CHAPTER

2 Basic Concepts of Bus-Based Communication Architectures

Buses are one of the most widely used means of communicating between components in a system-on-a-chip (SoC) design. The simplicity and efficiency of transferring data on buses has ensured that they remain the preferred interconnection mechanism today. A bus connects possibly several components with a single shared channel. The shared channel can be physically implemented as a single wire (i.e., a serial bus or a set of wires) which makes up a parallel bus. This parallel bus is the typical implementation choice for a bus in almost all widely used on-chip bus-based communication architectures. Although a bus is essentially a broadcast medium, in most cases data transmitted on the bus is meant for a particular component and is ignored by the other components. Any data transmitted by a component moves from its output pins to the bus wires and is then received at the input pins of the destination component. The destination component typically sends an acknowledgement back to the transmitting component to indicate if the data was received. A *bus protocol* is used to explicitly define a communication transaction through its temporal (e.g., duration and sequence of messages exchanged) and spatial (e.g., message size) characteristics. The bus protocol also determines which component may access the shared bus if multiple requests to send (or receive) data appear on the bus at the same time. Bus-based communication architectures usually consist of one or more shared buses as well as logic components that implement the details of a particular bus protocol. In this chapter, we review the basic concepts of bus-based communication architectures. Section 2.1 presents the terminology and major components used in describing bus-based communication architectures. In Section 2.2, we discuss the organization and characteristics of buses. Section 2.3 presents different types of data transfers on buses. Section 2.4 outlines the diverse topologies resulting from bus-based communication architecture implementations. Section 2.5 briefly describes issues arising from the physical implementation of bus wires (or signals). Finally, Section 2.6 discusses deep submicron (DSM) effects that are becoming increasingly dominant for on-chip buses with complementary metal-oxide semiconductor (CMOS) technology scaling.

2.1 TERMINOLOGY

We begin by reviewing the basic terminology used to describe bus-based communication architectures, and systems deploying these architectures. Figure 2.1 shows a simple SoC design in which several (computational) components are interconnected using a bus-based communication architecture. Components which initiate and control read and write data transfers are referred to as *masters*. The *Processor* and *DSP* (*digital signal processor*) components in Fig. 2.1 are examples of master components that read/write data from/to other components in the system. Every master component is connected to the bus using a set of signals which are collectively referred to as a *master port*. The components that simply respond to data transfer requests from masters (and cannot initiate transfers themselves) are referred to as *slaves*, and have corresponding *slave ports*. The three memory blocks in Fig. 2.1 are examples of slaves that can handle requests for data read and write from other components (e.g., Processor, DSP), but cannot initiate such transfers themselves. The component ports are actually a part of its *interface* with the bus. An interface can be simple, consisting merely of the set of connecting wires to the bus (i.e., master or slave ports). Or it could be more complex, consisting of buffers, frequency converters, etc. in order to improve communication performance.

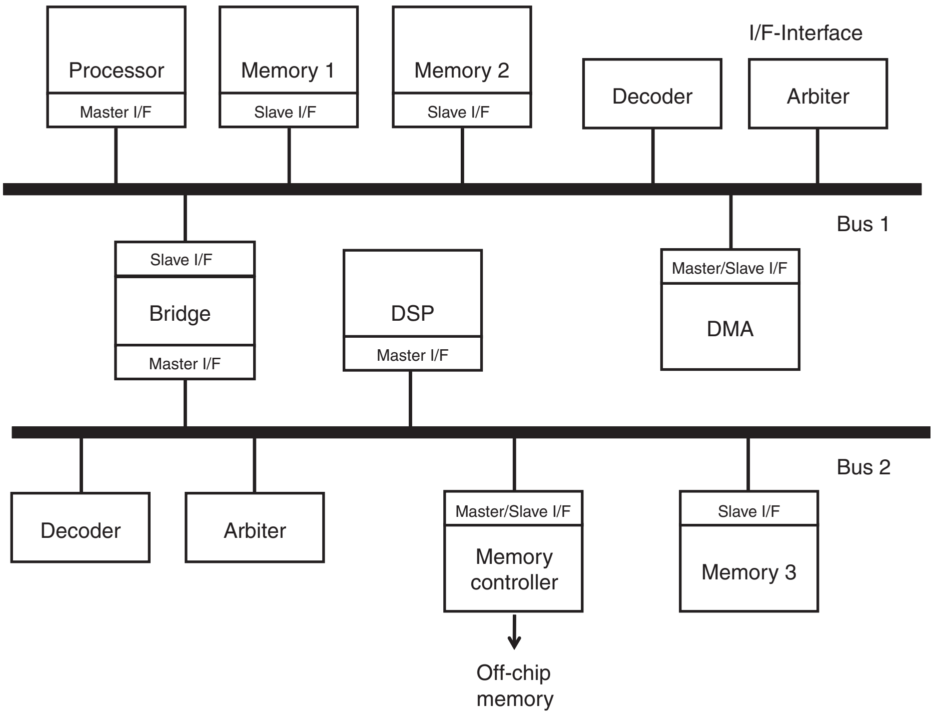

FIGURE 2.1

Example of an SoC with a bus-based communication architecture

Some components can have both master and slave ports, which means that they can act as both masters and slaves. These components are *master/slave hybrid* components. For instance, the *DMA* (*direct memory access*) component in Fig. 2.1 has a slave port that allows the *Processor* to write into (and read from) the *DMA* configuration register file, in order to initialize and configure it. Once configured, the *DMA* component uses its master port to initiate and control data transfers between memory blocks (which would otherwise have been managed by the *Processor*; as a result the *Processor* is freed up to perform other activity which typically improves system performance). Similarly, the *Memory Controller* component has a slave port which is used by the DSP component to initialize and configure its functionality. Once configured, the Memory Controller can initiate and control data transfers with external memory components connected to it, using its master port.

In addition to the wires, a bus-based communication architecture also consists of logic components such as *decoders*, *arbiters*, and *bridges*. A *decoder* is a logic component that decodes the destination address of a data transfer initiated by a master, and selects the appropriate slave to receive the data. It can either be a separate logic component, or integrated into a component interface. An *arbiter* is a logic component that determines which master to grant access to the bus, if multiple masters access the bus simultaneously. Typically, some form of a priority scheme is used, to ensure that critical data transfers in the system are not delayed. Finally, a *bridge* is a logic component that is used to connect two buses. It can have a fairly simple implementation if it connects two buses with the same protocols and clock frequencies. However, if the two buses have different protocols or clock frequencies, some form of protocol or frequency conversion is required in the bridge, which adds to its complexity. A bridge connects to a bus using a master or a slave port, just like any other component. The type of port used to connect to a bus depends on the direction of data transfers passing through it. For instance, in the example shown in Fig. 2.1, the *DMA* and *Processor* components on *Bus 1* initiate and control data transfers to *Bus 2* by sending data to the slave port of the bridge on *Bus 1*, which transfers it to its master port on *Bus 2* and sends the data to its destination. Since the *DSP* and *Memory Controller* do not initiate and control data transfers to components on *Bus 1*, a single bridge is sufficient as shown in Fig. 2.1. However, if these components needed to transfer data to *Bus 1*, another bridge with a slave port on *Bus 2* and a master port on *Bus 1* would be required.

2.2 CHARACTERISTICS OF BUS-BASED COMMUNICATION ARCHITECTURES

Bus-based communication architectures are defined by various architectural and physical characteristics that can have many different implementations. These implementation choices have trade-offs that can significantly affect the power, performance, and occupied area of the communication architecture. In this section, we describe the major characteristics of bus-based communication architectures and discuss some of their common implementation choices.

2.2.1 Bus Signal Types

Bus signals (or wires) are broadly classified into three categories, as shown in Fig. 2.2. *Address* signals are used to transmit the address of the destination for a data transfer on the bus. The number of signals used to transmit the address is typically a power of 2 (common values are 16, 32, or 64) and referred to as the address bus width. However, in some cases, this number can also be an arbitrary value (i.e., not a power of 2), depending on the number of components in a system. The address signals are collectively referred to as the *address bus*. Although most systems have a single shared address bus for both reads and writes, it is possible to have separate address buses for read and write data transfers. Having multiple address buses improves the concurrency in the system, since more data transfers can occur in parallel. However, this comes at the cost of larger number of wires which can increase area and power consumption.

Data signals are used to transmit data values to their destination addresses. The data signals are collectively referred to as the *data bus*. The typical number of signals in a data bus is 16, 32, 64, 128, 256, 512, and 1024 signals (called data bus width). However, this number can vary and have other values depending upon specific requirements of systems. The choice of data bus width is important because it determines whether any packing or unpacking of data is necessary at component interfaces. For instance, consider a case where the memory word size of a memory component is 64 bits and the data bus width is 32 bits. Then, every time a master requests data from the memory, the read data needs to be unpacked (or split) into two data items of 32 bits in width before being transmitted onto the bus. The data also needs to be packed (or merged) at the master interface before being sent to the master component. The packing and unpacking of data at the interfaces introduces an overhead in terms of power, performance, and area of the interface logic. Alternatively, if the data bus width was set to 64 bits, no such packing and unpacking of data would be required. Thus the size of the data bus is typically application specific, and in many cases depends on the memory word size of the memory components used in the system. Much like the address bus, the data buses can either be implemented as a single shared bus for both reads and writes, or separate data buses for reads and writes. Separate data buses improve concurrency and performance in the system, at the overhead of additional bus wire area and power

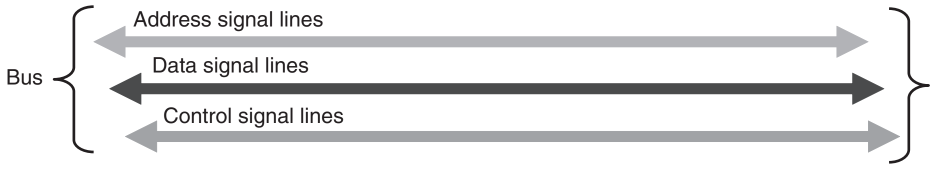

FIGURE 2.2

Classification of bus signals

consumption. It is also possible to combine the data and address buses by multiplexing them over a single set of wires. This may be useful for small, low cost SoCs where high performance communication is not as important as low area footprint, simplified wire routing, and low pin counts at component interfaces.

Control signals are used to send information about the data transfer, and are typically bus protocol specific. *Request* and *acknowledge* signals are the most common control signals, which transmit a data transfer request from a master, and the acknowledgment for a request or data received, respectively. Data size (or *byte enable*) control signals indicate the size of data being transmitted on the bus (particularly useful when the size of the data being transmitted is smaller than the data bus width). Sometimes slaves can signal an error condition to the master over special status control signals, if data cannot be read or written at the slave. Frequently, when multiple data items (called a *data burst*) need to be transmitted on the data bus by a component, there are control signals to indicate the number of data items to the destination. There are control signals that can transmit information about the source of the transmitted data such as a unique ID value identifying the transmitting component. Control signals can also transmit information about the data being transmitted to the destination, such as whether the data is cacheable, bufferable, write-through, or write-back.

2.2.2 Physical Structure

We now look at the structural implementation details of the shared bus signals. Traditionally, shared buses have been implemented using tri-state buffers that drive bidirectional lines, as shown in Fig. 2.3(a). Tri-state implementations of buses are commonly used in off-chip/backplane buses. The advantage of tri-state bidirectional buses is that they take up fewer wires and have a smaller area footprint. However, due to higher power consumption, higher delay (which can limit performance), and problems with debugging tri-state buffers, their use is restricted in modern bus-based on-chip communication architectures. Other efficient (and

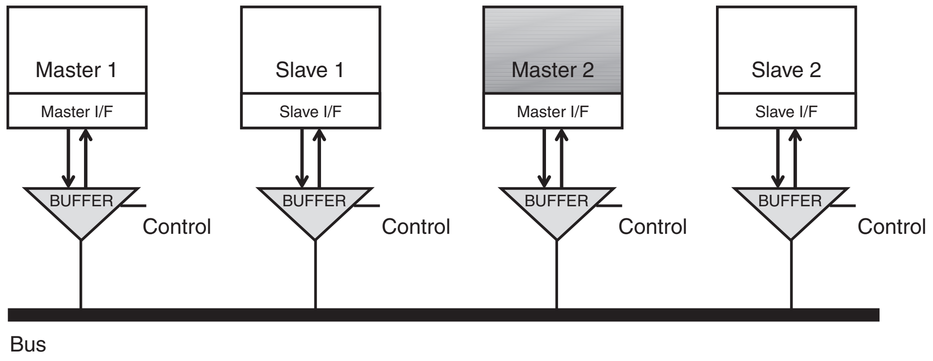

FIGURE 2.3(a)

Shared bus implementation alternatives: tri-state buffer based bidirectional signals

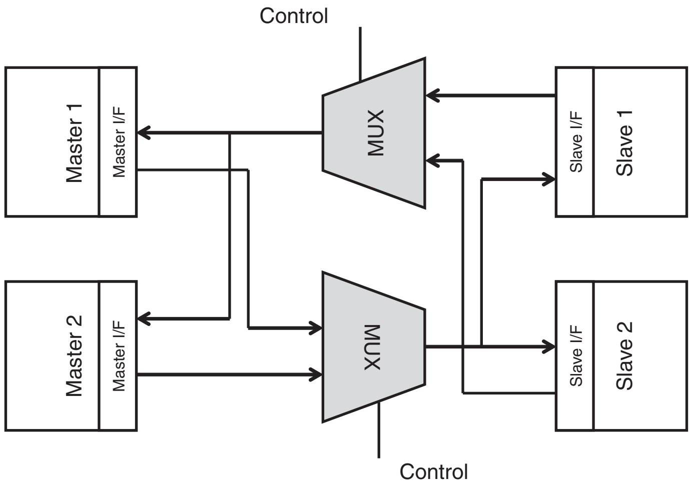

FIGURE 2.3(b)

MUX-based

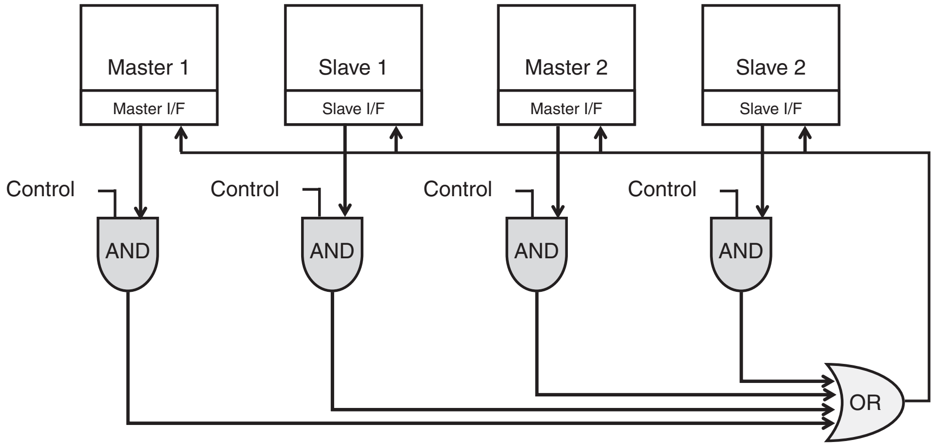

FIGURE 2.3(c)

AND-OR bus

more commonly used) alternatives to the tri-state buffer implementation are a multiplexer (MUX) based implementation, as shown in Fig. 2.3(b) and an AND–OR structure as shown in Fig. 2.3(c).

2.2.3 Clocking

An important characteristic of buses is the type of clocking used for data transfers. A bus that includes a clock signal as one of the control signals is called a *synchronous* bus. Transfers between components usually require some form of handshaking. An example of a synchronous bus is shown in Fig. 2.4(a), where a

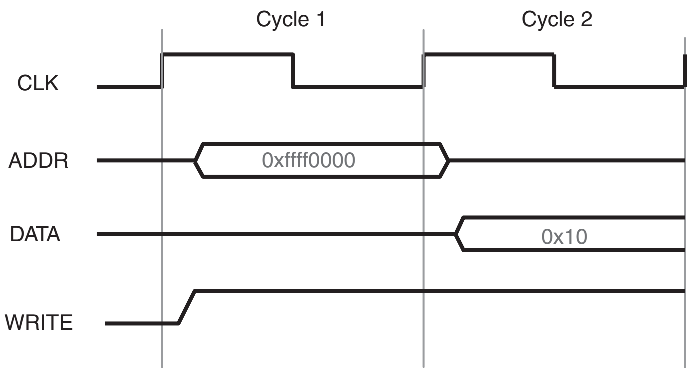

FIGURE 2.4(a)

Clocking strategies for buses: synchronous bus

data item is written by a master and is received by its destination in a total of two clock cycles. The master sends the address (*ADDR*) in the first clock cycle, and asserts the *WRITE* control signal which indicates to the destination that the master wants to write data to it. The data to be written is sent at the beginning of the second clock cycle, and is sampled off the bus by the destination slave at the rising edge of the third clock cycle. The slave can then optionally assert a response control signal (*RESP*) sending the bit value that corresponds to an *OK*, which indicates to the master that there was no error and the write proceeded as intended. The clock signal is essential for synchronization purposes. Synchronous buses enable fast data transfers, but can require frequency converters at component interfaces, since not all components connected to the bus can be expected to run at the same clock frequency as the bus. Indeed, in modern bus-based SoC designs, processors typically run at a clock frequency that is two to four times the frequency of the bus clock. It might also be required to pipeline the synchronous bus by inserting register slices (or buffers) [1] on it, because the signal delay for long wire lengths can actually exceed the bus clock cycle time [2]. In such cases, the register slices allow buffering of data to ensure that the destination can still sample the data as expected at the end of a clock cycle (although not at the end of the same cycle in which the data was transmitted). As an example, consider a synchronous bus clocked at a frequency $f = 100\,\text{MHz}$. A signal on the bus has $1/f = 10\,\text{ns}$ to travel from source to destination. If the signal delay on this bus is 20 ns, then a register slice inserted in the middle of the bus allows the signal to be buffered at the end of the first cycle, and then reach the destination at the end of the next cycle. This is discussed in more detail in Section 2.5. Most standard bus-based communication architectures [1, 3–5] (described in more detail in Chapter 3) use synchronous buses, since they provide high data throughput performance. Synchronous buses are almost always used in the critical path of an SoC design, such as the processor–memory interconnection.

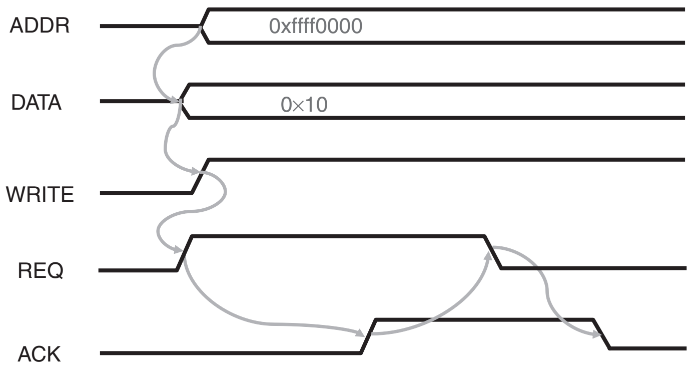

FIGURE 2.4(b)

Asynchronous bus

In an *asynchronous bus*, shown in Fig. 2.4(b), no clock signal is present in the control signals of the bus. In this case, bus synchronization occurs with the help of a handshake protocol that uses request-acknowledgement signals to ensure that the data transfer was completed successfully. For instance, for the example in Fig. 2.4(b) in which a master must write a data to a slave, the address (*ADDR*) and data (*DATA*) for the write data transfer is driven onto the bus by the master, along with a control signal to indicate a write transaction (*WRITE*). The master also asserts the request (*REQ*) control signal. When the destination slave sees the request signal, it samples the data off the bus, and then sends an acknowledgement (*ACK*) signal to inform the master that the data was received. The transmitting master then lowers the request (*REQ*) signal. The handshaking between the source and destination components is usually more extensive in asynchronous buses than in synchronous buses. An example of an asynchronous bus is the MARBLE bus, proposed by Bainbridge and Furber [6]. Asynchronous buses are typically slower than synchronous buses because of the additional overhead of the handshaking protocol, used for synchronization. They also require additional synchronization signals. However, they do not need additional frequency converters like synchronous buses and thus consume less area than synchronous buses. Asynchronous buses also do not suffer from *clock skew* (a phenomenon in which the clock signal from the clock arrives at different components at different times, causing potential timing errors) or the overhead of clock power dissipation, unlike synchronous buses.

2.2.4 Decoding

Whenever data needs to be transferred on a shared bus, the source component transmits the address of the destination component and the data. Each component in an SoC design is typically assigned an address map (i.e., a range of addresses). It is the job of a *decoder* to decode the address and select the appropriate destination component to receive the data. Decoding can be implemented either in a centralized or a distributed manner. Figure 2.5(a) shows a centralized implementation of a decoder. The decoder takes the address of a data transfer issued by a

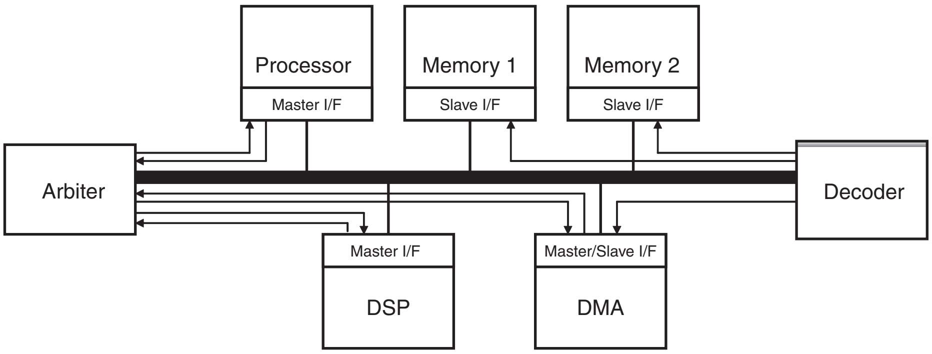

FIGURE 2.5(a)

Different implementation strategies for decoder and arbiter: centralized

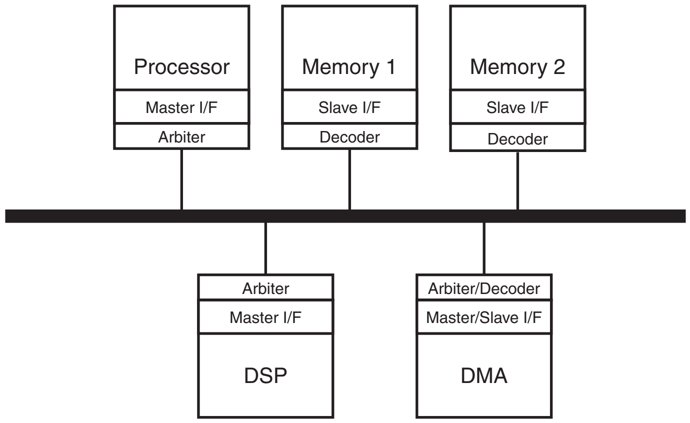

FIGURE 2.5(b)

Distributed

master as its input and then sends a select signal to the appropriate slave component to indicate that data needs to be read or written to the slave. The centralized implementation has the advantage that minimal change is required if new components are added to the system, which makes this scheme easily extensible. Another implementation scheme for the decoder is the distributed decoding approach, shown in Fig. 2.5(b). All the slaves have their own separate decoders in this scheme. When the address is transmitted on the shared bus by the master, the decoders at every slave interface decode the address to determine if the transfer is intended for them. Such a distributed scheme has the advantage of utilizing fewer signals compared to the centralized approach, which needs extra signals to connect the centralized decoder to every slave on the bus. However, there is more hardware duplication in the distributed case, because every slave on the bus now decodes the address, as opposed to a single decoder decoding the address in the centralized scheme. Thus, distributed decoding usually requires more logic and occupies more area. Also, for every new component added to the bus or in case of a change in the address map, changes might be required in the decoders at every slave interface.

2.2.5 Arbitration

It is possible that two or more masters on a shared bus might initiate a data transfer at the same time. Since the shared bus can only handle a single data transfer at any given time, an arbitration mechanism is needed to determine which master gets to proceed with its data transfer, and which has to wait. An *arbiter* is a component on the shared bus that uses certain criteria to determine which master gains access to the bus, if more than one master request access to it simultaneously. The criteria used to determine which master gains access to the bus is called the *arbitration scheme*. Every time a master needs to transfer data, it needs to first request the arbiter to grant it access to the bus. The arbiter uses its arbitration scheme to determine if the master is eligible to get access to the bus, and only when the arbiter signals the master to proceed can the master initiate the data transfer. Like the decoder, an arbiter can be implemented in either a centralized or a distributed manner. Figure 2.5(a) shows the case of an arbiter implemented in a centralized configuration, while Fig. 2.5(b) shows the arbiter in a distributed configuration. The trade-offs for the two schemes are the same as for the decoder. It should be noted that distributed arbitration for the more complex arbitration schemes can be more complicated, and may require synchronization signals between the arbiter logic at the interface of every master on the shared bus.

There are several arbitration schemes that are commonly used in bus-based communication architectures. Some of the basic underlying requirements for an arbitration scheme are that it should guarantee fairness of access, ensure that critical data transfers are completed as soon as possible, and prevent any starvation scenarios (for instance, due to a request from a master never getting access to the bus, which stalls the master execution). One of the most commonly used arbitration schemes is the *static priority* (SP) scheme, in which masters on a bus are assigned fixed priority values. The master having the highest priority always gets access to the bus. The SP scheme can be implemented in a pre-emptive or a non-pre-emptive manner. In a pre-emptive implementation, an ongoing lower priority data transfer is terminated immediately without being completed if a request for bus access is received from a higher priority master. In a non-pre-emptive implementation, the ongoing lower priority data transfer is allowed to complete before the bus is granted to a higher priority master. The SP scheme is simple to implement and can provide high performance by ensuring that critical data transfers, such as between processor and memory, always get higher priority. However, this scheme must be implemented carefully as it can lead to starvation of lower priority masters, which might never be able to get access to the bus if there are frequent bus accesses by higher priority masters. The *round-robin* (RR) arbitration scheme can ensure that there is no starvation in the system. In this scheme, access to the bus is granted in a circular (round-robin) manner, to every master on the bus, which guarantees that every master will eventually get access to the bus. A master relinquishes control over the bus when it no longer has any data to send (or has held the bus for the maximum allowed time) and passes the ownership to the next master in line. The RR scheme is also simple to implement, and can ensure equitable bandwidth distribution on a bus, but suffers from a drawback compared to the SP scheme, in that critical data transfers may have to wait a long time before they can proceed. The *time division multiple access* (TDMA) arbitration scheme is another popular scheme that can guarantee a fixed, higher bus bandwidth to masters with higher data transfer requirements, while also ensuring that lower priority masters do not starve. In this scheme, each master is assigned time slots (or time frames) of varying lengths, depending on the bandwidth requirements of the master. The choice of number of time slots to assign to each master is extremely important. The length of the allocated time frame should be long enough to complete at least a single data transfer, but not so long that other critical data transfers have to wait for a long time to get access.

While the schemes discussed above are essentially single level schemes, more complex arbitration schemes have also been proposed. For instance, it is possible to combine two arbitration schemes to create a two level arbitration scheme: Sonics SMART Interconnect [7] (described in Chapter 3) makes use of a *two level TDMA/RR* arbitration scheme. In this scheme, a TDMA arbitration scheme allocates time slots to various masters. If a master does not have any data to transfer during its time slot, a second level RR scheme selects another master to grant bus access to. Such a scheme thus enables better utilization of the bus, compared to the TDMA scheme, at the cost of a more complex implementation requiring more logic and occupying more area. Another complex, but highly efficient arbitration scheme is the *dynamic priority* (DP) scheme that can dynamically vary the priorities of the masters at runtime (i.e., while the system is executing). Unlike the SP scheme, additional logic is used to analyze data traffic at runtime, and the priorities are dynamically adapted to the changing traffic profiles of an application. Such a scheme can ensure better performance since it can efficiently track changing traffic profiles and ensure that masters that need to send larger amounts of data get higher priority. However, the implementation cost of such a scheme can be high, requiring several registers to keep track of priorities and data traffic profiles at various points during execution. A simpler variant of the DP scheme is the *programmable priority* (PP) scheme, which allows the application to write into the arbiter's programmable registers and set the priority for masters on the bus dynamically.

Since arbiters are invoked for every transfer on the bus, they are considered to be in the critical path of a bus-based communication architecture and must be designed with great care. An arbiter with a complex arbitration scheme implementation, that takes more than one cycle to make a decision, can severely reduce performance. While it might make sense to use a complex multi-cycle arbitration scheme for some applications, in other cases better performance can be achieved by using a simpler, single cycle arbitration scheme. Sometimes pipelining a complex multi-cycle arbiter implementation can also improve performance. These scenarios motivate the need to profile the application early in the design flow, to explore and select an appropriate arbiter implementation. Models for such performance exploration of bus-based communication architectures are presented in more detail in Chapter 4.

2.3 DATA TRANSFER MODES

Typically, data can be transferred over a bus using one of possibly several transfer modes. While some of the basic transfer modes are supported by all standard bus-based communication architectures, other modes are more specialized and specific to certain standard bus protocols. In this section, we review some of the main data transfer modes used in bus-based communication architectures.

2.3.1 Single Non-pipelined Transfer

The simplest form of data transfer on a bus is the single non-pipelined data transfer mode. In this mode, the master first requests access to the bus from the arbiter, and when it is granted access, sends out the address in the next cycle, and then writes data in the subsequent cycle (for a write data transfer) or waits for the slave to send the read data in the subsequent cycle(s). Figure 2.6 shows an example of a master performing two single read data transfers in a sequential manner. The master requests access to the bus from the arbiter at the beginning of the first cycle by asserting the *BUSREQ* control signal. The arbiter grants access to the master in the second cycle by asserting the *GRANT* control signal. Once the master sees that it has been granted access to the bus, it sends out the address (*A1*) of the slave to read data from at the beginning of the third cycle. The slave samples the read request at the beginning of the fourth cycle and then sends back the requested data (*D_A1*) in the same cycle. The read data is sampled off the bus by the master at the beginning of the fifth cycle. To read another data from the slave, the master again requests access to the bus from the arbiter, at the beginning

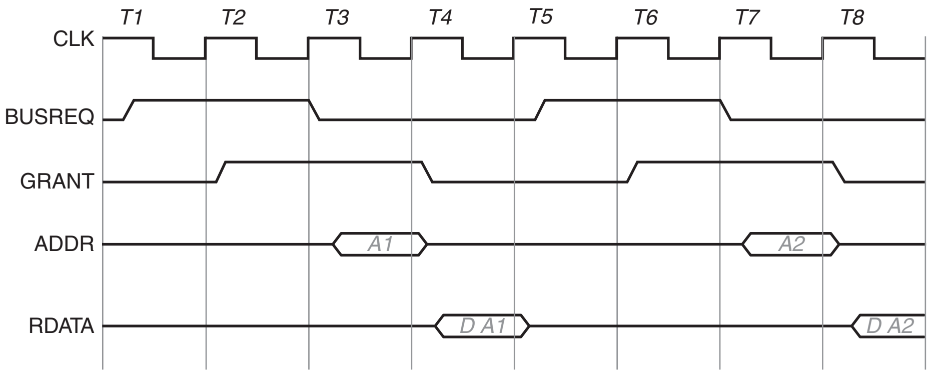

FIGURE 2.6

Single non-pipelined data transfer mode

of the fifth cycle. The sequence of events that follow are similar to that of the first data transfer, and the master samples the data from the slave at the beginning of the ninth clock cycle. Note that unlike in this example, slaves typically take multiple cycles to return data and in some cases even to write data. Also note that in the case where only one master is connected to the bus, arbitration is not needed since there is no possibility of simultaneous bus transfers. In such a case, the first two cycles (bus request and grant) are absent and the data transfer will take only two cycles. Conversely, it is possible that for the case when arbitration is required (i.e., when there are multiple masters connected to the bus), the arbiter takes multiple cycles to decide which master to grant bus access to, as discussed earlier. Such a scenario is possible when the arbiter makes use of a complex arbitration scheme, such as the DP-based one; or for the case when the bus clock frequency is so high that it takes multiple clock cycles for the arbiter to get a response from its chosen arbitration scheme. For the example shown in Fig. 2.6, the single non-pipelined data transfer mode takes as many as four cycles to complete a single read data transfer, under the assumption that arbitration is needed and takes a single cycle. Single non-pipelined transfers, as described in this section, typically occur in bus-based communication architectures with multiplexed address and data buses. We now look at transfer modes that allow us to reduce this number of cycles.

2.3.2 Pipelined Transfer

The pipelined data transfer mode overlaps the address and data phases of multiple data transfers to improve bus performance (i.e., bus throughput). Figure 2.7 shows an example of a pipelined data transfer for two write data transfers initiated by separate masters. At the beginning of the first cycle, both masters (*M1*, *M2*) request access to the bus. The arbiter grants master *M1* access to the bus in the same cycle. Master *M1* then sends the address of its destination slave (*A1*) in the second cycle and the data to write (*D_A1*) in the third cycle. The arbiter grants access to the bus to the second master *M2*, even before its write transfer is finished. This allows *M2* to send the address of its destination slave (*A2*) in the third cycle. Notice that the address phase of the transfer by master *M2* overlaps with the data phase of master *M1*. Finally, master *M2* sends the write data in the fourth cycle, to complete the transfer. Such an overlapped transfer improves bus utilization and reduces the time for a data transfer. Pipelined transfers typically require a more complex arbiter implementation that can perform pipelined (or overlapped) arbitration. Additionally, pipelined transfers are only possible in bus implementations with separate address and data buses (i.e., with no multiplexing of address and data signals).

2.3.3 Burst Transfer

We saw in Fig. 2.6 that multiple data transfers from the same master required arbitration for every individual data transfer. The burst transfer mode improves bus performance by requesting arbitration only once for multiple data transfers.

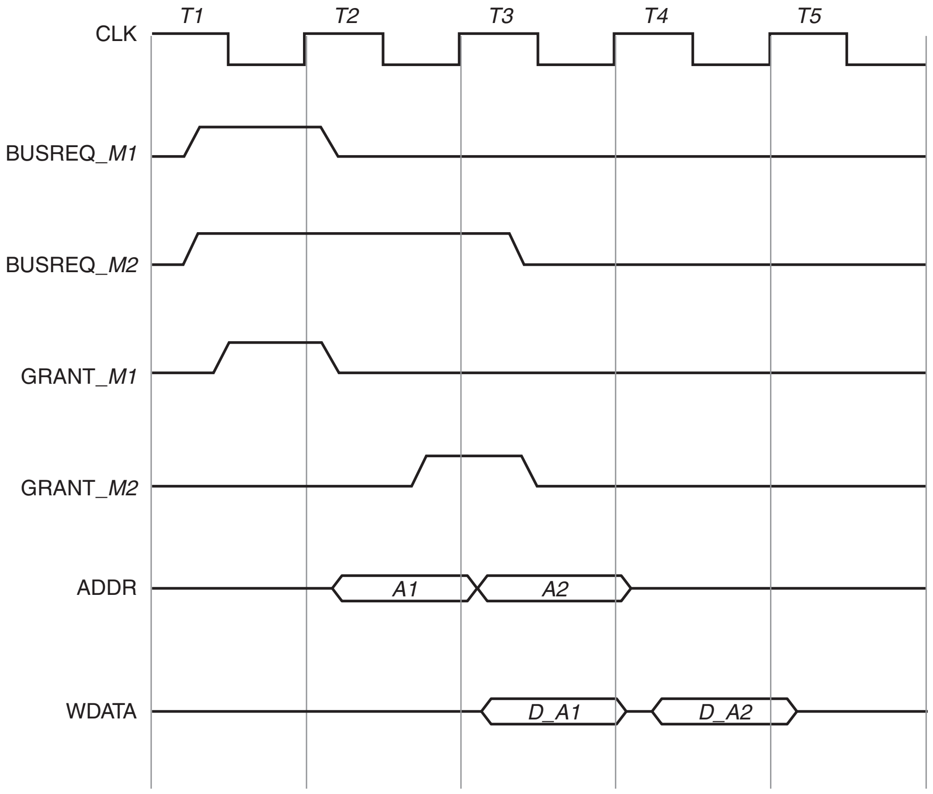

FIGURE 2.7

Pipelined data transfer mode

Figure 2.8(a) shows an example of a non-pipelined, burst data transfer by a master. The scenario depicted has a master needing to write four data items to a slave on the bus. At the beginning of the first cycle, a master requests access to the bus for a "burst" of four data items, and is granted the access by the arbiter at the beginning of the second cycle. Typically, control signals (not shown in the figure) from the master inform the arbiter of the length of the burst (four in this case). The master then proceeds to send the address of the first data (*A1*) item in the third cycle, and then the data to write to the slave (*D_A1*) in the fourth cycle. Since the arbiter has already granted bus access to the master for a burst of four data items, re-arbitration at this point is not required, and the master simply proceeds to send the address of the next data item (*A2*) in the burst at the beginning of the fifth cycle. The data transfer continues till all four data items have been sent to the slave. As can be seen from Fig. 2.8(a), the overhead of arbitration for each data item sent by the master is avoided in a burst transfer, which significantly reduces data transfer time, compared to the single transfer mode shown in Fig. 2.6. Performance can be improved even further if pipelining is allowed within the burst transfer. Figure 2.8(b) shows the same case as Fig. 2.8(a) where a master

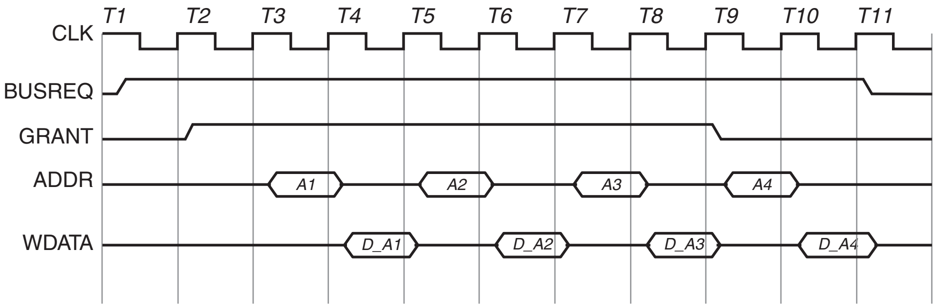

FIGURE 2.8(a)

Example of master writing four data items in burst transfer mode: non-pipelined burst transfer mode

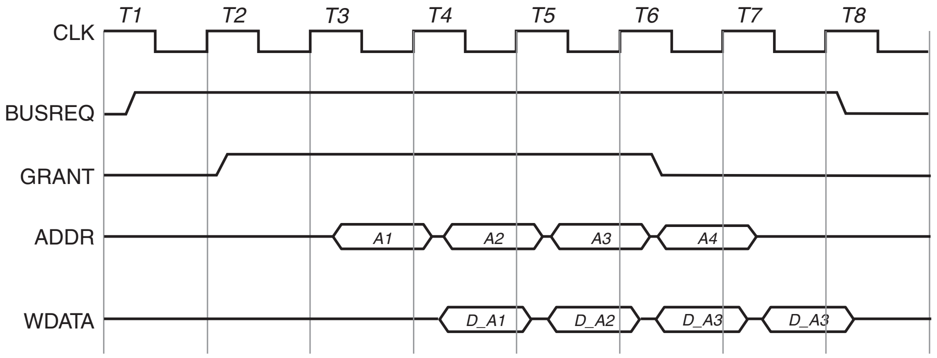

FIGURE 2.8(b)

Pipelined burst transfer mode

sends four data items to a slave, but this time the address and data phases of the data transfers within the burst are overlapped. This pipelined burst mode reduces the data transfer time compared to the non-pipelined burst mode in Fig. 2.8(a), and thus considerably improves bus utilization and performance.

2.3.4 Split Transfer

During a data transfer on a bus, it is possible that a slave can take multiple cycles to return the requested data or write the data. Since the bus in such a case is typically held by a master and no other master can gain access to it till the transfer is completed, the bus remains idle for multiple cycles till the slave completes the transfer. Such a scenario results in under-utilization of the bus and reduces performance. A *split transfer* [4] is a special type of transfer mode that can improve bus utilization in such cases by "splitting" the data transfer and allowing the idle cycles that would otherwise be spent waiting for the slave to be utilized for data transfers by other masters. A split transfer mode typically works as follows. When a data transfer occurs, a slave can decide to issue a SPLIT response to the arbiter if it believes that the transfer will take a large number of cycles to perform. Once the arbiter receives a SPLIT signal from a slave, it masks the request from the master that initiated the transfer (preventing the master from getting any further access to the bus) and uses its arbitration scheme to grant bus access to one of the other masters potentially waiting to initiate data transfers on the bus. Later, when the slave is ready to complete the transfer, it signals the arbiter to "un-split" the master. The arbiter un-masks the request from the master, and in due time the master gets access to the bus again. The slave can then finally complete the transfer. The split transfer mode thus allows the idle cycles in a data transfer to be utilized for other data transfers, and is therefore an effective mechanism for improving the communication performance in bus-based systems. Of course a prerequisite for using this transfer mode is the presence of split capable slaves and arbiters.

2.3.5 Out-of-Order Transfer

An extension of the SPLIT transfer mode described above is to allow multiple transfers from different masters, or even from the same master, to be SPLIT by a slave and be in progress simultaneously on a single bus. This is the basic idea behind *out-of-order (OO)* data transfers [1]. In this mode, masters can initiate data transfers without waiting for earlier data transfers to complete, which improves system performance because multiple data transfers can be processed in parallel. Each data transfer has an ID associated with it, and can complete in any order. This implies that even if a master issues two data transfers, in a sequential manner, it is possible for the second data transfer to complete before the first one. Data transfers having the same ID must complete in the same order in which the master issued them. However, data transfers originating from different masters or from the same master but having different IDs have no ordering restrictions and can complete in any order. This ability to complete data transfers out of order means that data transfers to faster memory blocks can be completed without waiting for earlier transfers to slower memory blocks. As a result, bus utilization and overall system performance are improved significantly.

Predictably, there is overhead involved in the implementation of such an advanced and complex data transfer scheme. Firstly, additional signals are needed to transmit IDs for every data transfer in the system. Secondly, master interfaces need to be extended to handle data transfer IDs and be able to reorder received data. Thirdly, slaves require additional logic at their interface to decode and process IDs, and ensure that the proper data transfer ordering is maintained. The *read (or write) data reordering depth* is a parameter that specifies the maximum number of read (or write) data transfers pending in the slave that can be reordered. Larger reordering depths can significantly improve performance, but also require more logic and increase system cost. Therefore, a designer must be careful in deciding a value for this parameter. There is typically a maximum value of reordering depth beyond which the performance does not improve for a given application [8]. This threshold corresponds to the maximum level of data traffic parallelism in the application, and can be obtained after performance profiling (described in more detail in Chapter 4). Finally, additional bus logic must also be added (to the arbiter module, or separately) to ensure that data transfer ordering is maintained, for transfers originating from multiple masters.

2.3.6 Broadcast Transfer

Typically, data transfers on the bus involve just two components—a master and a slave. Appropriate select signals ensure that only the source and destination components sample data onto and off the bus. However, it is possible for the data on the bus to be "visible" to other components on the bus, besides the two that are involved in the transfer. This is because every time a data item is transmitted over a bus, it is physically broadcast to every component on the bus. A *broadcast transfer* is one that involves a source component transmitting data on the bus, and multiple components sampling the data off the bus. One of the uses of this transfer mode is for snooping and cache coherence protocols. When several components on the bus have a private cache which is fed from a single memory module, a problem arises when the memory is updated (for instance, when a cache line is written to memory by a component). In such a case it is essential that the private caches of the components on the bus invalidate (or update) their cache entries to prevent reading incorrect values. Broadcasting allows the address of the memory location (or cache line) being updated to be transmitted to all the components on the bus, so that they can invalidate (or update) their local copies.

2.4 BUS TOPOLOGY TYPES

Bus-based communication architectures can have several different types of bus arrangements or topology structures which affect the cost, complexity, power, and performance profiles of the communication architecture. Figure 2.9 shows the major bus topology types that are used in SoC designs. The simplest scheme for component interconnection is the *single bus* topology shown in Fig. 2.9(a). All the components in the system are connected to a single shared bus. An example of a commercial SoC with a shared bus is the DaVinci family of digital video processing SoCs from Texas Instruments [25]. While such a configuration is sufficient for very small SoCs having only a few components, it does not scale well to handle larger systems. This is because a single bus allows only a single data transfer at a time. A more efficient topology that allows multiple data transfers in parallel is the *hierarchical bus* topology shown in Fig. 2.9(b). In this topology, the components are connected to multiple buses that interface with each other using a bridge component. Concurrent data transfers are possible on each bus, provided that the components are allocated to the buses in such a manner that there is minimum interaction between components on different buses. Since buses can have different clock frequencies, the bridge component can be quite complex, to handle interbus transactions, data buffering, frequency conversion, etc. There are several commercial SoCs today that make use of the hierarchical bus topology, such as the customizable multiprocessor ARM PrimeXsys SoCs [27] that are widely used

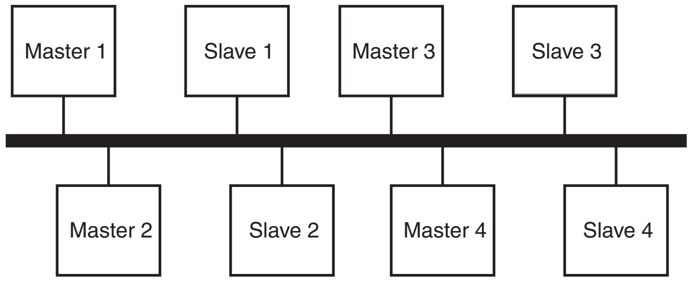

FIGURE 2.9(a)

Different bus-based communication architecture topology structures: single bus

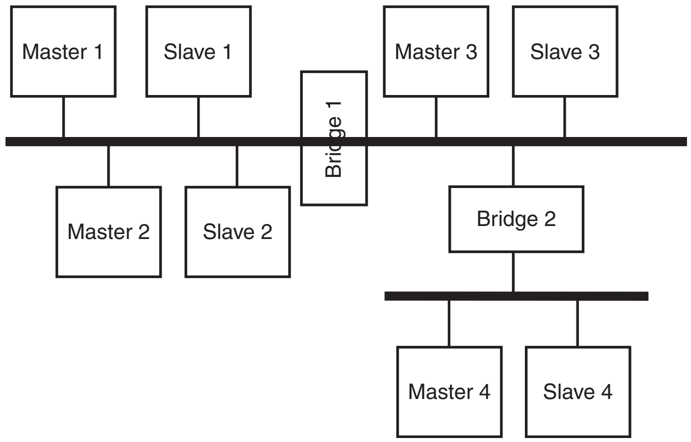

FIGURE 2.9(b)

Hierarchical bus

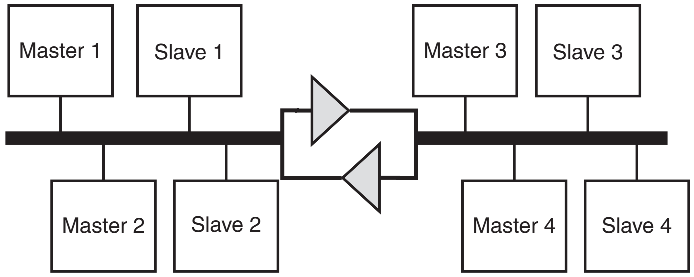

FIGURE 2.9(c)

Split bus

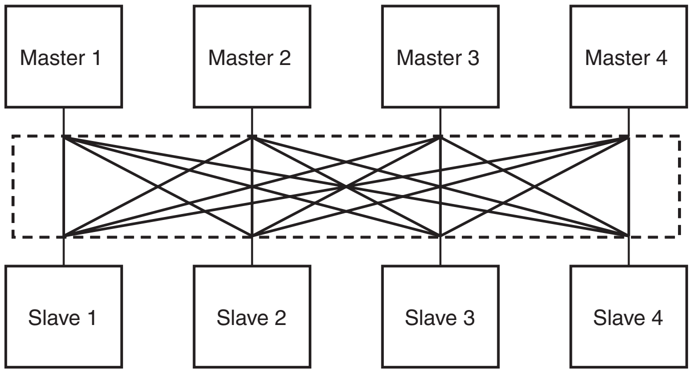

FIGURE 2.9(d)

Full bus crossbar (or point-to-point bus)

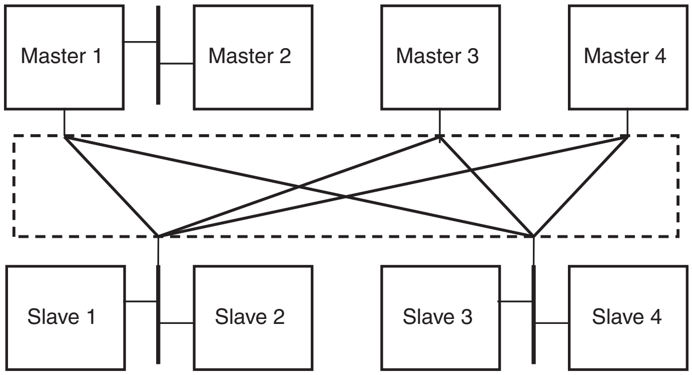

FIGURE 2.9(e)

Partial bus crossbar

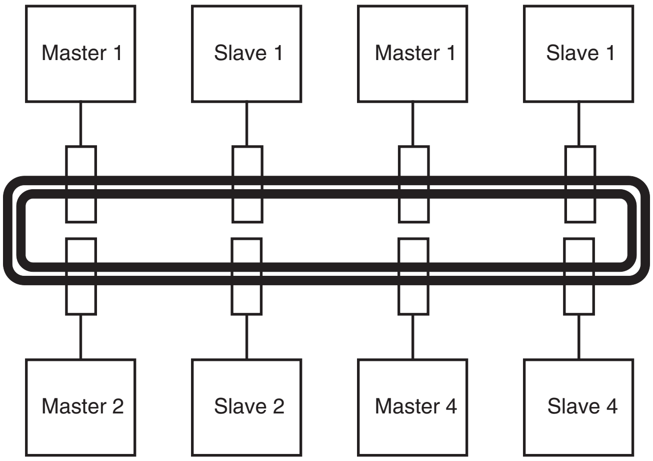

FIGURE 2.9(f)

Ring bus

in handheld devices such as mobile phones, PDAs (personal digital assistants), GPS (global positioning system) units, and PMPs (personal media players). A simpler variant of the hierarchical bus is the *split bus* topology shown in Fig. 2.9(c). This topology also uses multiple buses, but the interface between these buses is a simple tri-state buffer-based scheme. This prevents using a more complex protocol as in the case of the hierarchical bus topology, but the simpler tri-state interface can be more efficient as far as energy consumption is concerned [9].

For high performance systems that require extensive data transfer parallelism, the *full bus crossbar* (also called *full bus matrix*) topology shown in Fig. 2.9(d) is a suitable choice. An example of a commercial SoC with a full bus crossbar topology is the Niagara multiprocessor SoC from SUN [26], which connects eight SPARC processor cores (each having hardware support for up to four threads) to four L2-cache banks, an I/O bridge, and an FPU (floating point unit). While this solution might be excessive for smaller systems [10], several research efforts [11–13] have shown the utility of a full bus crossbar in providing significantly higher data throughput rates compared to single and hierarchical bus architecture alternatives. In this topology, every master is connected to every slave in the system with a separate bus, which can be considered to be a kind of point-to-point interconnection. The large number of buses allows multiple data transfers to proceed in parallel. Note that unlike the previously presented topologies, a full crossbar system requires separate arbitration for every slave. While a full crossbar bus topology offers superior parallel response, the excessive number of buses can take up a large area, increase power consumption, and make it practically impossible to achieve routing closure [14]. To overcome these limitations, one alternative is to use a hybrid shared bus/point-to-point topology, which clusters some of the components in the full crossbar bus, as shown in Fig. 2.9(e). Such a *partial crossbar* bus topology has a fewer number of buses, a smaller area, reduced power consumption, and less wire congestion than a full crossbar bus topology [14–16]. However, the clustering of components in the partial crossbar bus topology also reduces the parallelism in the system, which in turn reduces performance. Designers must therefore carefully trade-off these factors while designing a partial crossbar bus topology. Chapter 6 describes research efforts that attempt to optimally trade-off design cost and performance while designing crossbar bus architectures.

Finally, another commonly used high performance bus topology is the *ring bus* topology, shown in Fig. 2.9(f). In this topology, components are connected to one or more concentric ring buses. Data can be transferred from the source to the destination either in a clockwise or an anti-clockwise direction, depending on factors such as bus segment availability and shortest distance to destination. An example of such a ring bus can be found in the IBM Cell multiprocessor SoC [28] that has been used in the PlayStation 3 gaming console. The element interconnect bus (EIB) in the Cell multiprocessor consists of four ring buses, two of which transfer data in the clockwise direction and two in the anti-clockwise direction. The EIB connects the PPE (power processor element), eight SPEs (synergistic processor elements), a MIC (memory interface controller), and an external BIC (bus interface controller). The EIB ring bus was chosen over a full crossbar bus primarily because of its lower area footprint, while still offering an acceptable bandwidth.

2.5 PHYSICAL IMPLEMENTATION OF BUS WIRES

With the rising complexity and ever increasing levels of component integration in SoC designs, the volume of data transfers between components has also increased. To meet performance requirements, bus clock frequencies have been steadily increasing, since data throughput is a function of bus clock frequency, as given by the relation:

$$throughput_{bus} = width_{bus} \times clock\_frequency_{bus}$$

where the *throughput* is in terms of megabits per second if the *width* is specified in terms of bits and the *frequency* in terms of megahertz (MHz). Now, a rise in bus clock frequency implies a shorter bus clock cycle period. For instance, a bus with a clock frequency of 100 MHz has a bus clock cycle duration of 10 ns, whereas a bus with a higher clock frequency of 500 MHz has a bus clock cycle duration of only 2 ns. This has major implications as CMOS process technology continues to shrink. Bus wires are implemented as long metal lines on a silicon wafer, and transmit data using electromagnetic waves which cannot travel faster than a finite speed limit. With shrinking process technology, logic components such as gates have also correspondingly decreased in size. However, the wire lengths have not shrunk accordingly, resulting in relatively longer communication path lengths between logic components in newer technologies. Worse, due to increasing bus clock frequencies, the time allowed for a signal on the bus to travel from the source to its destination in a single bus clock cycle has reduced considerably, as discussed above. Another way of stating this fact is that the distance that can be covered by a signal on the bus in a single clock cycle has been reduced with increasing clock frequencies and shrinking CMOS process technology. Consequently, it can take multiple cycles to send a signal across a chip. For instance, it has been estimated that in the 50 nm process technology node, the signal propagation delay will be as high as 6–10 bus clock cycles [17] to send a signal from one end of the SoC to the other. This increase and unpredictability in signal propagation time can have serious consequences for the performance and correct functioning of the SoC design.

Several ways of tackling this problem have been proposed. Hierarchical [4, 5] or split bus [9] communication architectures partition otherwise long bus lines into shorter ones, separated by bridges, or tri-state buffer structures, respectively. This makes it possible for signals to traverse a bus segment in a single clock cycle. Hierarchical bus architectures such as AMBA 2.0 [4] allow different buses to operate at different bus clock frequencies. Utilizing multiple clock domains separated by bridge logic components allows better signal propagation management, since signals need to traverse smaller wire lengths. Another commonly used technique makes use of register slices [1] or buffers to pipeline long bus wires. Such a scheme enables a signal to be in flight for several cycles, taking a single clock cycle to move between successive pipeline stages, before finally reaching its destination. Carloni and Sangiovanni-Vincentelli [18] described one such approach to pipeline long wires having a latency of more than one cycle. Pipeline logic elements (called relay stations) were inserted into long wires after the physical layout phase, to ensure latency insensitive design. Yet another technique is to make use of asynchronous buses which discard the clock signal altogether, in favor of more elaborate handshaking based synchronization mechanism. The MARBLE bus architecture [6] is one example of an asynchronous communication architecture. It is also possible to make use of globally asynchronous, locally synchronous (GALS) techniques for communication, which use asynchronous handshaking synchronization for long wires that interconnect smaller synchronous regions, consisting of components connected via a synchronous bus. Finally, there are several low level techniques that are commonly used to reduce signal propagation delay on any wire, such as by inserting repeaters, or varying the dimensions of the wires (wire sizing) [19].

2.6 DISCUSSION: BUSES IN THE DSM ERA

With the scaling of CMOS technology below 90 nm, SoC designs have entered the DSM era, characterized by high levels of component integration, high clock frequencies, and low signal voltages. In addition to an increase in signal propagation delay, which requires making architectural changes to buses (as discussed in Section 2.5), DSM effects will create severe signal integrity problems that will make it harder to guarantee error-free data communication on buses. The signal integrity problem can be defined as the scenario where the received signal at the destination is different from the transmitted signal at the source driver, for a bus wire. This happens because of signal degradation caused by various DSM effects that create noise (i.e., a deviation of a signal from its intended or ideal value). Some of the important DSM effects that can cause noise on buses include crosstalk, external electromagnetic interference, transmission line effects, and soft errors. These effects are described below.

Crosstalk is the phenomenon of noise being caused on a signal A due to the coupling with another signal B. Due to the close proximity of bus wires, near-field electromagnetic coupling causes inductive and capacitive crosstalk on the bus signals. Even when wires are far apart, crosstalk can still be present between signals due to coupling facilitated by the common substrate, a shared power supply or ground, or a shared signal return path. As wires become narrower (with technology scaling) and clock frequencies increase, fringing field effects and inductance effects become larger for wires, leading to higher inductive and capacitive crosstalk. *Electromagnetic interference* (EMI) from large external electric and magnetic fields can couple into circuits and create unwanted noise. As highly integrated, portable wireless communication SoCs increasingly consist of analog, RF, and digital circuits, EMI due to external and internal coupling will increase. Long on-chip buses in particular will be the sources and receptors of EMI noise. *Transmission line effects* will arise due to discontinuities in wires that are modeled as transmission lines. In DSM technologies, when a wire is longer than 1/10 of the wavelength of the signal frequency component that is transmitted, the wave nature of the propagated signal must be modeled, otherwise significant errors may result. Wires will thus have to be modeled as transmission lines to avoid errors during signal analysis. Discontinuities in these transmission lines (due to various factors such as capacitive loads, vias, wire bends, package pins, crossover wires, and non-ideal receivers) can result in impendence mismatches. Such mismatches will create noise as a result of signal reflections at the discontinuities. Finally, signal integrity will also be influenced by *soft errors* that are caused by a collision of thermal neutrons (produced by the decay of cosmic ray showers) and/or alpha particles (produced by impurities in the substrate). Highly integrated SoCs will be particularly susceptible to soft errors that will create spurious pulses and interfere with signals on buses.

As a result of all the DSM effects described above, it will become harder to guarantee error-free data transfers on buses. Reduced signal swings in DSM technologies will result in a further reduction of voltage noise margins, increasing the probability of transmission errors in the presence of even the smallest sources of noise. Many other factors such as increasing wire resistance due to skin effect at high frequencies, increasing number of metal layers that increase cross-layer coupling, and timing errors due to jitters will cause new challenges in DSM technologies. These problems have been well summarized in several books [20–24]. It is very important that emerging tools and methodologies for on-chip communication architecture design be able to handle not only the increased number of wires, but also allow designers to predict and address DSM issues as early in the design flow as possible, to reduce design iterations (instead of finding and fixing the problems in post-layout). We will revisit DSM-aware methodologies, techniques, and architectures throughout this book.

2.7 SUMMARY

In this chapter, we presented some of the basic concepts of bus-based communication architectures. We first introduced the components and terminology used to describe these communication architectures and then covered some of their major characteristics such as bus signal types, physical structure, clocking, decoding, and arbitration. We presented an overview of some of the basic data transfer modes that are used during data transfers, and then described some of the more advanced transfer modes intended to improve bus utilization and throughput performance. Some commonly used bus topology structures were described, and finally we discussed some of the issues in the physical implementation of bus wires. In the next chapter, we will look at examples of some standard bus-based communication architectures that are widely used in SoC designs.

REFERENCES

[1] ARM AMBA AXI Specification www.arm.com/armtech/AXI.

[2] 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. 3, March 2006, pp. 241–253.

[3] "STBus Communication System: Concepts and Definitions," *Reference Guide*, STMicroelectronics, May 2003.

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

[5] IBMCoreConnect Specification, http://www.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.

[6] W. J. Bainbridge and S. B. Furber, "Asynchronous macrocell interconnect using MARBLE," in *Proceedings of Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 122–132.

[7] Sonics, "Sonics μNetworks technical overview," Sonics Inc, June 2000.

[8] 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.

[9] Cheng-Ta Hsieh and M. Pedram, "Architectural energy optimization by bus splitting," in *Proceedings of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)*, Vol. 21, No. 4, April 2002, pp. 408–414.

[10] S. Brini, D. Benjelloun and F. Castanier, "A flexible virtual platform for computational and communication architecture exploration of DMT VDSL modems," in *Proceedings of DATE*, 2003, pp. 164–169.

[11] V. Lahtinen, E. Salminen, K. Kuusilinna and T. Hamalainen, "Comparison of synthesized bus and crossbar interconnection architectures," in *Proceedings of the 2003 International Symposium on Circuits and Systems*, 2003, pp. 433–436.

[12] Y. Zhang and M. J. Irwin, "Power and performance comparison of crossbars and buses as on-chip interconnect structures," in *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers*, 1999, pp. 378–383.

[13] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 752–757.

[14] 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.

[15] S. Murali and G. De Micheli, "An application-specific design methodology for STbus crossbar generation," in *Proceedings of Design, Automation and Test in Europe* (DATE), 2005, pp. 1176–1181.

[16] 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.

[17] R. Ho, K. W. Mai and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, Vol. 89, April 2001, pp. 490–504.

[18] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in SoC design," *IEEE Micro*, Vol. 22, No. 5, September/October 2002, pp. 24–35.

[19] A. B. Kahng, S. Muddu and E. Sarto, "Interconnect optimization strategies for high-performance VLSI designs," in *Proceedings of Twelfth International Conference on VLSI Design*, 1999, pp. 464–469.

[20] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley, Reading, MA, 1990.

[21] W. J. Dally and J. H. Poulton, *Digital Systems Engineering*, Cambridge University Press, Cambridge, UK, 1998.

[22] C.-K. Cheng, J. Lillis, S. Lin and N. Chang, *Interconnect Analysis and Synthesis*, Wiley-Interscience, 1999.

[23] Q. K. Zhu, *Interconnect RC and Layout Extraction for VLSI*, Trafford, 2002.

[24] M. Nakhla and R. Achar, *Introduction to High-Speed Circuit and Interconnect Analysis*, Omniz Global Knowledge Corporation, 2002.

[25] D. Talla, "An innovative HD video and digital image processor for low-cost digital entertainment products," *Proceedings, HotChips*, 2007.

[26] S. Phillips, "VictoriaFalls: Scaling highly-threaded processor cores," *Proceedings, HotChips*, 2007.

[27] ARM PrimeXsys Platform, http://www.arm.com.

[28] IBM Cell Project, http://www.research.ibm.com/cell.