address and data phases of a transfer occur in separate cycles, it is possible to overlap the address phase of one transfer with the data phase of another transfer, in the same clock period. Such

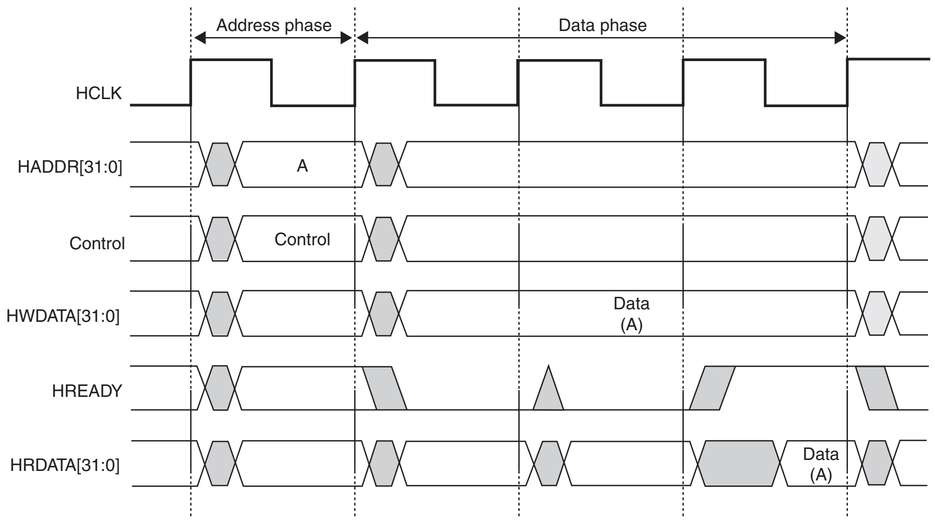

FIGURE 3.4

Basic data transfer on AHB bus with slave wait states [1]
Source: ARM Inc.

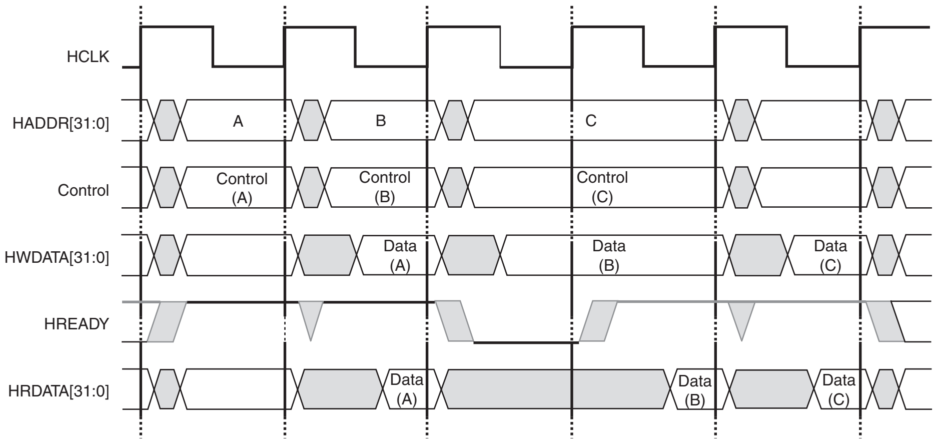

FIGURE 3.5

Pipelined data transfers on AHB bus [1]
Source: ARM Inc.

a *pipelined* operation of the AHB bus, depicted in Fig. 3.5, allows for high performance operation.

In addition to the address and data phases, an arbitration phase is necessary for all data transfers if there is more than one master connected to a bus. The arbitration phase ensures that only one master gains access to the bus and proceeds with its data transfer at any given point of time. Figure 3.6 shows the centralized arbitration scheme on an AHB bus to which three master components are connected. Whenever a master needs to initiate a read or write data transfer,

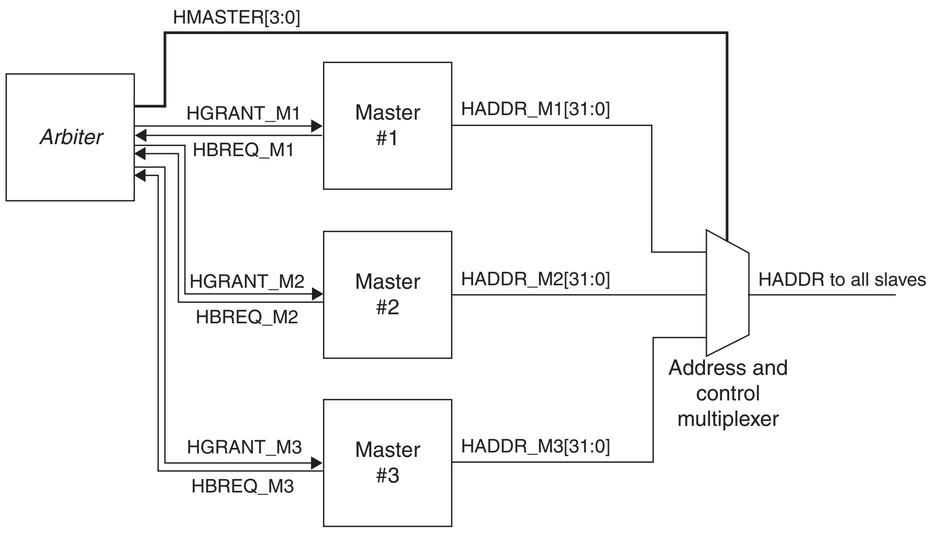

FIGURE 3.6

Arbitration on AHB bus [1]
Source: ARM Inc.

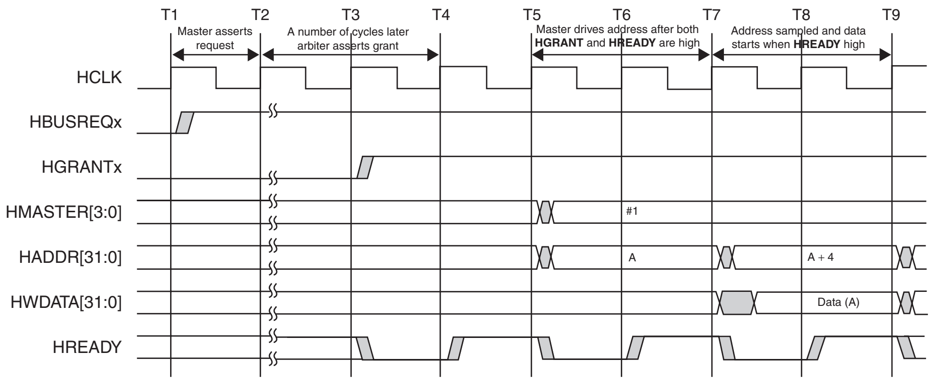
T1
T2
T3
T4
T5
T6
T7
T8
T9
Master asserts request
A number of cycles later arbiter asserts grant
Master drives address after both HGRANT and HREADY are high
Address sampled and data starts when HREADY high
HCLK
HBUSREQx
HGRANTx
HMASTER[3:0]
HADDR[31:0]
HWDATA[31:0]
HREADY
#1
A
A + 4
Data (A)

FIGURE 3.7

Cost of arbitration on AHB bus [1]
Source: ARM Inc.

it drives the *HBUSREQ* signal to the arbiter, requesting it for access to the AHB bus. The arbiter samples the *HBUSREQ* signals from all the masters and uses its arbitration policy to decide which master gets granted access to the bus access. The AHB specification does not identify a particular arbitration scheme—instead, a designer is given the freedom to implement any suitable scheme depending on the target application latency and bandwidth requirements. Once the arbitration scheme selects a master, the arbiter drives the *HGRANT* signal to the selected master, indicating it has been granted access to the bus and can proceed with its transaction. All the other masters must wait until re-arbitration, after completion of the current data transfer. Figure 3.7 shows an example of how the arbitration

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

FIGURE 3.8

Different burst modes on the AHB bus [1]
Source: ARM Inc.

proceeds on an AHB bus: the master asserts the *HBUSREQ* signal to request bus access from the arbiter, which samples the requests at the next rising clock edge, uses its internal arbitration scheme to select a master, and then grants it access to the bus by asserting the *HGRANT* signal. Note that there is typically at least a one cycle overhead for arbitration that can increase to several cycles in the case of a more complex arbitration scheme, or when there are a large number of masters connected to the bus. Such a large overhead for a single data transfer can limit the performance on the bus. To alleviate this overhead, the AHB bus supports burst data transfers that only need to arbitrate once (at the beginning of the transaction) for transferring multiple data items.

Figure 3.8 shows the different burst modes allowed for the AHB bus. The master uses the *HBURST* signal to indicate the size of a burst data transfer. *Incrementing bursts* (INCR, INCR4, INCR8, and INCR16) access sequential locations, and the address of each transfer in the burst is simply an increment of the previous address. *Wrapping bursts* are similar to incrementing bursts, but if the start address of the data transfer is not aligned to the total number of bytes in the burst, then the address of the transfers in the burst will wrap when the boundary is reached. For instance, a wrapping burst of length 4, transferring word-sized (4 byte) data items will wrap at 16 byte boundaries. So if the start address of a transfer is 0x64 h, then the four addresses in the burst will be 0x64 h, 0x68 h, 0x6Ch, and 0x60 h. Figure 3.9 presents an example of a wrapping burst of length 4 (WRAP4). Notice the wait state inserted by the slave (by lowering the *HREADY* signal) for the first data transfer. Since the burst of word transfers will wrap at 16 byte boundaries, the transfer to address 0x3C is followed by a transfer to address 0x30 h. An incrementing burst of length 4 (INCR4) on the other hand would have continued beyond the 16 byte boundary, and would access the following sequence of addresses: 0x38 h, 0x3Ch, 0x40 h, and 0x44 h.

It is possible for an AHB burst to be interrupted by the arbiter, in case a higher priority master needs to transfer data on the bus. If a master loses access to the bus in the middle of a burst, it must reassert its *HBUSREQ* signal to again arbitrate

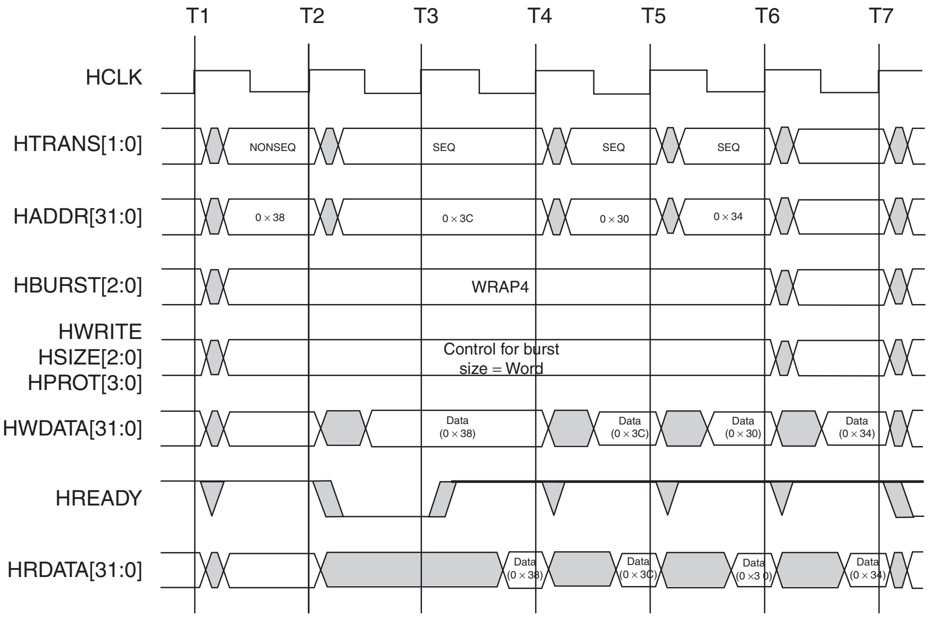

FIGURE 3.9

Example of a burst transfer—a wrapping burst of length 4 on the AHB bus [1]
Source: ARM Inc.

for access to the bus. Master and slave module AHB interfaces must be designed to correctly handle early burst termination. If a master requires that its burst transfer not be interrupted, it must assert the *HLOCK* signal when requesting the bus from the arbiter, to indicate that the bus needs to be *locked* away from other masters, for the duration of the burst transfer. When the arbiter sees the *HLOCK* signal for a master request, it must ensure that no other master is granted access to the bus once this master is granted access to the bus and its first data transfer has commenced. In addition to *HLOCK*, there are several other control signals used during data transfers, such as:

- *HWRITE*: A 1-bit signal generated by the master that indicates the transfer direction—a write when it is high, or a read when it is low.
- *HSIZE*: A 3-bit signal generated by the master that indicates the size of the data transfer. There are 8 possible allowed values for data transfer size ranging from 8 bits (000) to 1024 bits (111). This signal is particularly useful if the size of data being transmitted on the bus is smaller than the bus width.
- *HTRANS*: A 2-bit signal generated by a master to indicate the type of a transaction. There are four types of transfers possible in AHB: (i) *NONSEQ*, which is usually a single transfer or the first transfer in a burst, (ii) *SEQ*, which specifies the remaining transfers in a burst, (iii) *IDLE*, which indicates that no data transfer is required and is generally used when a master is granted access to the bus, but does not need to transfer any data, and (iv) *BUSY*, which indicates that the master is continuing with a burst transfer, but the

next transfer cannot take place immediately (e.g., when the master needs to process a read data for multiple cycles before being ready to receive the next data item in the burst; or when a write data in a burst takes multiple cycles to be generated).

- *HRESP*: A 2-bit signal generated by the slave that specifies the status of a data transfer. If the data transfer completes successfully, an OKAY response is returned on these signals. Otherwise, if an error occurs (e.g., an attempt to write to a ROM region), an ERROR response is returned. The master can choose to continue a burst transfer if an error occurs in the middle of the burst and rectify the error afterward, or it may decide to cancel the remaining transfers in the burst and handle the error immediately. The SPLIT and RETRY responses are used by slaves to free up the AHB bus when they are unable to provide the requested data immediately. The difference between these two responses is that when an arbiter sees a RETRY response, it will continue to use the normal priority scheme and grant access to the bus to higher priority masters, as usual; whereas on a SPLIT response, the arbiter adjusts the priority scheme and allows any other requesting master (even one with a lower priority) to gain access to the bus. To complete a SPLIT transfer, the slave must inform the arbiter when it has the data available.
- *HPROT*: A 4-bit protection signal that provides additional information about a data transfer. It is typically used by a component requiring some level of protection control, and can be used to specify, for instance, if the transfer is an opcode (*OPC*) fetch or a data access. Other possible uses include indications for a privileged mode access or a user mode access, and specifying if a transfer is bufferable or cacheable.

As mentioned above, a SPLIT transfer enables a slave to free up access to a bus, if it believes that the data requested from it (by the master) will not be available for several cycles. In such a scenario, the slave records the ID of the master from which the request initiated (to restart the transfer again at a later time) and asserts an *HSPLIT* signal to the arbiter. The arbiter then masks (i.e., ignores) requests from the master that was SPLIT, and grants other masters access to the bus. This process is shown in Fig. 3.10, where a SPLIT on the second cycle causes the arbiter to grant bus access to another master. When the slave is ready to complete the transfer, it signals the arbiter and sends the ID of the master that was involved in the split transfer. The arbiter unmasks the master, and eventually grants the master access to the bus, to complete the data transfer. The entire process is transparent to the masters making the request. Thus SPLIT transfers allow the time that would have otherwise been spent waiting for the data from the slave, to be utilized in completing another transfer, which enables better utilization of the bus. Note that an AHB master can only have a single outstanding transaction at any given time. If more than one outstanding transaction needs to be handled by a master component, it requires an additional set of request and grant signals for each such outstanding transaction. Both SPLIT and RETRY transfers can cause bus deadlocks, and therefore care must be taken while implementing them.

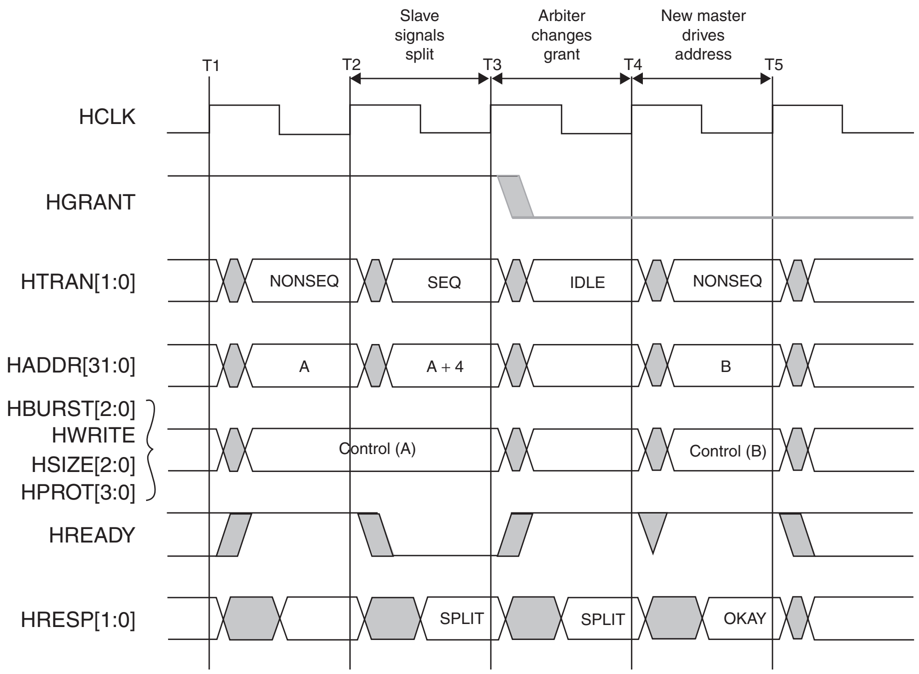

FIGURE 3.10

SPLIT transfer on the AHB bus [1]
Source: ARM Inc.

AHB Bus Matrix Topology

In addition to the basic hierarchical bus topology, where an AHB bus uses a bridge to interface with an APB bus, AHB bus-based communication architectures can have other topologies as well, such as a hierarchical bus topology with multiple AHB (and APB) buses interfacing with each other through bridges. For SoC designs that require very high bandwidths and require multiple concurrent data transfers, the hierarchical bus architecture may be insufficient. For such designs, an AHB multi-layer bus matrix [8] topology offers a more suitable communication infrastructure. Figure 3.11(a) shows an example of a 2 master, 4 slave AHB full bus matrix topology, that has multiple buses in parallel, to support concurrent data transfers and high bandwidths. The *Input Stage* is used to handle interrupted bursts, and to register and hold incoming transfers from masters if the destination slaves cannot accept them immediately. The *Decoder* generates select signals for slaves, and also selects which control and read data inputs received from slaves are to be sent to the master. The *Output Stage* selects the address, control and write data to send to a slave. It calls the *Arbiter* component, which uses an arbitration scheme to select the master that gets to access a slave, if there are simultaneous requests from several masters. Unlike in traditional hierarchical shared bus architectures, arbitration in a bus matrix is not centralized, but distributed so that every slave has its own arbitration. One drawback of the full bus matrix scheme is

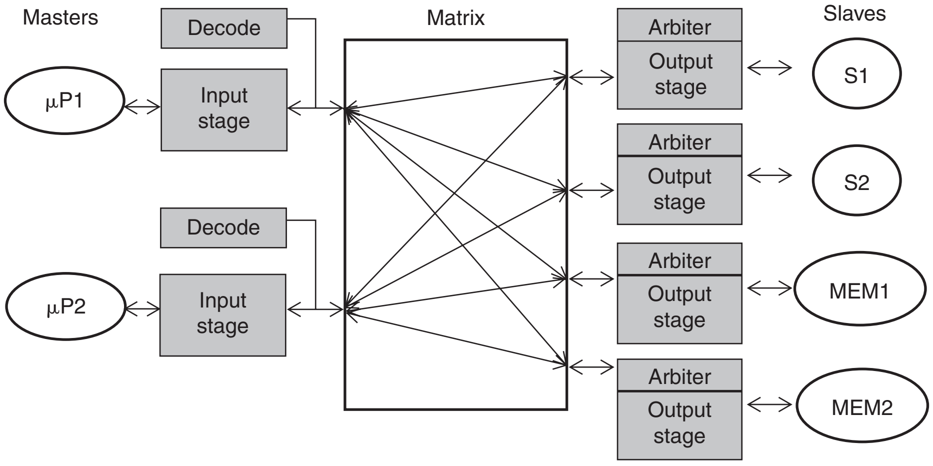

FIGURE 3.11(a)

An example of a 2 master, 4 slave AHB: full bus matrix topology

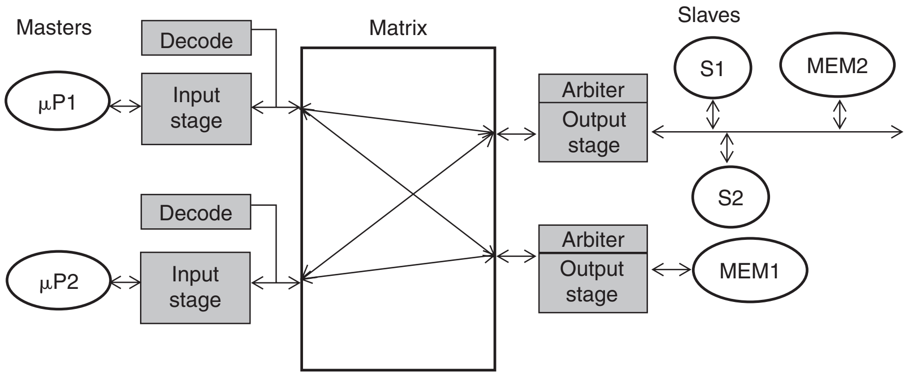

FIGURE 3.11(b)

Partial bus matrix topology

that it connects every master to every slave in the system, resulting in a very large number of buses (and consequently wires and bus logic components). Such a configuration, therefore, achieves high performance at the cost of high power consumption and a larger area footprint. For systems that have less stringent performance requirements, a partial AHB bus matrix [8, 9] topology can be used. Figure 3.11(b) shows a partial AHB bus matrix configuration that clusters components onto shared buses, to reduce the number of buses in the matrix. This partial matrix configuration offers less potential bandwidth due to the likelihood of data traffic conflicts on the shared buses, when compared to the full bus matrix. However, the partial matrix configuration consumes less power and takes up a smaller chip area, which is a desirable characteristic for communication architectures. Pasricha et al. [10] showed how a full AHB bus matrix can be reduced to a

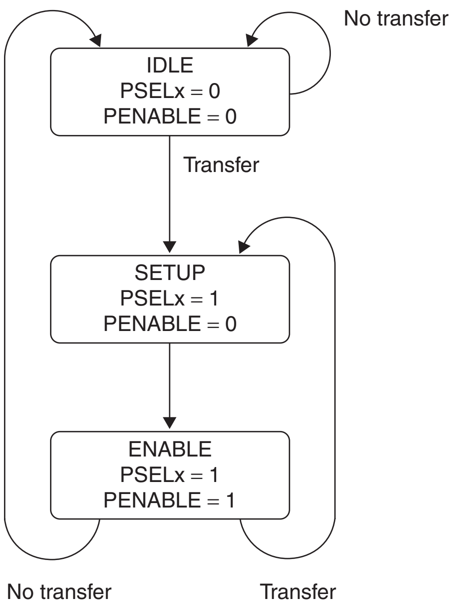

FIGURE 3.12

State diagram representing activity of the APB bus [1]
Source: ARM Inc.

partial bus matrix topology for a multiprocessor system-on-chip (MPSoC) design, by optimally reducing the number of buses while still meeting all application performance constraints. This methodology is described in more detail in Chapter 6.

3.1.1.2 *Advanced Peripheral Bus*

The APB bus standard defines a bus that is optimized for reduced interface complexity and low power consumption. This bus is meant to interface with the AHB (via a bridge), connecting low bandwidth (or high latency) peripheral components that do not require the advanced features of high performance buses such as the AHB. The APB allows only non-pipelined data transfers, and has only a single master—the bridge that connects the AHB bus to the APB bus. Typically, the APB bus operates at a much lower clock frequency (which helps reduce power consumption) and has a smaller bus width, compared to the AHB bus. Figure 3.12 depicts a state diagram that represents the activity on an APB bus. The *IDLE* state is the default state, in which the APB bus remains when there are no transfer requests from the AHB bus. When a transfer request arrives from the AHB bus via the AHB–APB bridge, the APB bus moves to the *SETUP* state and asserts the appropriate slave select signal to select the slave on the APB bus that is required to participate in the transfer. The APB bus remains in the *SETUP* state for one cycle, and this time is spent in decoding the address of the destination peripheral component. The APB bus moves to the *ENABLE* state on the next rising edge of the clock and asserts the *PENABLE* signal to indicate that the transfer is ready to be performed. This state also typically lasts for one cycle, after which it can go back

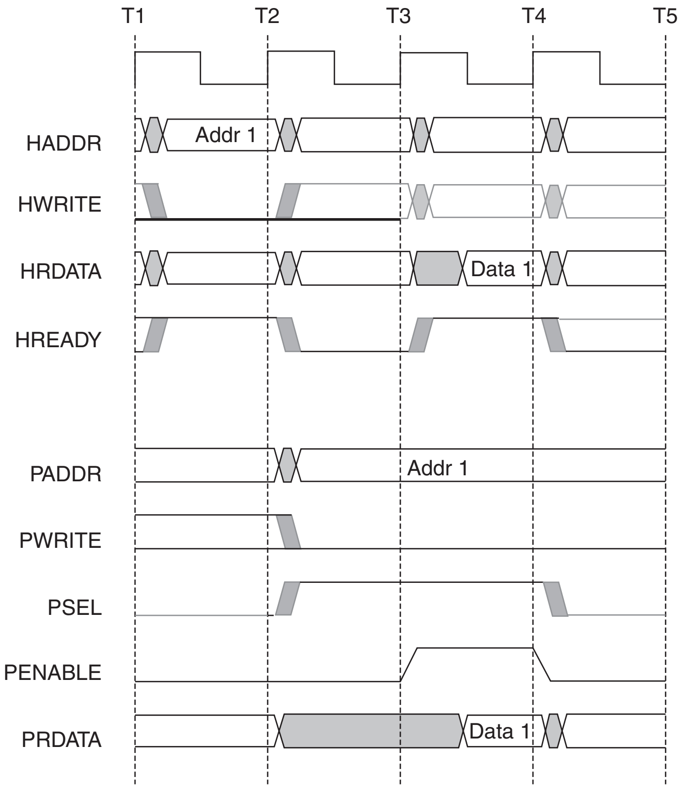

FIGURE 3.13

Read data request from the AHB to the APB bus [1]
Source: ARM Inc.

to the *SETUP* stage if another transfer follows the current transfer or to the *IDLE* state if no further transfers are required.

Figure 3.13 illustrates a read request as it propagates from the AHB to the APB bus. The top four signals belong to the AHB bus while the four signals at the bottom belong to the APB bus. The master component on the AHB bus drives the address and control signals onto the AHB bus in the first cycle. These are sampled by the AHB–APB bridge component on the rising edge of the next clock cycle (T2). Since the transfer is intended for the APB bus, the appropriate select signal (*PSEL*) is asserted and the APB bus transitions from the *IDLE* state to the *SETUP* state in cycle T2. This is followed by the *ENABLE* state in cycle T3, in which the slave receives the request and returns the read data. The returned data can usually be directly routed back to the master on the AHB bus, where it will be sampled off the bus at the next rising clock edge (T4). However, for high performance systems, the returned data can be first registered at the bridge, and then driven to the appropriate master in the following cycle. While this approach requires an extra cycle, it can allow the AHB bus to operate at a much higher clock frequency, which allows an overall improvement in system performance. Figure 3.14

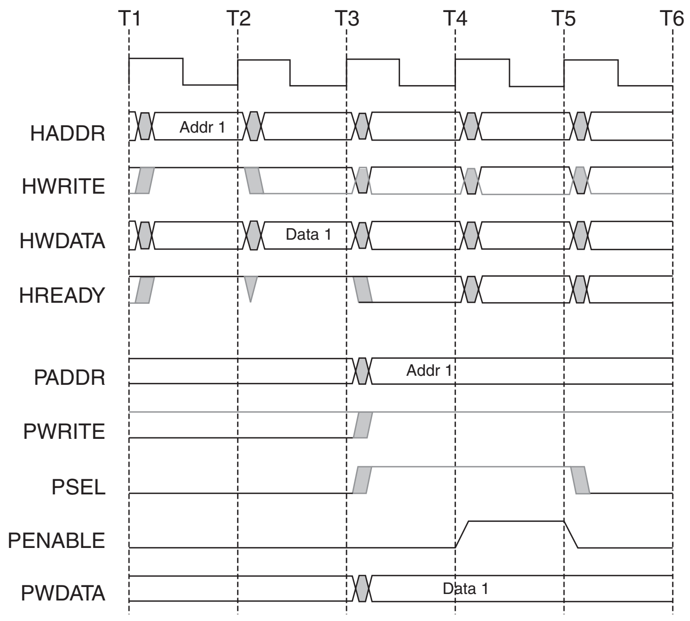

FIGURE 3.14

Write data request from the AHB to the APB bus [1]
Source: ARM Inc.

shows a similar scenario, for a write transfer on the APB bus. The bridge samples the address, control, and data signals from the master, holding these values for the duration of the write transfer on the APB bus (as it switches through the *SETUP* and *ENABLE* states).

3.1.2 AMBA 3.0

The AMBA 3.0 bus architecture specification [2] introduces the Advanced eXtensible Interface (AXI) bus that extends the AHB bus with advanced features to support the next generation of high performance MPSoC designs. The goals of the AXI bus protocol include supporting high frequency operation without using complex bridges, flexibility in meeting the interface, and performance requirements of a diverse set of components, and backward compatibility with AMBA 2.0 AHB and APB interfaces. We now look at the AXI specification in more detail (Figure 3.15).

3.1.2.1 *Advanced Exensible Interface*

The AXI bus standard proposes a burst-based, pipelined data transfer bus, similar to the AHB bus, but with additional advanced features and enhancements. The main features of AXI and its differences with the AHB standard are presented in Table 3.1, and elaborated in more detail below.

The AXI specification describes a high level channel-based architecture for communicating between masters and slaves on a bus. Five separate channels are defined: *read address*, *read data*, *write address*, *write data*, and *write response*.

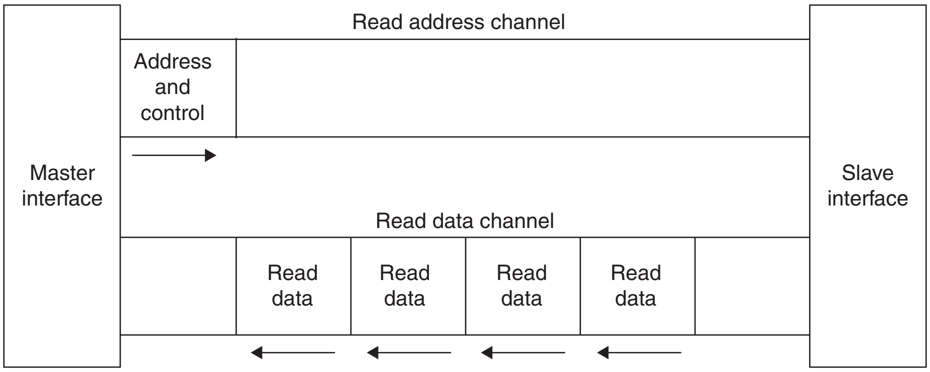

FIGURE 3.15(a)

AMBA AXI channel architecture: read address and read data channels

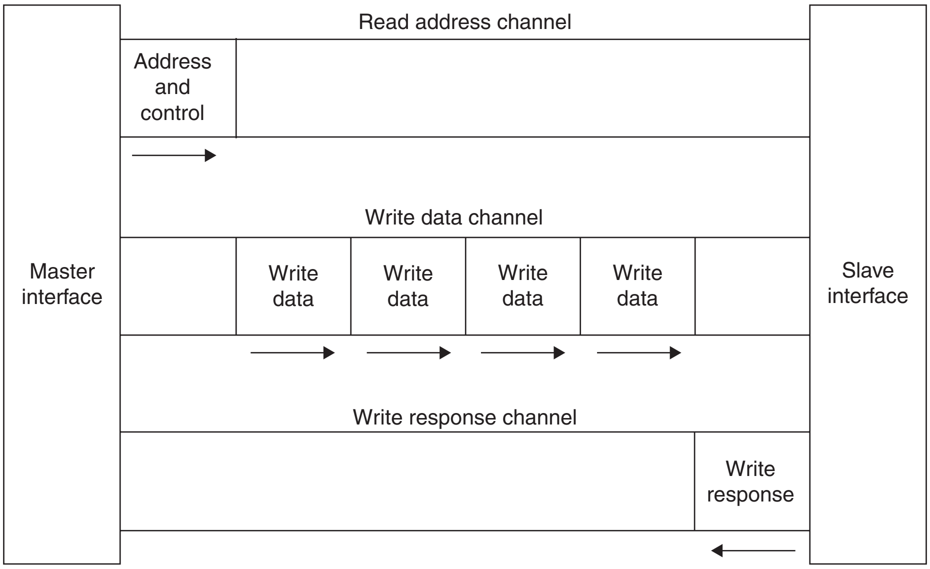

FIGURE 3.15(b)

Write address, write data, and write response channels [2]
Source: ARM Inc.

Just like for the data bus in AHB, the data channel width in AXI can range from 8 to 1024 bits. The read channels are shown in Fig. 3.15(a). The address and control information for a read transfer is sent by the master on the read channel, while the read data and response information from the slave is received on the read data channel. Figure 3.15(b) shows the write data channels. The address and control information for a write transfer is sent on the write address channel, while the write data is transmitted on the write data channel. A one byte strobe signal is

Table 3.1 Contrasting features of AXI and AHB

AMBA 3.0 AXI	AMBA 2.0 AHB
Channel-based specification, with five separate channels for read address, read data, write address, write data, and write response enabling flexibility in implementation.	Explicit bus-based specification, with single shared address bus and separate read and write data buses.
Burst mode requires transmitting address of only first data item on the bus.	Requires transmitting address of every data item transmitted on the bus.
OO transaction completion provides native support for multiple, outstanding transactions.	Simpler SPLIT transaction scheme provides limited and rudimentary outstanding transaction completion.
Fixed burst mode for memory mapped I/O peripherals.	No fixed burst mode.
Exclusive data access (semaphore operation) support.	No exclusive access support.
Advanced security and cache hint support.	Simple protection and cache hint support.
Register slice support for timing isolation.	No inherent support for timing isolation.
Native low-power clock control interface.	No low-power interface.
Default bus matrix topology support.	Default hierarchical bus topology support.

included for every 8 bits of write data, to indicate which bytes of the data bus are valid. This is useful for cases where there is a mismatch between the size of the data being transferred and the data bus width. A separate write response channel provides the slave a means to respond to write transactions. A write completion signal occurs once for every burst (and not for every data transfer) to indicate the status of the write at the slave. The five separate channels provide implementation flexibility to a designer and can be implemented in any one of three ways:

1. *Shared address bus and shared data buses (SASD)*: A single shared address bus is coupled with a bidirectional data bus that handles both reads and writes. Such a configuration is typically useful for smaller, low complexity embedded systems.
2. *Shared address bus and multiple data buses (SAMD)*: A single shared address bus is coupled with separate, unidirectional read and write data buses. Since the address bus bandwidth is typically less than that of the data buses (as only one address needs to be sent for a burst data transfer), interconnect complexity can be reduced while still maintaining performance, by using a shared address bus.

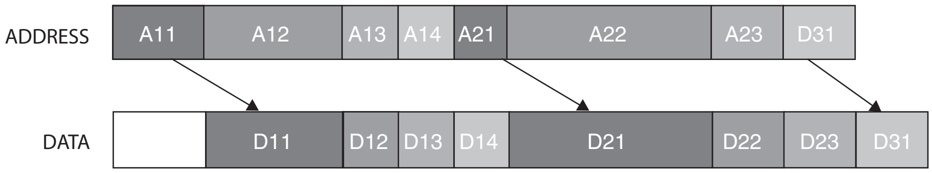

FIGURE 3.16(a)

Burst addressing modes for AMBA 2.0 AHB bus

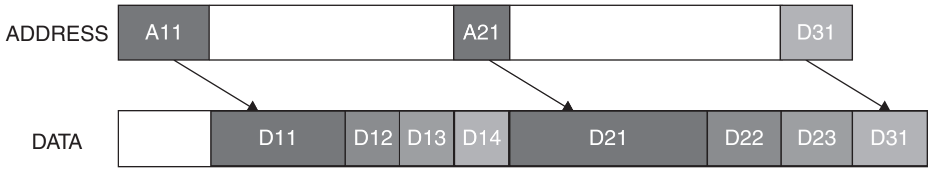

FIGURE 3.16(b)

AMBA 3.0 AXI bus

3. *Multiple address buses, multiple data buses (MAMD)*: A separate address bus for reads and writes is coupled with separate read and write data buses. This configuration has the largest interconnect complexity, but also the best performance of the three implementation alternatives.

The *MAMD* mode in particular allows multiple concurrent read and write transactions to occur independent of each other, which can be very useful for high performance SoC designs. In contrast, AHB only explicitly supports the *SAMD* implementation mode.

One significant difference between AXI and AHB is the way addressing is handled during burst data transfers. In AHB, every data transfer in a burst requires an address to be transmitted on the address bus. In contrast, AXI requires the address of only the first data item in the burst to be transmitted. Figure 3.16(a) shows burst data transfers on an AHB bus that require an address for every data item transmitted. Contrast this with the same scenario on an AXI bus, shown in Fig. 3.16(b), where only the address of the first data item in a burst is transmitted. It is the responsibility of the slave to calculate the address of the subsequent transfers in the burst. Because only a single address is transmitted per burst, the address buses in AXI are freed up to handle other transactions. Figure 3.17 shows how read and write transactions can occur simultaneously in an *SAMD* implementation of AXI, because the address bus is freed up during a burst. The AHB in contrast must wait for a burst to complete before initiating another transfer, which results in under-utilization of its data buses. Another enhancement in AXI, compared to AHB, comes in the form of support for an additional burst type. AXI not only supports all the incrementing and wrapping burst types present in AHB, but also an additional *fixed burst mode*. In this mode, the address of every data item transferred in a burst remains the same. This burst type is very useful for repeated accesses to the same location, such as for data transfers with an I/O peripheral FIFO (first-in-first-out).

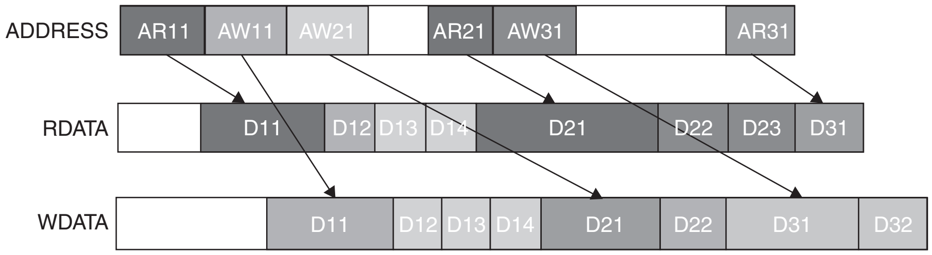

FIGURE 3.17

Better utilization of data buses in AXI

Another important AXI feature is its support for *out-of-order (OO) transaction completion* which is an advanced feature that maximizes data throughput and improves system efficiency. AXI masters have the ability to issue multiple outstanding addresses, which means that transaction addresses can be issued without waiting for earlier transactions to complete. This is accomplished by assigned IDs to read and write transactions issued by the masters. The AXI specification lays down certain guidelines to govern the ordering of transactions. Transactions from different masters have no ordering restrictions and can complete in any order. Transactions from the same master but with different ID values can complete in any order. However, a sequence of write transactions with the same ID value must complete in the same order in which the master issued them. For a sequence of read transactions with the same ID value, two possible scenarios exist: (i) reads with the same ID value are from the same slave, in which case it is the responsibility of the slave to ensure that the read data returns in the same order in which the addresses are received; and (ii) reads with the same ID value are from different slaves, in which case the AXI bus logic must ensure that the read data returns in the same order that the master issued the addresses in.

The ability to finish transactions OO allows completion of transactions to faster regions of a memory (or peripheral), without waiting for earlier transactions to slower regions. This feature reduces the effect of transaction latency and improves system performance. Figure 3.18 shows a comparison between AHB and AXI for a scenario where a master must access a slow (i.e., high latency) slave. Figure 3.18(a) shows how an access to a slow slave in AHB holds up the master and the bus till the slave is ready to return the data. Note that using SPLIT transactions on AHB can free up the bus, but not the master, which is still stalled. In contrast, in the AXI case, as shown in Fig. 3.18(b), the master need not wait to get the data back from the slave before issuing other transactions. Both the bus and the master are freed up in AXI, which allows better performance, higher efficiency, and greater bus utilization. Although theoretically any number of transactions can be reordered on the AXI bus, a practical limit is placed by the read/write data *reordering depth* at the slave interfaces. The read or write data reordering depth of a slave is the number of addresses pending in the slave that can be reordered. A slave that processes all transactions in order is said to have a data reordering depth of 1. This reordering depth must be specified by the designer for both reads and writes, and involves a trade-off between hardware complexity and parallelism in the system—a larger reordering

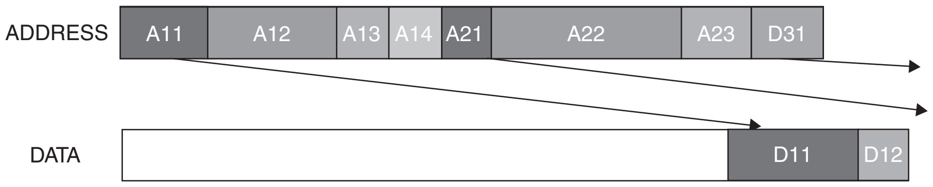

FIGURE 3.18(a)

Transaction sequence for access to a slow slave on AHB bus

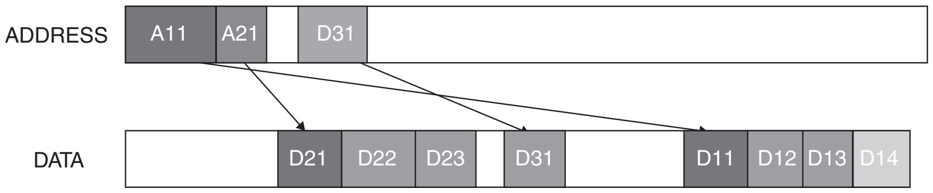

FIGURE 3.18(b)

AXI bus

depth requires greater hardware complexity, but also improves parallelism and possibly overall system performance. However, there is a limit beyond which increasing the reordering depth for a slave does not improve performance because there is typically a limit to the maximum number of concurrent transfers possible in an application, as shown by Pasricha et al. [11].

Other areas where AXI improves on and differs from the AHB feature set include:

- *Semaphore operations*: AXI provides support for semaphore type operations using an exclusive access mechanism that does not require the bus to remain locked to a particular master for the duration of the transaction. The support for semaphore type operations in AXI therefore does not affect maximum achievable bandwidth or data access latency. In contrast, AHB does not provide any support for semaphore type operations. An exclusive access on AXI is initiated when a master performs an exclusive read from a slave address location. At some later time, the master attempts to complete the exclusive access by attempting to write to the same location. The exclusive write access is signaled by the slave as successful if no other master has written to the location between the read and write accesses, or as a failure if another master has written to the location between the read and write accesses. A slave is required to have additional logic (such as a monitor unit for each exclusive-capable master ID that can access it) if it supports exclusive accesses.
- *Cache support*: AXI provides support for system level caches and other performance enhancing components with the help of two 4-bit cache hint signals, one for each of the read and write channels. These cache hint signals provide additional information about how the transaction can be processed. While AHB provides support for basic cache hints (with a 2-bit signal) such as if data is bufferable or cacheable, AXI extends this by providing additional

signals to specify write-through and write-back cache allocation strategies for reads and writes, as well as providing the designer the option of customizing the hint signals for other purposes such as flushing cache and page tables.

- *Protection support*: AXI utilizes two 3-bit signals, one each for the read and write data channels, to provide protection against illegal transactions. There are three levels of protection possible, each represented by a single bit of the protection signals: (i) *normal or privileged access*, used by certain masters to indicate their processing mode and to obtain access to special resources in a system, since privileged accesses typically provide greater access within a system, (ii) *secure or non-secure accesses*, used in systems where a greater degree of differentiation between processing modes is required, and (iii) *instruction or data accesse*s, to indicate if the transaction is an instruction or a data access. In comparison, AHB provides support for normal/privileged accesses and instruction/data accesses, but not for secure/non-secure accesses.
- *Low power support*: AXI supports an optional set of signals for low power operation. These signals target two classes of peripherals. The first consists of peripherals that require a power-down sequence, and can have their clocks turned off (to save power) only after they enter a low power state. These peripherals require a signal from the system clock controller to determine when to begin the power-down sequence. The second type of peripherals are those that do not require a power-down sequence, and can assert a signal to indicate when it is acceptable to turn off their clock. AXI provides support for both types of signals. In contrast, AHB does not include any signals for low power peripheral operation support.
- *Recommended topology*: The AXI specification assumes a default bus matrix topology implementation, without any requirement for complex bridges. Such an implementation is in keeping with the advanced feature set of AXI, and is suitable for contemporary high performance designs. However, this does not limit AXI in any way from being used in a hierarchical bus topology, if required. The AHB specification, on the other hand, assumes a default hierarchical bus topology arrangement, involving an AHB bus interfacing with an APB bus via a bridge. This is in keeping with the comparatively less advanced AHB feature set, but does not limit it from being implemented in a bus matrix topology.
- *Register slice support*: The clock frequency on a bus puts a limit on the length of its bus wires during physical layout. This is because a signal can only travel a finite distance on the chip in a single clock cycle [12]. As the clock frequency of the bus is increased, the clock cycle period (which is the inverse of the clock frequency) is reduced, and the distance that can be traveled by a signal shrinks. In fact, it can take multiple cycles for a signal to travel between the ends of a chip [13] in Deep Submicron (DSM) process technologies. For high performance SoC designs, high clock frequencies on the bus are essential to meet performance constraints. To sustain these high bus clock frequencies and ensure correct operation on long interconnects, AXI proposes using one or more register slices on a bus. These register slices latch signal information coming from the source and then retransmit it toward the destination.

The advantage here is that the signal need only cover the distance between latches in a single clock cycle. Thus, by inserting register slices, timing closure (i.e., ensuring timing requirements of a design are met) becomes relatively easier. Introducing the register slices will of course increase the latency (in terms of number of cycles required) for communicating on the bus. However, the bus can now be operated at a much higher frequency (compared to a bus with no register slices inserted) which can improve overall system performance. AHB, in contrast, does not provide any such means of alleviating the problem of meeting timing for high performance, high frequency systems.

AXI Bus Matrix Topology

Just like the AHB protocol, AXI can be connected in a bus matrix topology. ARM distributes an AXI configurable interconnect (ACI) IP at the register transfer level (RTL) called *PL300* [30] (recently superseded by the next version—*PL301* [31]) that allows designers to connect several masters and slaves together with a configurable AXI bus matrix fabric. The structure of the AXI bus matrix is somewhat similar to that of the AHB bus matrix described earlier, but differs in its support for additional features of the AXI protocol such as independent control for decoupled read, write, and response channels, and OO transaction completion. Components are connected to the AXI bus matrix using interface routers. Each master is connected to a slave interface router, and each slave is connected to a master interface router. The interface routers are part of the AXI bus matrix fabric. These routers essentially consist of multiplexing and de-multiplexing elements to ensure appropriate connectivity with other components connected to the bus matrix. The select signal for each router is generated from a control block that is unique for each channel and interface. The control blocks store the routing information necessary to enforce the ordering constraints within the AXI protocol, and consist of arbiters, decoders, content addressable buffers (CABs), and FIFO elements. The bus matrix fabric does not buffer addresses and data—slaves supporting outstanding transactions must provide the required storage locally. Much like in the case of the AHB bus matrix, a full AXI bus matrix (that connects all the masters to all the slaves in a system) supports high bandwidth but can be prohibitively expensive as it requires a very large number of wires and bus logic components. In a lot of cases where a somewhat lower performance is acceptable, a partial AXI bus matrix that clusters components onto shared buses to reduce the number of wires and bus logic components in the matrix may be more suitable. Pasricha et al. [9] showed how a partial AXI bus matrix can be automatically synthesized from a full AXI bus matrix for MPSoC designs. This methodology, which reduces the number of buses in the matrix while satisfying all application performance constraints, is described in more detail in Chapter 6.

3.1.3 IBM CoreConnect

The IBM CoreConnect [3] on-chip communication architecture standard is another popular synchronous bus-based standard that shares many similarities with the AMBA standard. It defines three types of buses: (i) *processor local bus (PLB)*, which is a high performance bus used to connect high speed processor and memory components, (ii) *on-chip peripheral bus (OPB)*, which is used to connect lower performance peripheral components, and (iii) *device control register (DCR)* bus, which is a simple, high latency bus used to access the status and control registers of the PLB and OPB masters. The CoreConnect standard targets a hierarchical bus topology implementation, similar to AMBA 2.0, with the OPB and PLB buses interfacing with each other using a bridge as shown in Fig. 3.19. We now look at the specifications of the PLB, OPB, and DCR buses in more detail.

3.1.3.1 *Processor Local Bus*

The PLB is a synchronous, high performance bus, similar in many aspects to the AMBA 2.0 AHB, and used to interconnect high performance processor, ASIC, and memory cores. The main features of PLB are summarized below:

- Shared address, separate read and write data buses (*SAMD*).
- Decoupled address, read data, write data buses.
- Support for 32-bit address, 16, 32, 64, and 128-bit data bus widths.
- Dynamic bus sizing—byte, half-word, word, and double-word transfers.
- Up to 16 masters and any number of slaves.
- AND–OR implementation structure.
- Pipelined transfers.
- Variable or fixed length burst transfers.
- Support for 16–64 byte bursts.
- SPLIT transfer support.

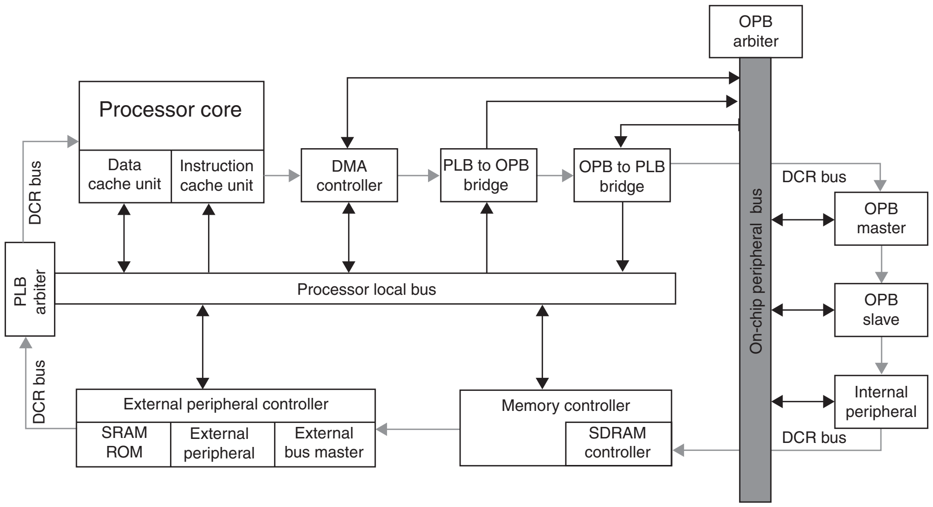

FIGURE 3.19

An example of CoreConnect-based SoC design [3]
Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

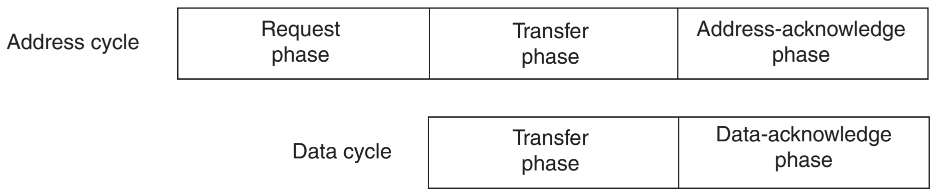

FIGURE 3.20

PLB address and data cycles [3]
Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

- Overlapped read and write transfers (up to 2 transfers per cycle).
- Centralized arbiter.
- Four levels of request priority for each master, programmable secondary arbitration.
- Locked transfer support for atomic accesses.
- Latency timer (to limit a master's tenure on PLB during bursts).

A PLB transaction consists of two cycles: address and data, as shown in Fig. 3.20. The address cycle has three phases: request, transfer, and address acknowledge. A PLB bus transaction is initiated when a master drives its address and control signals, and sends a bus access request to the arbiter, during the request phase. Once the arbiter grants the master access to the bus, the address and control information is sent to the slave in the transfer phase. The address cycle is terminated by the slave latching the address and control information during the acknowledge phase. The data cycle has two phases: transfer and acknowledge. The master drives the write data or samples the read data bus, during the transfer phase. Data acknowledge signaling is required for every data item transmitted on the bus, during the acknowledge phase.

Figure 3.21 shows an example of overlapped transfers on the PLB bus, where two masters perform a single read and a single write transfer each. Due to the decoupled nature of the address, read data, and write data buses, the address cycles can be overlapped with the read or write data cycles, and the read data cycles can be overlapped with the write data cycles. The split bus capability allows for the address and data buses to have different masters at the same time. Support for address pipelining allows a new transfer to begin even before the current transfer has finished. This reduces bus latency by allowing the latency associated with a new transfer request to be overlapped with an ongoing data transfer in the same direction.

A simple write transfer on the PLB bus is shown in Fig. 3.22. A master requests access to the bus for a single write transfer in the first cycle. Due to the slave asserting its wait signal (*Sl_wait*) to the arbiter, which indicates that the slave is unable to participate in the transaction, the transaction is stalled. The arbiter continues to drive the address and control signals to the slave through this entire period, till it receives the slave address acknowledge (*Sl_AddrAck*) signal in the fourth cycle. The slave then asserts the write data acknowledge (*Sl_wrDAck*) and write transfer complete (*Sl_wrComp*) signals to indicate the end of the transaction.

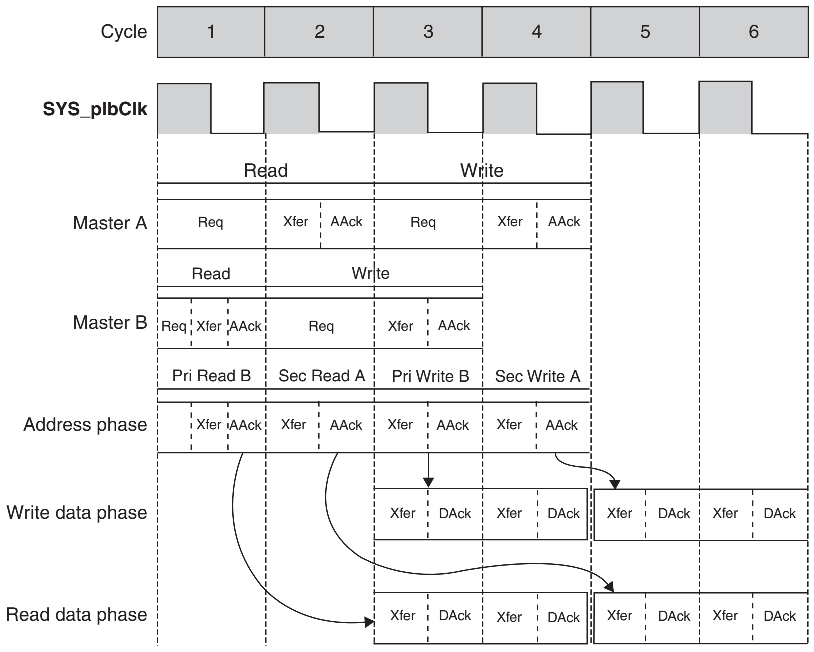

FIGURE 3.21

Example of overlapped PLB transfers [3]

Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

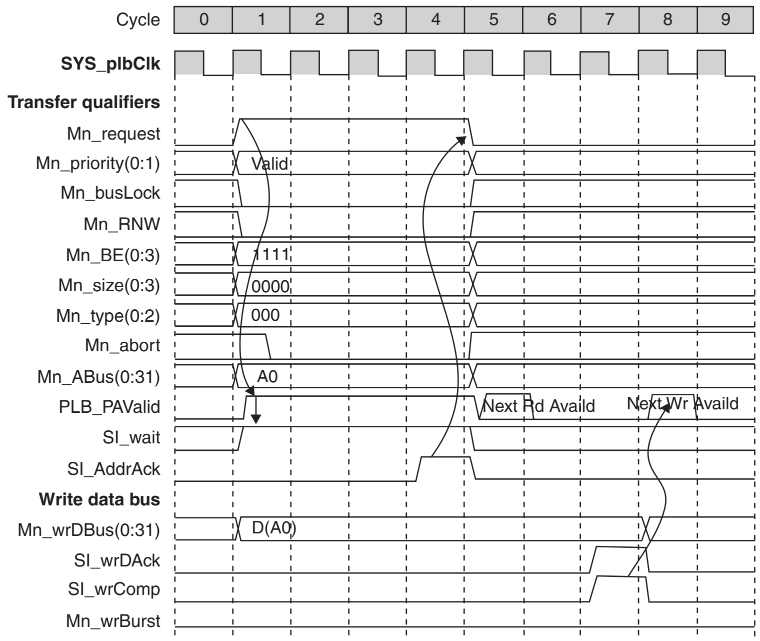

FIGURE 3.22

Example of a single write transfer on a PLB bus [3]

Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

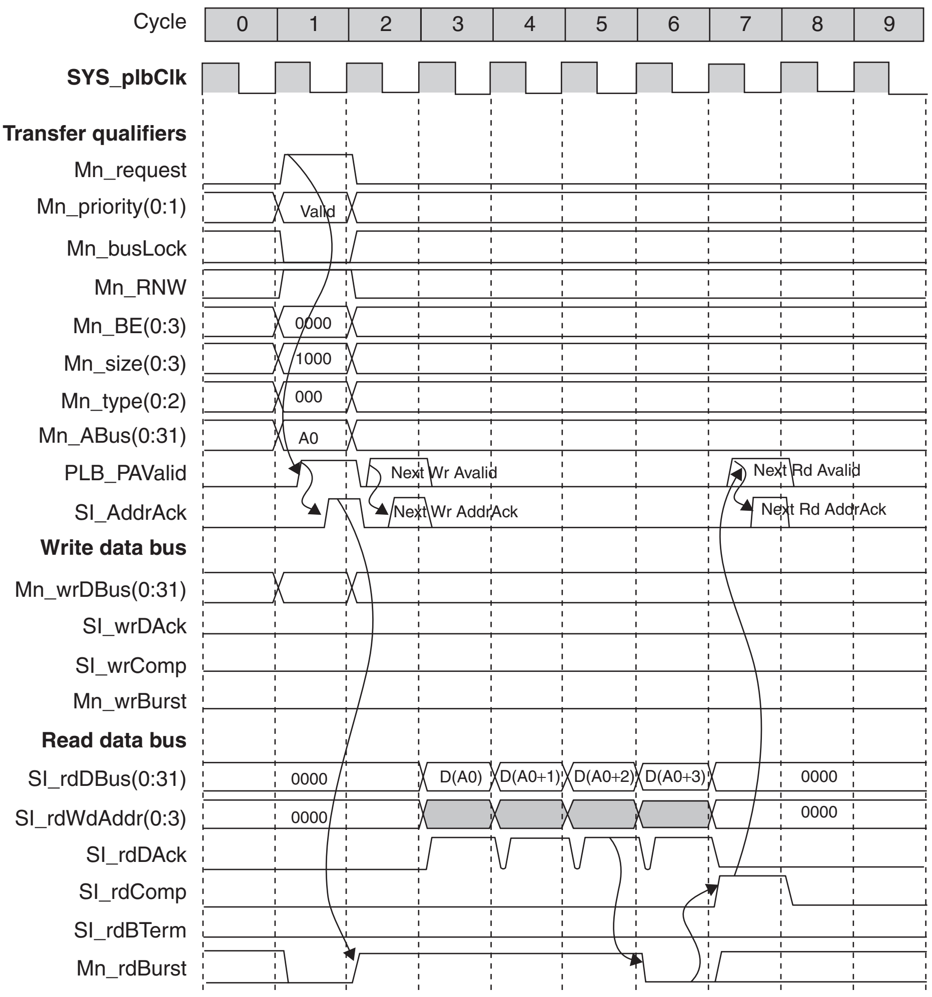

FIGURE 3.23

Read burst transfer (of length 4) on a PLB bus [3]

Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

Figure 3.23 illustrates a read burst transfer of length 4 on the PLB bus. The master asserts the request for bus access to the arbiter in the first cycle and is granted access in the same cycle. The slave receives the address and control information and drives the address acknowledge (*SI_addrAck*) signal to the arbiter in the same cycle. The read data is driven onto the read data bus starting from the third cycle, along with a read data acknowledge (*SI_rdDAck*) signal for each data item. The slave asserts the read transaction complete signal (*SI_rdComp*) in the cycle prior to the last read data acknowledge (*SI_rdDAck*). Note that only the address of the first burst data needs to be transmitted by the master and it is the responsibility of the slave to internally increment the addresses sequentially for each transfer (just like in AMBA 3.0 AXI).

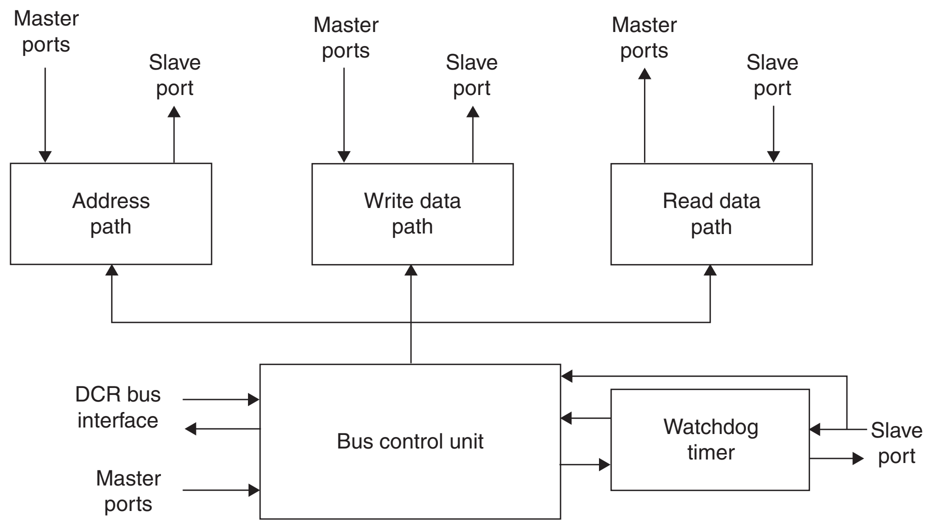

FIGURE 3.24

PLB arbiter block diagram [3]
Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

Figure 3.24 shows a block diagram of the PLB arbiter that handles arbitration for up to eight masters. It consists of several components that are described below:

- *Bus control unit (BCU)*: This supports arbitration for masters on the PLB bus. Each master typically drives a 2-bit signal that encodes four priority levels (highest, high, low, lowest), indicating the priority of the request to the arbiter. The arbiter uses this information, in conjunction with the requests from other masters to decide which master to grant the bus access to. In case of a tie, the arbiter provides a choice of using either a fixed (static) priority scheme or a fair, round-robin (RR) priority scheme. The BCU consists of four 32-bit DCRs to control and report its status: (i) *PLB arbiter control register (PACR)*, which is used to program the choice of static or RR priority schemes to be used in case of an arbitration tie, (ii) *PLB error address register (PEAR)*, which contains the address of the access where a bus time-out error occurred, (iii) *PLB error status register (PESR)*, which contains bits to identify time-out errors on PLB bus transfers, the master initiating the transfer, and the type of transfer (read or write), and (iv) *PLB revision ID register (PREV)*, that contains the revision ID of the PLB arbiter core.
- *Address Path Unit:* It contains necessary MUXes to select master address that will be driven to the slaves.
- *Read Data Path Unit:* It contains necessary steering logic for the read data bus.
- *Write Data Path Unit:* It contains necessary steering logic for the write data bus.
- *Watchdog Timer:* It provides the necessary handshake to complete a transfer, if a master's request times out on the PLB.

To control the maximum latency of a particular application, PLB supports a master latency timer in each master. This latency timer consists of two 8-bit registers: a *Latency Count Register* and a *Latency Counter*. The Latency Count Register is software programmable, with the option of hardwiring the lower 4 bits to ensure a minimum latency of 16 clock cycles. The Latency Counter is used as a clock cycle counter and is not accessible via software code. It is enabled and begins counting the clock cycles during burst data transfers. Once the value of the Latency Counter reaches the value programmed in the Latency Count Register, the master is required to terminate its burst if a request to the bus arrives from another master of equal or higher priority. This timeout mechanism ensures that no master remains parked on the bus for excessive periods of time, and ensures that high priority requests are serviced with low latency.

3.1.3.2 *On-Chip Peripheral Bus*

The OPB is a synchronous bus used to connect high latency peripherals, and alleviate system performance bottlenecks by reducing the capacitive loading on the PLB (since connecting a large number of components on a bus results in high capacitive loading of the bus that increases signal propagation delay and reduces performance). A bridge module is used to interface the OPB bus with a PLB bus. The OPB is more advanced than the simple APB peripheral bus used in AMBA, as indicated by its main features, summarized below:

- Shared address bus, multiple data buses (*SAMD*).
- Up to a 64-bit address bus width.
- 32- or 64-bit read, write data bus width support.
- Support for multiple masters.
- Bus parking (or locking) for reduced transfer latency.
- Sequential address transfers (burst mode).
- Dynamic bus sizing—byte, half-word, word, and double-word transfers.
- MUX-based (or AND–OR) structural implementation.
- Single cycle data transfer between OPB masters and slaves.
- 16 cycle fixed timeout provided by arbiter, to reduce transfer latency (can be disabled by a slave).
- Slave retry support, to break possible arbitration deadlocks.

A basic data transfer on an OPB bus is shown in Fig. 3.25. A master requests access to the bus for a read data transfer in the second cycle. The arbiter grants access to the bus, based on its arbitration scheme, and the master subsequently assumes ownership of the bus by asserting the select (*M1_select*) signal. The master then drives the address and control signals, which is sampled off the bus by the slave. The slave drives the read data onto the data bus and asserts the transfer acknowledge (*Sl2_xferAck*) signal in the next cycle. The master latches the data off the bus in the following cycle and de-asserts select to end the transaction.

For reducing arbitration latency during multiple transfers, a master can park or lock itself on the OPB bus. The parked master is allowed to access the bus without any delay due to an arbitration cycle, as long as no request is asserted by another master. Like in the case of the PLB, the OPB masters capable of long parked/locked

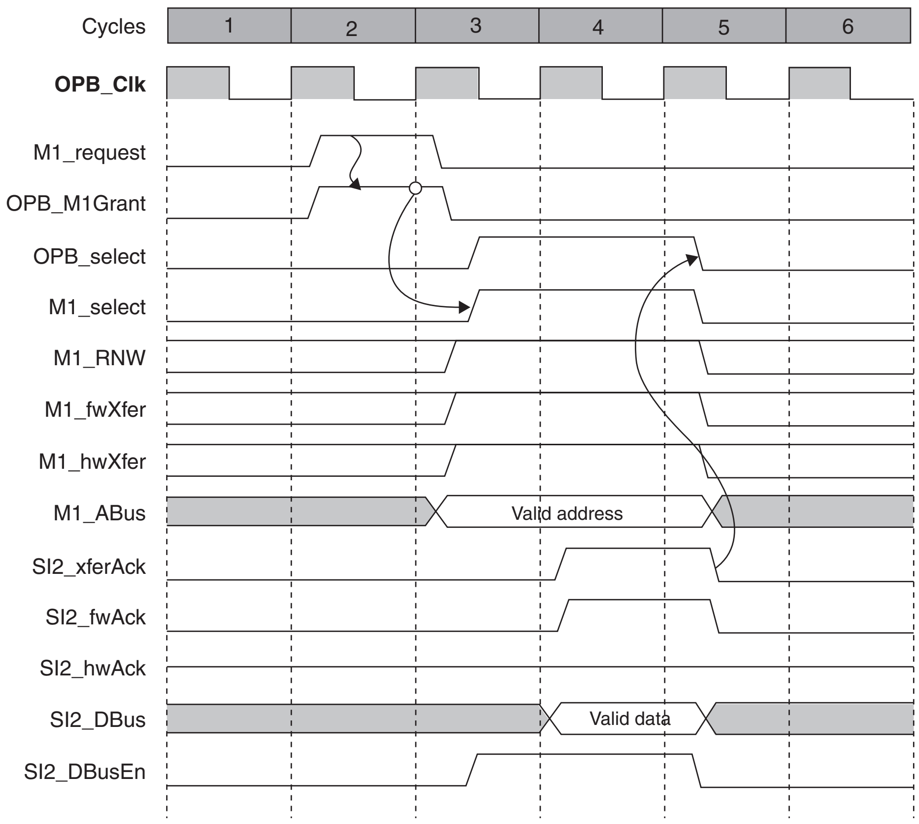

FIGURE 3.25

Basic read data transfer on OPB [3]

Reprint Courtesy of International Business Machines Corporation copyright (2001) © International Business Machines Corporation

access have Latency Counters to insure a low latency response for requests from other masters. Multiple data transfers to sequential addresses can take advantage of the sequential transfer mode in OPB, which is similar to a burst transfer. Unlike the PLB, a master on the OPB bus must drive addresses for each data item transferred on the bus. OPB also allows the overlapping of a new arbitration request with the final cycle of an ongoing data transfer, to avoid wasting a cycle for bus request and improve performance. The arbiter module is simpler than the PLB arbiter, with only two registers: one for programmable priority and another for controlling bus parking. There is flexibility in choosing either a fixed (static) scheme or a fair, RR scheme. These options are dynamically programmable, allowing for adjustments in priority based on varying traffic profiles or operation modes. The watchdog timer module in the arbiter implements a timeout feature if a slave does not respond to a master request within 16 clock cycles. If a slave must take longer to complete the transfer, it is allowed to inhibit the timer counter in the watchdog.

To alleviate possible deadlock scenarios on the bus, OPB supports the slave retry operation. This allows an OPB slave to assert a retry signal if the slave cannot perform the required bus operation at a particular instant of time. The bus master responds to this signal by immediately terminating its transfer and relinquishing control of the bus for at least one cycle, so that the arbiter can re-arbitrate the bus. This allows the slave a chance to access the OPB in order to resolve the deadlock condition. Note that this mechanism may still be insufficient to guarantee that all possible deadlock conditions will be alleviated. However, the retry operation does provide OPB masters with sufficient information to detect a deadlock situation, and to take corrective action.

OPB, much like PLB, also supports dynamic bus sizing which allows components that have different data interface widths than the OPB data bus to operate seamlessly. When a master transfers data that is wider than the data bus width, it must split the transfer into two or more operations. Similarly, if a data item being transferred has a smaller width than the data bus width, it must be appropriately aligned on a subset of the bus lines. The OPB bus permits byte (8-bit), half-word (16-bit), full-word (32-bit), and double-word (64-bit) sized transfers.

3.1.3.3 *Device Control Register Bus*

The DCR bus is a synchronous bus designed to transfer data between a CPU's general-purpose registers (GPRs) and the device control registers (DCRs) of the slave components in the system. The DCR bus removes device configuration registers from the global memory address map. It allows the lower performance status and control read/write transfers to occur separately, and concurrently with high speed transfers on the PLB and OPB buses, thus improving system response time and overall performance. It is assumed that in a typical SoC environment where DCR master and slave components are operating at different clock frequencies, the slower clock's rising edge always corresponds to the faster clock's rising edge. DCR transactions control the configuration of on-chip peripherals such as interrupt controllers, timers, arbiters, bridges, etc. The main features of the DCR bus are summarized below:

- 10-bit, up to 32-bit address bus.
- 32-bit read and write data buses.
- 4-cycle minimum read or write transfers (extendable by slave or master).
- Slave bus timeout inhibit capability.
- Multi-master arbitration.
- Privileged and non-privileged transfers.
- Daisy-chain (serial) or distributed-OR (parallel) bus topologies.

The DCR bus consists of the address, read and write data buses, and the DCR read, DCR write, master ID, privilege level, timeout wait, and acknowledge signals. Slaves can have privileged registers that can only be accessed by privileged transactions from a master. Any non-privileged transaction meant for a privileged DCR is ignored by the slave and results in a timeout. A typical transfer on the DCR bus is initiated by a master asserting the DCR read or write command signals, and driving the address and appropriate control signals. Slaves decode the command, address, privilege level, and master ID to determine whether to claim the transfer or not.

A slave can claim a transfer by asserting the timeout wait or acknowledge signals. Since requests can time out if a response is not received by the master, a slave that takes longer to complete the transfer must assert the timeout wait signal to prevent a timeout. Asserting the acknowledge signal implies that a write operation is complete, or that read data has been driven onto the read data bus. If no slave responds to the transfer request, a timeout occurs and the master terminates the command.

The DCR bus can be implemented by daisy-chaining the slave components or by creating a distributed-OR structure out of the slave devices. The daisy-chain approach allows for easier chip level wiring while the distributed-OR approach allows for easier chip level timing closure. For the case of the daisy-chain configuration, data moves along the ring-like network connecting all the slave components, and each slave component either passes along the unmodified data input, or puts its data onto its data bus output. In the distributed-OR implementation, each slave directly receives data from the master, and places its data output onto the system OR logic. It is possible for multiple masters to be connected to the DCR bus. In such a case, an arbiter is required to negotiate access to the DCR bus.

3.1.4 STMicroelectronics STBus

STMicroelectronics' STBus [4] on-chip communication architecture is an evolutionary on-chip interconnection standard developed for microcontroller consumer applications such as set-top boxes, ATM networks, digital cameras, etc. It is closely related to the VSIA (virtual sockets interface alliance; described in Section 3.2.2) industry interface standard [14] to ease compatibility and integration with third party IP blocks. The STBus standard defines three types of synchronous buses (or bus protocols) having varying levels of complexity in terms of performance and implementation: (i) *Type 1*, which is the simplest bus protocol intended for peripheral register access, (ii) *Type 2*, which is a more complex bus protocol that supports pipelined operations and SPLIT transactions, and (iii) *Type 3*, which is the most advanced bus that implements OO transactions, compound operations, and transaction labeling/hints. These buses are implemented with a MUX-based structure, and can be arranged in either a shared bus, partial crossbar (partial bus matrix), or a full crossbar (full bus matrix) topology. We describe each of these bus protocols in more detail below.

3.1.4.1 *Type 1*

The Type 1 or *peripheral* STBus standard is the simplest protocol in the STBus family that is meant to interconnect components such as general-purpose input/output (GPIO), UARTs, and simple controllers that require medium data-rate communication with the rest of the system. A simple handshaking mechanism is used to ensure correct transmission. Operations on the buses are defined in terms of *OPCs* which define, for instance, whether an operation is a read or a write. The Type 1 bus supports the LOAD (read) and STORE (write) data operations, with an address bus size of 32 bits, and a possible data size of a byte (8-bit), half-word (16-bit), word (32-bit), and double-word (64-bit). Figure 3.26 shows a simple data transfer on the Type 1 bus. The initiator (master) sends a data transfer request (*REQ*) to

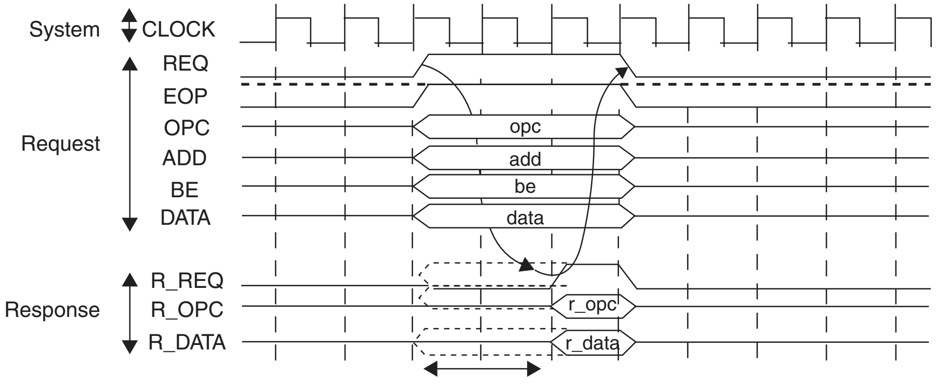

FIGURE 3.26

Basic data transfer on STBus Type 1 bus [4]
Source: STMicroelectronics

the target (slave) by sending the *OPC* for the transfer (either read or write), the transfer address (*ADD*), and the byte enable (*BE*) to specify which bytes in the bus are significant, based on the width of the data involved in the transfer (1–8 bytes). The write data bus (*DATA*) to send the write data (for a write data transfer). The slave indicates it has received the transfer request by asserting the handshake signal (*R_REQ*), and then proceeds to return the read data on the read data bus (*R_DATA*) or write data in its address space for a write data transfer. The slave returns an optional response opcode (*R_OPC*) to indicate any errors during the transaction.

The Type 1 bus is similar to the IBM CoreConnect DCR bus, since it is also used to program the internal configuration registers of components connected to the STBus communication architecture. A register decoder block is responsible for performing address decoding, data transfer routing, and arbitration (if more than one master is connected to the bus) for transfers on the Type 1 bus.

3.1.4.2 *Type 2*

The Type 2 or the *basic* STBus standard supports all the Type 1 functionality, and additionally provides support for pipelined operations, SPLIT transactions, compound operations, source labeling, and some priority and labeling/hint information. This bus protocol is targeted at high performance components. The use of SPLIT transactions and pipelined operation improves bus efficiency and performance. The Type 2 standard supports the basic LOAD (read) and STORE (write) operations, with an address bus size of 32 bits and allowed data bus sizes of 8, 16, 32, 64, 128, or 256 bits. Additionally, the Type 2 standard also supports compound operations which are built from one or more primitives. The supported standard compound operations include:

- *READMODWRITE*: An atomic operation that transfers read data from the slave to the master, but leaves the slave *locked* until a write transfer from the same master completes, replacing the information at the specified address in the slave.

- *SWAP*: An atomic operation that exchanges a data value from the master with the data held in a specified location in a slave.
- *FLUSH*: An operation used to ensure the coherence of main memory while allowing local copies associated with a slave to remain coherent. The operation returns a response when any copies of the data associated with a physical address (which are held by a slave module) are coherent with the actual data at the physical address. The slave may retain a copy of the data.
- *PURGE*: An operation used to ensure the coherence of main memory while ensuring that stale local copies are destroyed. The operation returns a response when any copies of the data associated with a physical address (which are held by a slave module) are coherent with the actual data at the physical address, while removing any copies of the data held by the slave.
- *USER*: This is reserved for user defined operations that can implement useful operations specific to particular applications.

3.1.4.3 *Type 3*

The Type 3 or *advanced* STBus standard supports all Type 1 and Type 2 functionality, but additionally supports packet shaping and OO transaction completion. These features make this bus protocol suitable for very high performance components. The Type 3 supports the same basic and compound operations as the Type 2 standard. Packet shaping allows optimum bandwidth allocation in Type 3 buses, with only the minimum number of clock cycles required to carry out a transaction being used. In Type 1 and Type 2 buses, every request requires a response, which wastes bandwidth. In contrast, a Type 3 requires only a single response from a slave for multiple write data operations by a master, or a single read request from a master for multiple data reads from a slave. Due to this asymmetry between request and response phases, the bandwidth allocation in the Type 3 protocol is optimized compared to the Type 1 and Type 2 protocols. The use of OO transaction completion further reduces latency on the bus since a master waiting for a response from a slave no longer blocks access to other slaves. Transaction IDs associated with data transfers allow the components to have up to 16 transactions in progress.

3.1.4.4 *STBus Components*

The STBus node shown in Fig. 3.27 is the main component of the STBus on-chip communication architecture. It consists of two main blocks: the *control logic* and the *data path*. The control logic is responsible for the arbitration, decoding, and contains other bus logic required for implementing advanced protocol features (such as OO transaction completion), while the data path represents the topology of the communication architecture, which can be a shared bus, a full crossbar, or a partial crossbar configuration. Master components connect to the initiator interfaces, while slaves connect to the target interfaces of this node architecture.

The control logic is responsible for arbitration, and supports several arbitration schemes which are described below:

- *Fixed priority*: The priorities of the masters are static or hardwired.
- *Variable priority*: The priorities of the masters can be dynamically changed during system operation by writing to special programmable priority registers

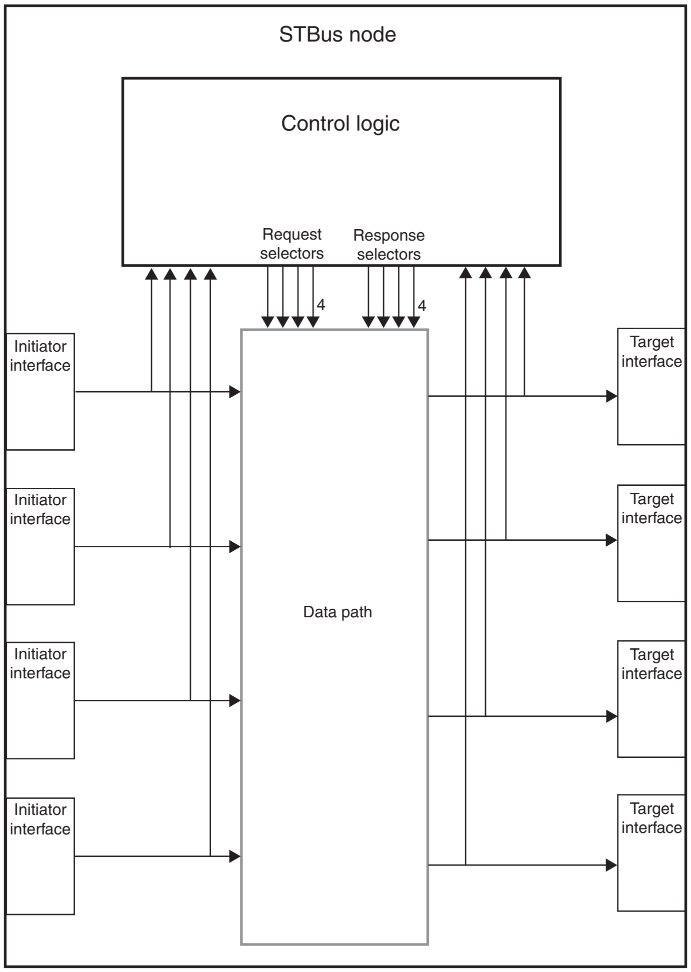

FIGURE 3.27

Block diagram of STBus node [4]
Source: STMicroelectronics

in the node. If two or more masters have the same priority value stored in their registers, the master with the higher fixed (hardwired) priority gets preference.

- *Least recently used (LRU)*: Masters are granted access to the bus in the order of the longest time since the last grant. If two or more masters have been waiting for the same amount of time, the master with the higher fixed (hardwired) priority gets preference.
- *Latency-based*: Each master has a register associated with it containing the maximum allowed latency in clock cycles. If the value is 0, then it needs to have zero cycle latency when a request is received, and such a master must be granted bus access as soon as possible. Each master also has a counter, which is loaded with its maximum latency value each time the master

makes a request. At every subsequent cycle, the counter is decremented. The arbiter grants bus access to the master having the lowest counter value. If two or more masters have the same counter values, the grant will be given to the master with the higher fixed (hardwired) priority.

- *Bandwidth-based*: Each master has a register associated with it containing its bandwidth, expressed in terms of clock cycles per fixed period. Each master also has a counter, loaded with a starting value obtained from its bandwidth register. At the beginning of an operation, the counter for the first master (selected based on the hardwired priority) starts decrementing for the entire duration of the time slot allocated to this master. If during this period the first master makes any requests, they are granted. If during this period no master makes any requests, the first master eventually loses its slot, and the next master based on hardwired priority is selected. However, if during this period the first master makes no requests but requests are received from other masters, its counter is stopped and the bus is granted to the next master in order of hardwired priority (whose counter subsequently begins to decrement since it is now using bandwidth). If a master consumes its bandwidth (i.e., its bandwidth counter has reached 0), it cannot be granted bus access again till the end of the fixed time period. This process continues till all masters have consumed their allocated bandwidth, at which point the arbitration process starts again. This scheme is similar to a TDMA/static-priority two level arbitration scheme.
- *STB*: This is a hybrid of the latency-based and variable priority schemes. In its normal state, arbitration proceeds just like in a variable priority scheme. As in the latency-based scheme, masters also have an associated maximum latency register and counter. Each master also has an additional *latency-counter-enable* bit. If this latency-counter-enable bit is set, and the counter value is 0, then a master is said to be in *panic state*. In the case when one or more masters are in a panic state, the normal variable priority scheme is overridden, and the masters in panic state are granted bus access, in the order of highest priority.
- *Message-based*: This is a fixed priority scheme which allows masters having a higher priority and a priority flag set, to interrupt the message transfer on the bus. Note that in the normal fixed priority scheme, a message transfer cannot be interrupted while it is in progress.

From the description of the supported arbitration schemes in STBus, it can be seen that the control unit requires several registers to hold the latency, bandwidth, and/or priority values for the masters in the system. In addition to the node, other components used in the STBus communication architecture include:

- *Size converters*: Components used to allow communication between two STBus IP blocks having different data bus widths.
- *Type converters*: Components used to allow communication between two STBus IPs following different STBus protocol types (e.g., between Type 1 and Type 3).
- *Buffer*: A FIFO-based component that is used as a retiming stage between two IPs following the Type 2 or Type 3 protocol. A buffer is useful to break

critical paths between components that are far apart from each other on the system floorplan. In this aspect, it is similar to the register slice proposed in AMBA 3.0 AXI.

3.1.5 Sonics SMART Interconnect

The Sonics SMART Interconnect [5] is another on-chip communication architecture standard designed to ease component interoperability and provide high performance for a wide range of applications. The standard comprises of three synchronous bus-based interconnect specifications which differ in their level of complexity and performance: (i) *SonicsMX*, which is a high performance interconnect fabric with advanced features; (ii) *SonicsLX*, which is also a high performance interconnect fabric, but with less advanced features compared to SonicsMX; and (iii) *S3220*, which is a peripheral interconnect, designed to connect slower peripheral components. Both SonicsMX and SonicsLX natively support the open core protocol (OCP) version 2.0 [15] wrapper-based interface standard (described in Section 3.2.1). Additionally, components with AMBA 2.0 AHB and AMBA 3.0 AXI natives interfaces can also be plugged into these interconnection fabrics using pre-designed interface bridge logic components. These features are crucial in maximizing reuse of IP cores. Sonics SMART Interconnect provides a highly configurable communication architecture solution for contemporary SoC designs. It is supported by the *SonicsStudio* development environment [5] that allows automated configuration, data analysis, and performance verification for the communication architecture. We now examine the specifications of the three bus-based interconnect fabrics that make up the Sonics SMART Interconnect, in more detail.

3.1.5.1 *SonicsMX*

SonicsMX is the third generation of *socket-based* (described in Section 3.2) synchronous interconnect fabrics from Sonics, targeted at high performance, sophisticated SoC applications such as WCDMA/3G wireless headsets, video game consoles, and portable multimedia players. Its main features are summarized below:

- Pipelined, non-blocking, and multi-threaded communication support.
- Split/outstanding transactions for high performance.
- Configurable data bus width: 32, 64, or 128 bits.
- Multiple topology support—shared bus, full crossbar, partial crossbar.
- Socket-based connection support, using native OCP 2.0 [15] interface between components and interconnect.
- Bandwidth and latency-based arbitration schemes to obtain desired quality of service (QoS) for components threads.
- Register points (RPs) for pipelining long interconnects and providing timing isolation.
- Protection mode support.
- Advanced error handling support.
- Fine-grained power management support.

SonicsMX supports the full crossbar, partial crossbar, and shared bus interconnection topology. A crossbar configuration can have a maximum size of 8×8, with a maximum of 8 masters connected to 8 slaves. A shared bus configuration can connect up to 32 components with up to 16 masters connected to 16 slaves. *Protocol points (PPs)* are used to interconnect different topology configurations, and consist of logic for frequency and data width conversion. Up to 4 crossbar and shared buses can be joined in a single instance of SonicsMX, to support up to 64 cores. A single *register target (RT)* component is used as interface to the internal configuration registers of a SonicsMX instance, for any dynamic reconfiguration. Multiple SonicsMX instances (interconnected via bridges) can be used in an SoC design. SonicsMX makes use of *RPs* to pipeline long interconnects in order to break long combinatorial timing paths and achieve the desired bus clock frequency of operation. *RPs* are essentially small FIFOs with a depth that is configurable, to provide queuing support if needed. This is similar in concept to register slices in AMBA 3.0 AXI and buffers in STMicroelectronics' STBus.

Initiator (master) and *target* (slave) components are connected to the interconnect fabric (which natively support the OCP 2.0 interface) through *initiator* and *target agents (TAs)*, respectively. An *agent* contains bus logic such as a bridge to connect a component with a mismatched OCP version 1.0 [16], AMBA AHB [1] or AXI [2] interface, *data width converters* to handle mismatched component and interconnect data widths, *flip-flops* to adapt component timing to the interconnect clock frequency, and *RPs*, which provide FIFO-based transaction buffering if required. Figure 3.28 shows an example of a SonicsMX instance that consists of a *crossbar topology (XB)* and a *shared link/bus (SL)* topology interconnected using *PP* connectors. The initiators connect to the architecture via *initiator agents (IAs)* while the targets connect to it via the *TA*. Components with a

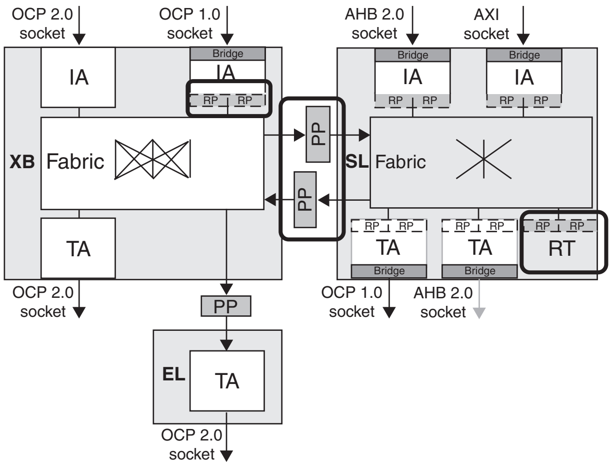

FIGURE 3.28

Example of SonicsMX system with crossbar (*XB*) and shared bus (*SL*) topologies interconnected using *PP* connectors [5]
Source: Sonics Inc.

non-native (i.e., non-OCP 2.0) interface require a bridge at their respective agents. *RPs* provide FIFO buffering/queuing between the components and the interconnect. An *extender link (EL)* is used to connect a target (or initiator) that lies far away from the interconnect fabric on the system floorplan. *ELs* are optimized to span large distances, and use *PP* connectors to interface with the interconnection fabric. An *RT* component is used for interfacing with the internal configuration registers of the SonicsMX instance. Note that all the buffering components in the SonicsMX interconnect fabric can increase the transaction latency, and must be used judiciously. For instance, there is typically at least a one cycle penalty for using flip-flops, *RPs*, and *PPs*.

For multi-threaded components in a system, SonicsMX supports defining mappings between initiator threads and target threads. Resources within the interconnect fabric such as *RPs* and *PPs* are allocated for each of the thread mappings. For cases where independent resources and flow control is not required for every thread (e.g., for multi-threaded initiator components with limited concurrency), thread collapsing at the initiator socket is supported. This can reduce the overhead of allocating unnecessary resources in the interconnect fabric, without degrading performance.

The arbitration schemes used in SonicsMX guarantee QoS requirements for an application. Three QoS levels are defined for use, each characterized by a different arbitration policy:

- *Weighted QoS*: This mode uses a bandwidth-based arbitration scheme, where the available bandwidth is distributed among initiators, based on the ratio of bandwidth weights configured at each initiator agent.
- *Priority QoS*: This mode uses two arbitration schemes. It extends the bandwidth-based arbitration scheme from the weighted QoS mode, by adding support for static priority. One or two threads are allowed to be assigned a static priority instead of bandwidth weights, and always get preference over other threads that are allocated bandwidth weights. These threads that are assigned bandwidth weights may starve for bandwidth, if there is excessive traffic from priority threads. The bandwidth-based weight allocation scheme can therefore be considered to be a kind of *best-effort* bandwidth scheme.
- *Controlled QoS*: This mode uses three arbitration schemes, which dynamically switch among each other based on traffic characteristics. In addition to the priority and best-effort bandwidth scheme from the priority QoS mode, an additional allocated bandwidth scheme is used to ensure that certain threads are guaranteed a certain bandwidth during operation.

For power management, SonicsMX utilizes several mechanisms to reduce idle and active power levels. A fine grained internally implemented clock gating mechanism is used to remove the clock from inactive portions of the interconnect to reduce power consumption. Power management interfaces (signals) at each interface socket allow its activity status to be observed externally, and enable activity-dependent power management (e.g., waking up powered-down targets). Coarse grain power management is supported by using power control logic external to the SonicsMX instance. The external power control logic manages power by removing clock or supply voltage from the entire interconnect. Every SonicsMX instance provides a single power management interface for the entire interconnect, which allows the external power control logic to know when the clock or supply voltage can be restored or removed without disrupting communications.

An optional access protection mechanism is implemented in SonicsMX to designate protected regions within the address spaces of certain targets. This mechanism can dynamically specify protected region sizes and locations at runtime. The mechanism can define access permissions as a function of which initiator can access a protected region, the type of transaction (read or write) being requested, or what state the target is currently in. Each target is allowed to have up to eight protection regions.

3.1.5.2 *SonicsLX*

SonicsLX is a third generation of socket-based synchronous interconnect fabrics from Sonics, targeted at mid-range SoC designs. It supports pipelined, multi-threaded, and non-blocking communication on its buses. It also has support for SPLIT transactions to improve bus utilization. SonicsLX can be arranged in a full or partial crossbar topology and supports the weighted and priority QoS modes, as described above for SonicsMX. The SonicsLX features are a subset of the SonicsMX feature set. Table 3.2 summarizes the main differences between SonicsMX and SonicsLX.

3.1.5.3 *Sonics Synapse 3220*

The Sonics Synapse 3220 synchronous interconnect fabric is targeted at low bandwidth, physically dispersed peripheral target (slave) cores. The main features of the 3220 interconnect are a subset of the SonicsMX and SonicsLX interconnect fabrics. Its main characteristics are summarized below:

- Up to 4 OCP-compliant initiators, and 63 OCP-compliant targets.
- Up to 24-bit configurable address bus.
- Configurable data bus widths—8, 16, 32 bits.
- Fair arbitration scheme, with high priority allowed for a single initiator thread.

Table 3.2 Comparison of SonicsMX and SonicsLX feature set

Features	SonicsMX	SonicsLX
Data Width Conversion	Full	Full
Quality of Service Management	Multi-level QoS	Two level QoS
Advanced Power Management	Configurable	Fixed
Advanced Security Management	Full	Reduced
Interrupt and Error Management	Full	Reduced
Side Band Signaling Management	Full	None

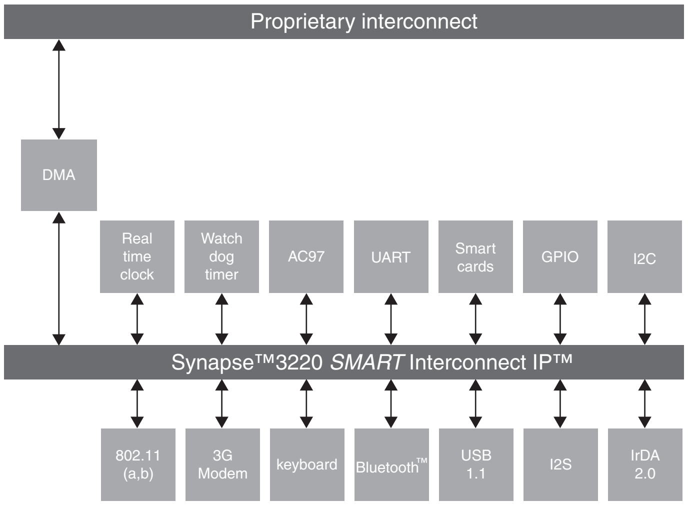

FIGURE 3.29

Example of Synapse 3220 interconnect fabric interfacing with the rest of the system [5]
Source: Sonics Inc.

- Power management interface.
- Exclusive (semaphore) access support.
- Error detection and recovery—watchdog timer to identify unresponsive peripherals.
- Protection mode support.

Figure 3.29 shows some of the typical peripheral targets that are connected to the Synapse 3220 interconnect fabric. The fabric interfaces with the main interconnect fabric (SonicsMX or another proprietary interconnect) using a DMA type block that acts like a bridge.

3.1.6 OpenCores Wishbone

The Wishbone bus-based on-chip communication architecture standard [6] is an open-source standard that proposes a single, high speed synchronous bus specification to connect all the components in an SoC design. Since it is open source, designers can download synthesizable Wishbone RTL components available for free from the OpenCores website [6]. However, due to lack of default support for advanced features (e.g., OO transaction completion, SPLIT transactions, power management, etc.), its scope is limited to small- and mid-range embedded systems. The main features of the Wishbone high speed bus standard are summarized below:

- Multiple master support.
- Up to 64-bit address bus width.
- Configurable data bus width 8- to 64-bit (expandable).
- Supports single or block read/write operations.
- Read–modify–write (RMW) support for semaphore type operations.

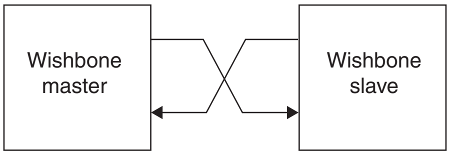

FIGURE 3.30(a)

Different topologies of the Wishbone bus architecture: point-to-point

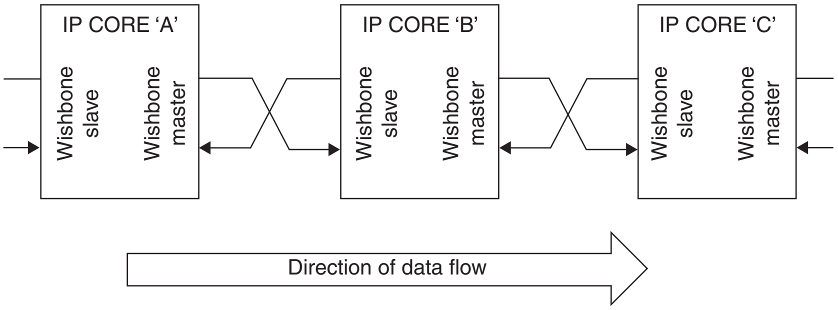

FIGURE 3.30(b)

Data flow

- Supports point-to-point, data flow, shared bus, or crossbar topology.
- Retry mode support.
- User defined tags, for error signaling, cache hints, etc.
- Flexible arbitration scheme (fixed priority, RR, etc.).

The Wishbone interface is highly configurable, and allows a user to customize tags or signals to support specific application requirements. Thus, designers can create their own customized version of the Wishbone standard to suit a particular applications need. While this customizability is desirable, it can make developing generic components (e.g., bridges to interface with other standards such as AMBA) more difficult. Figure 3.30 shows the different possible topologies in which the Wishbone on-chip communication architecture can be structured. For very small systems, the master and slaves can be directly connected using *point-to-point* bus links, as shown in Fig. 3.30(a). For systems with a sequential data flow, where data flows from component to components, the *data flow* topology is recommended, as shown in Fig. 3.30(b). Data flows in a pipelined manner, exploiting parallelism and thus speeding up execution. To interconnect several masters and slaves effectively and with the fewest resources, a *shared bus* topology can be used, as shown in Fig. 3.30(c). The arbitration scheme of the arbiter (not shown in the figure) is left up to the system designer. The shared bus can have either a tri-state-based, or a MUX-based implementation. For higher performance systems, a *crossbar* topology can be used, as shown in Fig. 3.30(d). While this topology offers better performance due to higher parallelism, it requires more logic and

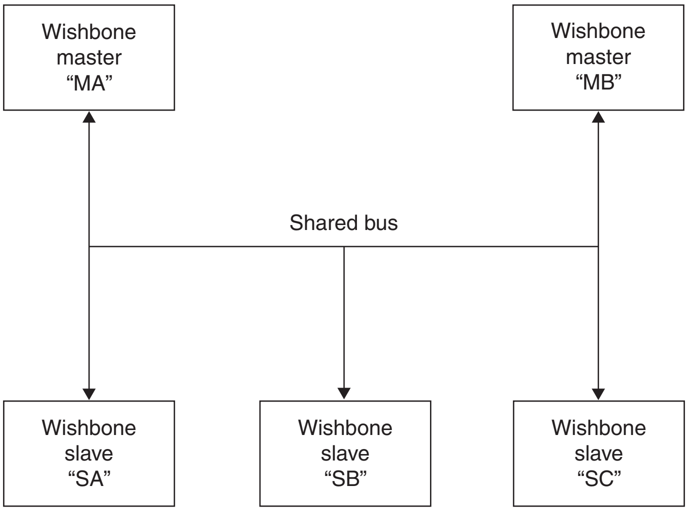

FIGURE 3.30(c)

Shared bus

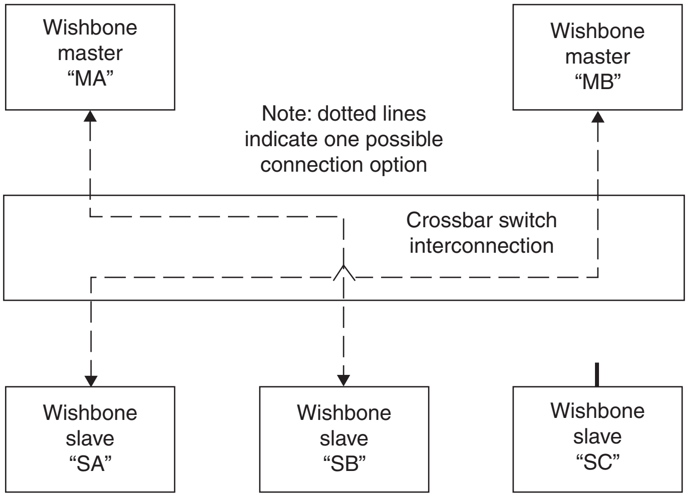

FIGURE 3.30(d)

Full or partial crossbar [6]

routing resources. As a rule of thumb, a crossbar switch with two masters and two slaves takes up twice the interconnect logic as a similar shared bus system with the two masters and two slaves.

Wishbone supports the basic single read/write data transfers with handshaking, and block (burst) read/write data transfers, like the other standards. Semaphore type operations, that allow multiple components to share common resources, are supported via the RMW operation. This type of operation is commonly used in disk controllers, serial ports, and memories. In this operation, once an arbiter grants the bus to a master, no other master is allowed to access the bus till the selected master has read, modified, and written back data to the slave. Such a *locking* of the bus can be inefficient, especially if the slave takes multiple cycles to respond. More efficient semaphore type operation support was discussed earlier in the context of other bus standards such as AMBA 3.0 AXI and STBus Type 3, where the bus need not remain locked for the entire duration of the semaphore operation.

3.1.7 Altera Avalon

The Altera Avalon [7] synchronous bus-based communication architecture is targeted at system-on-programmable-chip (SoPC) designs, and is comprised of two standards: Avalon memory mapped (Avalon-MM) [32] and Avalon streaming (Avalon-ST) [33].

3.1.7.1 *Avalon-MM*

The Avalon-MM standard defines an interface to connect memory-mapped master and slave peripherals such as microprocessors, UARTS, memory, timers, etc. Configurability is an important attribute of the Avalon-MM interface, and components can choose to use a small set of signals if they only support simple transfers (e.g., a simple ROM interface requiring only address, read-data, and control signals). Components requiring more complex transfer types will support a larger set of signals (e.g., high speed memory controller that supports pipelined bursts). The Avalon-MM signals are a superset of several other bus standards. For example, most Wishbone interface signals can be mapped to Avalon-MM signals types, making it easy to include Wishbone components into Avalon-MM systems.

Avalon-MM is implemented as a synchronous bus crossbar, as shown in Fig. 3.31. The crossbar has an integrated interrupt controller, and supports optional logic to transfer data across multiple clock domains, and across multiple interface widths. Pipeline registers can be added at any point on a crossbar bus to increase the value of the maximum allowed clock frequency on the bus. Separate read and write data buses can have widths of up to 128 bits. The crossbar buses support the burst transfer mode to hide arbitration latency for individual transfers, as well as fixed latency and variable latency pipelined reads. For the fixed latency case, with a specified pipeline latency of *N*, the slave must present valid read data on the *N*th rising clock edge after the end of the address phase. Figure 3.32(a) shows an example of pipelined read data transfers with a fixed latency of 2 cycles. In the variable latency case, the slave can take an arbitrary number of cycles after the address phase to put the read data on the bus, as shown in Fig. 3.32(b). An additional signal is used by the slave in this case to signal to the switch fabric when valid data has been put on the read data bus.

Several arbitration schemes are allowed in the Avalon-MM crossbar, such as:

- *Fairness-based shares*: This scheme assigns each master port with an integer value of transfer shares with respect to a slave port. One share signifies permission to perform one transfer. As an example, if *Master 1* is assigned

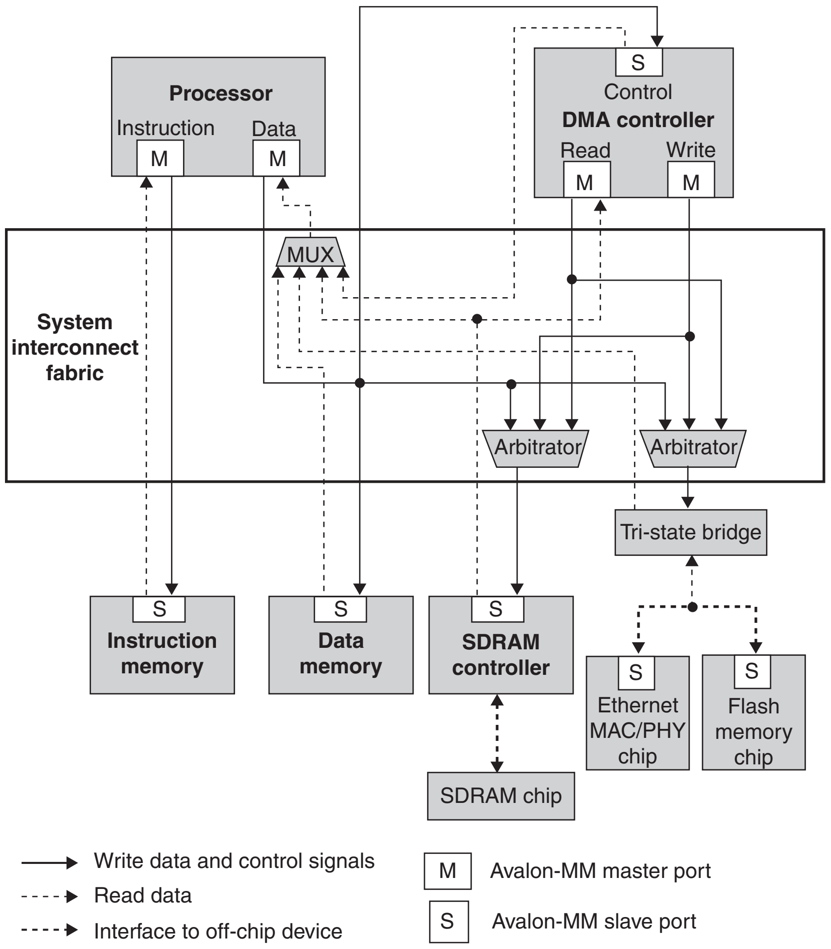

FIGURE 3.31

Avalon-MM crossbar switch implementation example [7]

Altera is a trademark and service mark of Altera Corporation in the United States and other countries. Altera products are the intellectual property of Altera Corporation and are protected by copyright laws and one or more U.S. and foreign patents and patent applications

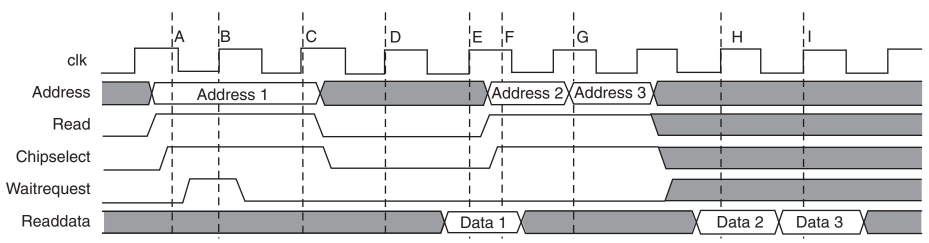

FIGURE 3.32(a)

Pipelined read data transfer in Avalon-MM, with fixed latency = 2 cycles

Altera is a trademark and service mark of Altera Corporation in the United States and other countries. Altera products are the intellectual property of Altera Corporation and are protected by copyright laws and one or more U.S. and foreign patents and patent applications

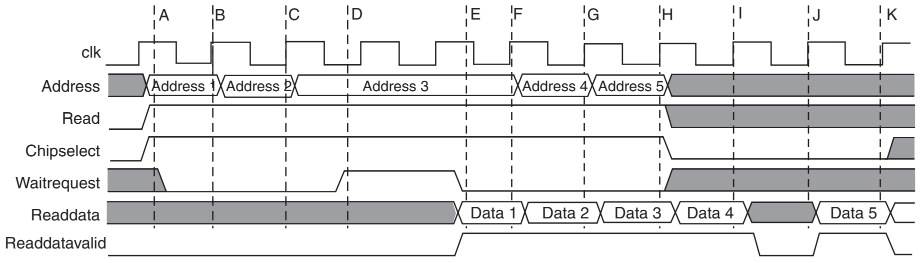

FIGURE 3.32(b)

Variable latency [7]

Altera is a trademark and service mark of Altera Corporation in the United States and other countries. Altera products are the intellectual property of Altera Corporation and are protected by copyright laws and one or more U.S. and foreign patents and patent applications

3 shares and *Master 2* is assigned 4 shares, then the arbiter grants *Master 1* access for 3 transfers, followed by a grant to *Master 2* for 4 transfers. The arbiter cycles through this process indefinitely. If a master stops requesting transfers before it exhausts its shares, it forfeits all of its remaining shares, and the arbiter grants access to another requesting master.

- *RR*: Masters are granted access to a slave in a cyclic, RR manner, ensuring a fair bus access distribution among requesting masters.
- *RR with minimum share values*: In this scheme, a slave port can define a value for the minimum number of shares in each RR cycle. This results in the arbiter granting at least *N* shares to any master port when it begins a sequence of transfers. By declaring a minimum share value *N*, a slave indicates that it is more efficient at handling continuous sequential transfers of length *N*. Since burst transfers provide even higher performance for continuous transfers to sequential addresses, the minimum share value does not apply for slave ports that support burst transfers—the burst length takes precedence over minimum share value.

3.1.7.2 *Avalon-ST*

The Avalon-ST standard defines an interface optimized for the unidirectional flow of data, with support for multiplexed streams, packets, and DSP (digital signal processor) data. Avalon-ST is implemented as a synchronous point-to-point communication bus. The interface signals can be used to describe traditional streaming interfaces that consist of a single stream of data without knowledge of channels or packet boundaries. The interface can also support more complex protocols such as burst and packet transfers with packets interleaved across multiple channels. Packet transfer between the source and destination components is supported by using three interface signals: *startofpacket*, which is used by the source to indicate the cycle with the start of packet; *endofpacket,* which indicates the cycle containing the end of the packet; and the optional *empty* signal, which indicates the number of symbols that are empty during the cycles that mark the end of a packet. At the time of publishing of this book, Avalon-ST is a relatively new standard and its specification document is sparse on advanced features and supported modes.

3.2 SOCKET-BASED ON-CHIP BUS INTERFACE STANDARDS

A socket-based on-chip bus interface standard defines the interface of a component that connects to a bus-based communication architecture. Unlike the bus-based communication architecture standards described in the previous section that define the bus–component interface and the architectural implementation of the bus, socket-based bus interface standards only define the interface and do not address the bus architecture implementation. The computational components are truly decoupled from the communication architecture and its implementation in this scenario. Figure 3.33 illustrates an example of a system utilizing a socket-based bus interface standard. The standard interface definitions allow components to be designed with a standard interface, without committing to a particular communication architecture implementation. This improves IP reuse flexibility, since components can now be connected to any of the wide array of standard bus-based communication architectures described previously. Figure 3.33 shows how designers are free to choose any standard or proprietary bus architecture (e.g., AMBA, CoreConnect, STBus) to implement the actual communication primitives, when using a socket-based interface (I/F) standard. The only requirement for seamless component and bus architecture integration in this case is *adapter* logic components that can map the component interface to the bus architecture fabric protocol. Such adapter or translation logic is not required if the bus architecture implementation natively supports the socket-based interface definition. As an example, since the Sonics SMART Interconnect [5] bus architecture fabric natively supports the signals in the OCP 2.0 [15] socket-based interface standard, no translation logic is required between the bus and components that have an OCP 2.0 socket interface.

Socket-based interface standards must be generic, comprehensive, and configurable to capture the basic functionality and advanced features supported by a

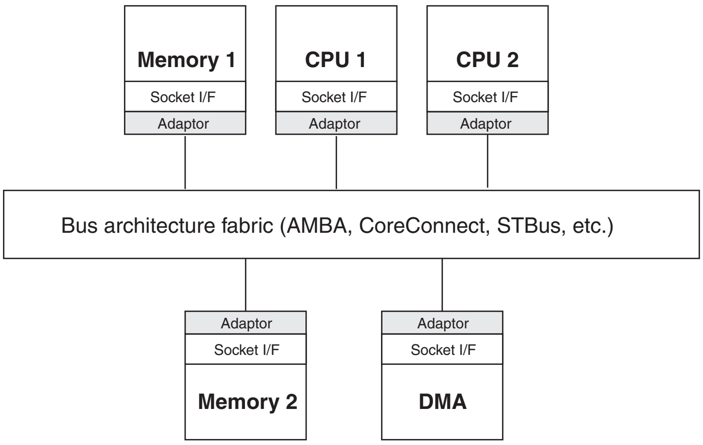

FIGURE 3.33

Example of system implemented with socket-based interface standards wide array of standard bus-based communication architectures. This ensures that there are no possible incompatibilities when a component with a socket-based interface is connected to the bus-based communication architecture that actually implements the communication primitives. Socket-based interface standards have the disadvantage of requiring adapter or translation logic at the bus–component interface. This results in additional design time to create the adapter logic, an increase in chip area, and possible performance penalties due to every data transfer propagating through the adapter logic that has an intrinsic delay associated with it. However, the adapter logic only needs to be created once and can be subsequently reused in several designs. The benefits that accrue from using socket-based bus interface standards such as improved IP reusability across designs and greater flexibility to explore (or change) diverse bus architecture implementations should not be underestimated.

Several socket-based bus interface standards have been proposed over the years. Some of the popular interface standards include OCP [15], virtual component interface (VCI) [14], and device transaction level (DTL) [17]. Because of the implementation flexibility it offers, AMBA 3.0 AXI [2] can also be considered to be a type of bus interface standard. OCP, however, is by far the most popular industry standard as far as socket-based bus interface specifications go. Because the VCI interface standard is a subset of the OCP, and DTL is a proprietary standard with very little publicly available information, we will only briefly review these two interface standards. We now present a comprehensive overview of the OCP socket-based interface standard.

3.2.1 Open Core Protocol

The OCP version 2.0 [15] socket-based bus interface standard defines a high performance, synchronous, bus architecture-independent interface between IP cores. It promotes IP design reuse by allowing IP cores to be independent of the architecture and design of the systems in which they are used. It is also highly configurable and can be optimized to use only the necessary features required for communicating between two components, which saves chip area. OCP essentially defines a point-to-point interface between two components, one of which must be a master and the other a slave. Its key features include:

- Point-to-point synchronous interface.
- Bus architecture independent.
- Configurable data flow (address, data, control) signals for area-efficient implementation.
- Configurable sideband signals to support additional communication re-quirements.
- Pipelined transfer support.
- Burst transfer support.
- OO transaction completion support.
- Multiple threads.

Figure 3.34 shows an example of a simple SoC system consisting of three IP cores with OCP interfaces connected to an on-chip bus. The on-chip bus can

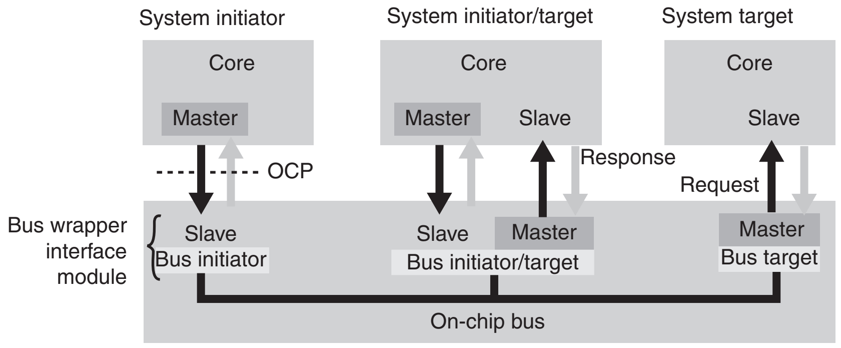

FIGURE 3.34

Example of system implemented with OCP socket-based interface standard [15]
Source: OCP-IP

belong to any one of the standard bus-based on chip communication architectures such as AMBA 2.0/3.0, CoreConnect or STBus, described in Section 3.1. A bus wrapper interface module is required to translate and map the OCP interface signals of the IP cores, to the signals of the on-chip bus. The wrapper interface module must act as the complementary side of the point-to-point OCP connection for each IP core port that is connected to the bus that is, for the case of a master port on the IP core, the connected wrapper module must act as a slave, and for a slave port on the IP core, the connected wrapper module must act as a master (as shown in Fig. 3.34). A data transfer in such a system proceeds as follows: the master (initiator) sends address, data and control information via the OCP interface signals to its corresponding slave (target) bus wrapper interface module. The interface module converts the OCP request to an on-chip bus request, which is then transmitted to the destination. The request is received by the wrapper interface module at the destination, and converted from an on-chip bus request to an OCP request. This OCP request is then transferred from the (master) wrapper interface module to the (slave) destination, which takes the appropriate action.

3.2.1.1 *OCP Signals*

The OCP interface is synchronous, with a single clock signal. Thus all its signals are driven with respect to, and sampled by the rising edge of the clock. The OCP interface signals are divided into three categories: data flow, sideband, and test signals. A small subset of the data flow signals are required to be supported by all OCP interfaces. The remaining data flow signals, as well as all the sideband and test signals are optional. With the exception of the clock, all OCP signals are unidirectional and point-to-point.

Data flow signals consist of a set of signals, some of which are used for data transfers, while others are configured to support any additional communication requirements between the master and slave components. Data flow signals can be divided into the following categories:

- *Basic signals*: These include the clock, address (separate), read and write data, transfer type, and handshaking/response signals between the master and the slave. Only the clock and transfer type signals are mandatory for an OCP interface, the remaining signals being optional. The widths of the address, read data, and write data are configurable, and not limited to being multiples of eight. The transfer type indicates the type of data transfer operation issued by a thread running on a master, and can be any one of the following:

— *Read*: Reads data from the addressed location in a slave.
— *Write*: Writes data to the addressed location in a slave.
— *Idle*: No operation is required to be performed.
— *Broadcast*: Writes data to the addressed location, which may be mapped to more than one slave.
— *Exclusive read*: Reads from a location in a slave and locks it, preventing other masters from writing to the location (exclusive access). The location is unlocked after a write to it from the original master that caused the lock to be set.
— *Linked read*: Reads data from the addressed location in a slave, and sets a reservation in a monitor for the corresponding thread, for the addressed location. Read or write requests from other masters to the reserved location are not blocked from proceeding, but may clear the reservation.
— *Non-posted write*: Writes data to the addressed location in a slave, unlocking the location if it was locked by an exclusive read, and clearing any reservations set by other threads.
— *Conditional write*: Only writes to the addressed location in a slave if a reservation is set for the corresponding thread. Also clears all reservations on the location. If no reservation is present for the corresponding thread, no write is performed, no reservations are cleared, and a FAIL response is returned.

The handshaking signals are used by the master and the slave to synchronize data transfers, and the response signals are used by the slave to signal whether a request is valid or if an error occurred.

- *Simple extensions*: These include signals to indicate the address region (e.g., register or memory), BEs for partial transfers and core-specific configurable signals that send additional information with the transfer request, read data, write data, and the response from the slave. The configurable signals can transmit information about data byte parity, error correction code values, FIFO full or empty status, and cacheable storage attributes.
- *Burst extensions*: These signals are used to support burst transfers. They specify details about the data burst, such as
  — whether it is a precise length or unknown length burst;
  — burst length (for a precise length burst);
  — wrapping, incrementing, exclusive-OR (used by some processors for critical-word first cache line fill from wide and slow memory systems) or streaming (fixed address) burst mode;
  — packing or non-packing mode – for the scenario where data is transferred between OCP interfaces having different widths, the packing mode aggregates data when translating from a narrow to wide OCP interface, while the non-packing mode performs stripping when translating from a wide to narrow OCP interface;

— minimum number of transfers to be kept together as an atomic unit when interleaving requests from different masters onto a single thread at the slave;
— whether a single request is sufficient for multiple data transfers, or if a request needs to be sent for every data transfer in the burst;
— the last request, write data, or response in a burst.

- *Tag extensions*: These signals are used to assign tags (or IDs) to OCP transfers to enable OO responses and to indicate which transfers should be processed in order.
- *Thread extensions*: These signals are used to assign IDs to threads in the master and slave, and for a component to indicate which threads are busy and unable to accept any new requests or responses.

Sideband signals are optional OCP signals that are not part of the data flow phases, and can change independent of the request/response flow (but are still synchronous to the rising edge of the clock). These signals are used to transmit control information such as interrupts, resets, errors, and other component-specific information. They are also used to exchange status and control information between a component and the rest of the system. Finally, the OCP *Test* signals are also a set of optional signals, and are responsible for supporting scan, clock control, and IEEE 1149.1 (JTAG), for testing purposes.

Some of the OCP signals can be grouped together because they must be active at the same time. The OCP data flow signals can be combined into three *groups* of request, response, and data handshake signals. These groups in turn map one-on-one onto their respective protocol *phases*—request, response, and data handshake phases. An OCP *transfer* consists of several phases, and different types of transfers are made up of different combinations of phases. Every transfer must have a request phase. Read type requests always have a response phase, but write type transfers can be configured to with or without the response and data handshake phases. Generally, in an OCP read or write transfer, the request phase must precede the data handshake phase which in turn must precede the response phase. Burst *transactions* are comprised of a set of transfers linked together, and having a defined address sequence and number of transfers. This hierarchy of elements that are part of OCP is summarized in Fig. 3.35.

3.2.1.2 *OCP Profiles*

As mentioned earlier, the OCP interface can be configured to meet the requirements of the communicating components. The OCP 2.0 specification [15] introduces several pre-defined *profiles* that define a configuration of an OCP interface. These profiles consist of OCP interface signals, specific protocol features, and application guidelines. Two sets of profiles are provided:

1. *Native OCP profiles*: These profiles are meant for new components implementing native OCP interfaces. There are three profiles defined in this set:
   - *Block data flow profile*: Master type (read-only, read–write, or write-only) interface for components that require exchanging data blocks with memory.

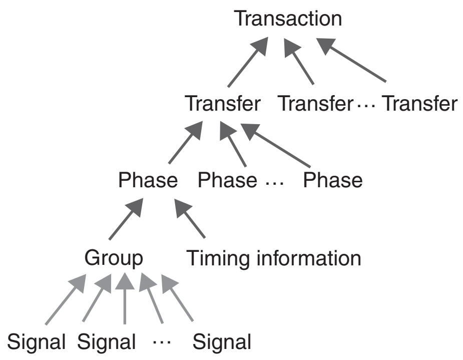

FIGURE 3.35

Hierarchy of elements that compose the OCP [15]
Source: OCP-IP

This profile is useful for managing pipelined access of defined-length traffic (e.g., MPEG macro-blocks) to and from memory.

- *Sequential undefined length data flow profile*: Master type (read-only, read–write, or write-only) interface for cores that communicate data streams with memory.
- *Register access profile*: Supports programmable register interfaces across a wide range of IP cores, such as simple peripherals, DMA, or register-controlled processing engines. Offers a control processor the ability to program the functionality of an attached component.

2. *Bridging profiles*: These profiles are meant to simplify or automate the creation of bridges between OCP and other bus protocol interfaces. There are three profiles in this set:
   - *Simple H-bus profile*: Intended to provide a connection through an external bridge to a CPU with a non-OCP interface (e.g., ARM9 processor [34] with AMBA 2.0 AHB native interface). This profile thus allows creation of OCP master wrappers to native interfaces of simple CPU type masters with multiple-request/multiple-data, read and write transactions.
   - *X-bus packet write profile*: Supports cacheable and non-cacheable instruction and data write traffic between a CPU and the memories and register interfaces of other slaves. Allows creation of OCP master wrappers to native interfaces of CPU type masters (e.g., ARM11 CPU master with AMBA 3.0 AXI native interface) with single-request/multiple-data, write-only transactions.
   - *X-bus packet read profile*: Supports cacheable and non-cacheable instruction and data read traffic between a CPU and the memories and register interfaces of other slaves. Allows creation of OCP master wrappers for native interfaces of CPU type masters (e.g., ARM11 CPU [35] master with AMBA 3.0 AXI native interface) with single-request multiple-data, read-only transactions.

These profiles are useful in several ways such as simplifying the task of integrating OCP components from different vendors, reducing the learning curve when applying OCP for standard purposes, simplifying logic needed to bridge an OCP component with another communication interface standard, improving component maintenance, and easing test bench creation. Figure 3.36 shows an example of an SoC design using the two types of profiles: the CPU and the CPU subsystem make use of the *bridging* profiles, whereas the rest of the components use the *native* OCP profiles. The *X-bus packet read* and *X-bus packet write* profiles used by the CPU (that might internally support a native AMBA 3.0 AXI interface, for instance) support cacheable and non-cacheable instruction and data traffic between the memories and register interfaces of other slaves. The CPU bus subsystem (which might consist of a native AMBA 2.0 AHB interface, for instance) connects to the OCP-based interconnect using the *H-bus profile*, through an external bridge. The MPEG2 decoder component uses multiple OCP interfaces. It has two OCP master interfaces that make use of the *block data flow profile* that is suitable for managing pipelined access of defined-length traffic (e.g., MPEG macro-blocks) to and from memory. The reason for using two master interfaces is to improve parallelism and achieve higher performance. The decoder also has an OCP slave interface that uses the *register access profile*, to allow the CPU to program its operation. The DMA and media controllers also have OCP slave interfaces that use the *register access profile* for the same purpose. The DMA controller has an OCP master interface that can use either a *block data flow profile* or an *OO system interface profile* (TBD—to be defined in future revisions [15]) depending on the amount of parallelism required. The media controller has an OCP master interface that uses the *sequential undefined length data flow profile* which is a good fit for the controller because it needs to communicate a data stream with a memory-based buffer. Finally, the shared synchronous dynamic random access memory (SDRAM) controller optimizes bank and page accesses to SDRAM and can maximize performance (and minimize latency) by reordering requests. Therefore its slave OCP interface uses the *OO memory interface profile* (TBD—to be defined in future revisions [15]).

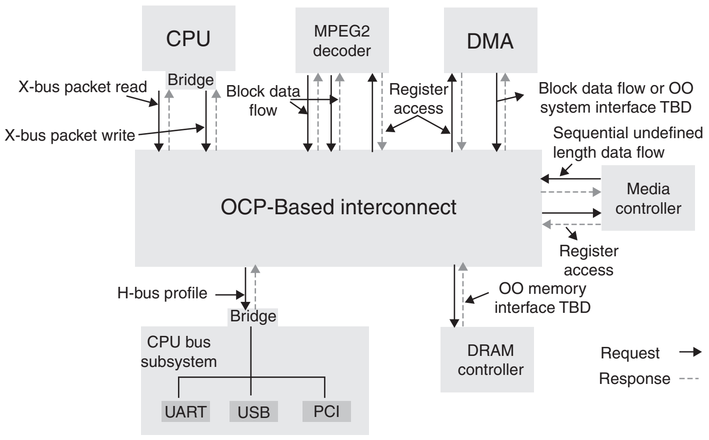

FIGURE 3.36

Example of SoC using several OCP profiles [15]
Source: OCP-IP

3.2.2 VSIA Virtual Component Interface

The virtual socket interface alliance (VSIA) VCI [14] is another point-to-point, synchronous, socket-based bus interface standard. It defines three types of interfaces having varying levels of complexity:

1. *Peripheral VCI (PVCI)*: Defines a simple handshake interface for data transfers, with support for burst transfers, address bus widths up to 64 bits, and data bus widths up to 32 bits.
2. *Basic VCI (BVCI)*: Superset of PVCI; adds support for SPLIT transactions, additional burst modes (e.g., wrapped, fixed), additional data transfer modes (e.g., locked/exclusive read), and data bus widths up to 128 bits.
3. *Advanced VCI (AVCI)*: Superset of BVCI; adds additional data transfer modes (e.g., new wrap, defined transfer modes), and support for advanced features such as OO transaction completion and multiple threads.

As can be seen, each of the interfaces described above is an enhanced and enriched version of the previous one. The interfaces proposed by the AMBA APB, AHB, and AXI bus architecture standards can be considered as somewhat analogous to the Peripheral, Basic, and AVCI interfaces, respectively. VCI actually shares many parallels with the OCP 2.0 socket-based interface standard. Unlike OCP 2.0, however, the VCI interface only contains data flow signals and does not address issues pertaining to test and control. Since the data flow signals in the VCI interface are quite similar to the OCP 2.0 interface signals (although AVCI has some additional features not found in OCP 2.0; however these have yet to be tested in silicon or verified in any form), OCP 2.0 is generally considered a functional superset of VCI.

3.2.3 Philips Device Transaction Level Protocol

The Philips DTL standard [17] defines another point-to-point, synchronous data transfer protocol. The DTL interface supports all the basic signals for single and block (or burst) data transfers, error signaling, and subword operations. Additionally, an extended DTL interface protocol specification defines optional application (or domain) specific extensions to the basic protocol. These extensions include signals for:

- *Addressing modes*: Wrapped, fixed, or decrementing addresses for block transfers.
- *2-D block operations*: Useful when operating on data stored in memory that represents a large 2-D area such as a frame buffer, and smaller 2-D accesses (such as a JPEG 8×8 block) are required.
- *Secure operations*: To indicate if a particular transaction is secure or not.
- *Buffer management*: To allow a component to request a flush of a write buffer, or to request notification when a certain data element reaches its destination.

Four major applications or OCP-like profiles related to traffic types are supported by DTL. Each of these four profiles has requirements for signals that must be implemented at the component interfaces. These profiles are presented below:

1. *Memory-mapped input/output (MMIO)*: For low bandwidth and latency critical control traffic.
2. *Memory-mapped block data (MMBD)*: For moving a block of data between a CPU (or any other component) and memory. Examples include cache line fills or cache line write-back on a CPU. This type of traffic may be both bandwidth and latency critical.
3. *Memory-mapped streaming data (MMSD)*: For moving a sequence of data items between components and memory. Such traffic is usually bandwidth critical, while latency may be less important.
4. *Peer-to-peer streaming data (PPSD)*: For moving a sequence of data items between two components. Like with the MMSD case, such traffic is usually bandwidth critical, while latency may be less important.

DTL is a proprietary interface standard (unlike OCP 2.0 and VCI which are open standards) developed by Philips, and has been used in the popular Philips Nexperia platform [29]. Although currently support for some of the more advanced features such as SPLIT or OO transaction completion, semaphore type operations, cache coherency, and read buffer management is not present, it is planned to be incorporated in future revisions of the DTL specification [17].

3.3 DISCUSSION: OFF-CHIP BUS ARCHITECTURE STANDARDS

While the focus of this book is on on-chip bus-based communication architectures, we include a brief discussion of off-chip bus architecture standards here for the sake of completeness. Off-chip buses are used to connect an SoC with external components, the most common of which are off-chip DRAM memory components such as synchronous dynamic random access memory (SDRAM), dual data rate DRAM (DDR DRAM), and Rambus DRAM (RDRAM). While on-chip embedded DRAM is beginning to become more ubiquitous in SoCs, it is more expensive. Applications with large memory requirements still rely on cheaper off-chip memories to meet storage requirements. An important motivation for the design of off-chip interconnects to connect an SoC with off-chip memory is the need to reduce pin counts, because a large number of pins can significantly increase packaging costs and system complexity. Therefore, unlike on-chip buses that make use of unidirectional multiplexed or AND–OR implementations, off-chip buses prefer bidirectional tri-state implementations to reduce pin counts.

Several off-chip, shared bus standards such as S-100 [18], PC-AT [19], Multi-Bus (II) [20], VME [20], PCI [21], and PCI-X (PCI Extended) [22] have been proposed and used in designs. PCI has undoubtedly been the most popular standard in this category, with almost the entire software infrastructure of the computer industry tied to the PCI interconnect model. However, these shared, parallel off-chip interconnects have inherent limitations such as crosstalk, excessive circuit capacitive loading, delays due to capacitive loads, high power dissipation, signal skew effects due to large distances covered, and reliability issues. As performance requirements of applications increase, higher bus clock speeds and shrinking process technology make crosstalk and capacitive delays more significant, and limit the maximum bus clock speed achievable. To alleviate the problems faced by shared parallel interconnects, switched parallel interconnect standards such as HyperTransport [23] and RapidIO [24] have been proposed, which use narrow, point-to-point connections. HyperTransport, for instance, supports narrower widths between 2 and 32 bits, and higher clock frequencies up to 800 MHz. These switched, point-to-point parallel interconnects solve the electrical loading, speed, and reliability issues of shared parallel interconnects. However, crosstalk and signal skew are still a problem. Switched serial interconnect standards such as PCIe (PCI Express) [25] and Infiniband [26] use a single signal for transmission and can achieve very high speeds, without suffering from any crosstalk effects. PCIe is fast becoming one of the most dominant off-chip standards in system design, especially because of its support for legacy PCI infrastructure. The emerging trend of using optical interconnects instead of copper-cabling for chip-to-chip interconnection is also well suited to a serial communication approach. The interested reader is directed to surveys of off-chip communication architectures by Mayhew and Krishnan [27] and Sassone [28] for related discussions on this topic.

3.4 SUMMARY

In this chapter, we presented the prevailing standards for on-chip communication architectures. Standards are essential in order to promote IP reuse and reduce the design time of the increasingly complex SoC designs today. On-chip bus-based communication architecture standards define the interface signals for components, as well as bus logic components such as arbiters, decoders, and bridges that are needed to implement the features of the proposed standard. We looked at some of the popular on-chip bus architecture standards such as ARM's AMBA 2.0 and 3.0, IBM's CoreConnect, STMicroelectronics' STBus, Sonics' SMART Interconnect, OpenCores' Wishbone, and Altera's Avalon. Another set of standards focuses on defining the component interface, but not the architecture implementation (which is left to the designer). These are the socket-based bus interface standards. Since these standards only define the component interface, the designer is free to use either a proprietary, custom bus architecture implementation, or any one of the bus architecture standards described above, such as AMBA 2.0/3.0 or CoreConnect. Socket-based bus interface standards require additional adapter logic to interface components to non-native bus architectures, and this can increase area, cost, and delay. However, the benefits of improved IP reusability across designs and greater flexibility to explore (or change) diverse bus architecture implementations is also substantial. We described some of the popular socket-based bus interface standards such as OCP, VCI, and DTL in this chapter. Finally, we briefly covered popular off-chip buses and standards which are used to connect SoCs to external DRAM memory blocks and other SoCs.

With a background on bus-based communication architectures in the last chapter (Chapter 2), and a description of prevalent communication architecture standards in this chapter, we now proceed to address the important problem of understanding the on-chip communication architecture design space, to aid in selecting the best communication architecture configuration for an application. The next chapter (Chapter 4) presents models for the performance estimation of communication architectures. These models capture details of the communication architecture design space and allow designers to estimate the performance of different communication architecture configurations. The subsequent chapter (Chapter 5) presents models for power estimation of communication architectures that allow designers to know more about the power characteristics of different communication architecture configurations. These performance and power models for communication architectures are used as part of various techniques (presented in Chapter 6) to select, configure and design a communication architecture that meets the requirements of a given application.

Brief Discussion: Evolution of On-Chip Communication Protocols

As the trend for SoCs moves toward multiple processors on a chip, on-chip communication protocols are continuously evolving. Most of the popular communication architecture (e.g., AMBA) and socket-based standards (e.g., OCP) have evolved over the last few years to accommodate the need for high performance and customizable on-chip data communication. Going forward, these standards will likely continue to evolve further, to handle the many needs of multiprocessor SoC (MPSoC) designs. Support for cache coherence mechanisms will be important in the next generation communication protocols, as multiple processors will frequently access shared memories both on and off the chip. The excessive power consumption of complex communication architectures will also necessitate more explicit support for dynamic power management, to switch off parts of the communication architecture fabric when not in use via power/clock gating. Finally, the number of on-chip communication standards has been growing over the past few years, and will possibly continue to grow in the coming years, requiring more emphasis on techniques to handle interface mismatches. Some recent research in the area of handling interface mismatches between different protocols is presented in Chapter 9.

REFERENCES

[1] ARM AMBA Specification and Multi layer AHB Specification (rev2.0), http://www.arm.com, 2001.

[2] ARM AMBA 3.0 AXI Specification, www.arm.com/armtech/AXI.

[3] IBM CoreConnect Specification, http://www.ibm.com/chips/techlib/techlib.nsf/product families/CoreConnect_Bus_Architecture.

[4] "STBus Communication System: Concepts and Definitions," *Reference Guide*, STMicro electronics, May 2003.

[5] Sonics SMART Interconnect, http://www.sonicsinc.com.

[6] Wishbone Specification, http://www.opencores.org/wishbone.

[7] Altera Avalon Interface Specification, April 2006, http://www.altera.com/.

[8] AMBA AHB Interconnection Matrix, www.synopsys.com/products/designware/amba_solutions.html.

[9] S. Pasricha, N. Dutt and M. Ben-Romdhane, "Constraint-driven bus matrix synthesis for MPSoC," *Asia and South Pacific Design Automation Conference (ASPDAC 2006)*, Yokohama, Japan, January 2006, pp. 30–35.

[10] S. Pasricha, Y. Park, F. Kurdahi and N. Dutt, "System-level power-performance trade-offs in bus matrix communication architecture synthesis," *International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS 2006)*, Seoul, Korea, October 2006 pp. 300–305..

[11] S. Pasricha, N. Dutt and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," *Design and Automation Conference (DAC 2004)*, San Diego, CA, June 2004, pp. 113–118.

[12] S. Pasricha, N. Dutt, E. Bozorgzadeh and M. Ben-Romdhane, "FABSYN: Floorplan-aware bus architecture synthesis," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, Vol. 14, No. 2, March 2006, pp. 241–253.

[13] R. Ho, K. W. Mai and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, Vol. 89, 2001.

[14] VSI Alliance™ On-Chip Bus Development Working Group, Virtual Component Interface Standard Version 2 (OCB 2 2.0), April 2001.

[15] Open Core Protocol International Partnership (OCP-IP). OCP Datasheet, Release Version 2.0/2.1, http://www.ocpip.org.

[16] Open Core Protocol International Partnership (OCP-IP). OCP Datasheet, Release Version 1.0, http://www.ocpip.org.

[17] Philips Semiconductors. Device Transaction Level (DTL) Protocol Specification, Version 2.4, February 2005.

[18] "IEEE Standard 696 Interface Devices," IEEE Computer Society, June 1982.

[19] "PC/104 Specification," PC/104 Embedded Consortium, August 2001.

[20] J. Zalewski, *Advanced Multimicroprocessor Bus Architectures*, IEEE Computer Society Press, 1995.

[21] PCI Special Interest Group, "PCI Local Bus Specification, Revision 2.2," December 1998.

[22] PCI Special Interest Group, "PCI-X 2.0 Protocol Specification Revision 2.0," July 2003.

[23] HyperTransport Consortium, "HyperTransport Technology: Simplifying System Design," October 2002, http://www.hypertransport.org.

[24] RapidIO Trade Association, "RapidIO Technical Whitepaper Rev 3," http://www.rapidio.org.

[25] PCI Special Interest Group, "PCI Express Base Specification Revision 1.0a," April 2003.

[26] Infiniband Trade Association, "Infiniband Architecture Specification, Release 1.0," October 2000. http://www.infinibandta.org.

[27] D. Mayhew and V. Krishnan, "PCI express and advanced switching: evolutionary path to building next generation interconnects," in *Proceedings of 11th Symposium on High Performance Interconnects*, 2003, pp. 21–29.

[28] P. Sassone, "Commercial trends in off-chip communication," Technical Report, Georgia Institute of Technology, May 2003.

[29] J.A. de Oliveira and H. van Antwerpen, "The Philips Nexperia digital video platform," *Winning the SoC Revolution*, G. Martin and H. Chang (Eds.), Kluwer Academic Publishers, 2003.

[30] PrimeCell® AXI Configurable Interconnect (PL300) Technical Reference Manual, http://www.arm.com/products/solutions/AXISystemComponents.html.

[31] PrimeCell® AXI Configurable Interconnect (PL301) Technical Reference Manual, http://www.arm.com/products/solutions/AXISystemComponents.html.

[32] Avalon Memory Mapped Interface Specification, May 2007, http://www.altera.com/.

[33] Avalon Streaming Interface Specification, June 2007, http://www.altera.com/.

[34] ARM9 Processor Family, http://www.arm.com/products/CPUs/families/ARM9Family.html.

[35] ARM11 Processor Family, http://www.arm.com/products/CPUs/families/ARM11Family.html.