# Chapter 2
# The AMBA SOC Platform

SoCs contain numerous IPs that provide varying functionalities. The interconnection of IPs is non-trivial because different SoCs may contain the same set of IPs but have different behaviour. In this chapter, we focus on how IPs are connected in a typical SoC, and how the information about their interconnection can be used to lead us to a formal model for IPs and SoCs. Figure 2.1 presents the mobile phone SoC example introduced previously in Chap. 1. This SoC contains processors, memories, and several peripherals. These IPs have varying functionalities, speeds, and data-exchange behaviours, and hence their interconnection must be comprehensive enough to be compatible to their individual behaviours, and at the same time provide a standard method to connect such diverse IPs.

Typically, IPs are connected using a *SoC bus*. A SoC bus is the interconnect fabric that provides a standard interface, or a *platform*, that can be used to connect different IPs in essentially the same way using *platform-based design* [SV02]. A *platform* is a combination of generic hardware (buses, processors, memory etc.) and software (microcontroller code) that can be customized by incremental changes in software and hardware to allow for a faster turnaround for differentiated products. Popular SoC buses available today include Altera Avalon [XZ03], IBM CoreConnect [HD02], IDT IPBus [IDT04], Open Core Protocol [Web00], and Wishbone [Pet99]. In this chapter we study the Advanced Microcontroller Bus Architecture, or AMBA [ARM99] in detail, since it is the de facto standard for creating SoCs. Examples of AMBA's use in industry include many chip makers, such as NVidia, Qualcomm, Actel, etc [HLP+10, AZ10].

AMBA, provided by ARM, describes a generic set of buses for use in SoCs, application-specific integrated circuits (ASICs), and traditional micro-controllers. The AMBA standard was first released in 1996, and is an open standard, which allows it to be used by all SoC manufacturers that use ARM-based processors in their SoCs. AMBA contains two types of buses: high performance *system* buses to connect core IPs, and a flexible *peripheral* bus to connect a multitude of peripherals and components. In Fig. 2.1, the AMBA advanced high performance bus (AHB) (a system bus) is used to connect a processor, a DSP, and high-performance memory controllers, and the AMBA advanced peripheral bus (APB) is used to connect

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_2, © Springer Science+Business Media New York 2014

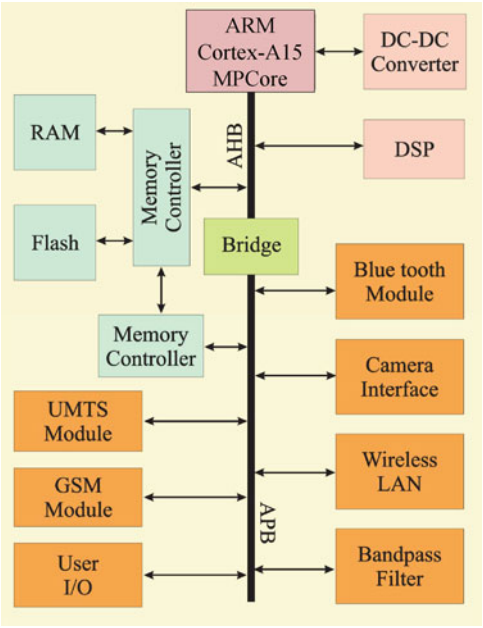



**Fig. 2.1** An AMBA-based mobile phone SoC example

various peripheral IPs. AMBA allows us the flexibility of upgrading this SoC by replacing some IPs with newer versions (such as the Bluetooth module) or by the addition of more IPs that add functionality to the existing design (such as an FM receiver). Note that Fig. 2.1 also contains a *Bridge*, which connects the AHB and APB buses. Bridges are standard bus-to-bus interfaces that allow IPs connected to different buses to communicate with each other in a standardized way.

This chapter first describes the AMBA standard in Sect. 2.1. This allows us to gain insights into how typical IPs are interconnected to create a SoC. As discussed previously in Chap. 1, our goal is to formally model IPs and SoCs in order to precisely capture individual IP behaviours as well as their interactions. Therefore, in Sect. 2.2, we look at how we can move towards a formal model for IPs and their interactions in SoCs based on AMBA.

## 2.1 The AMBA Standard

### *2.1.1 Terminology*

AMBA uses specific terminology for SoC components. Since this terminology will be used throughout this book, we introduce it here with an illustration of a simplified SoC based on the AMBA advanced system bus (ASB) shown in Fig. 2.2.

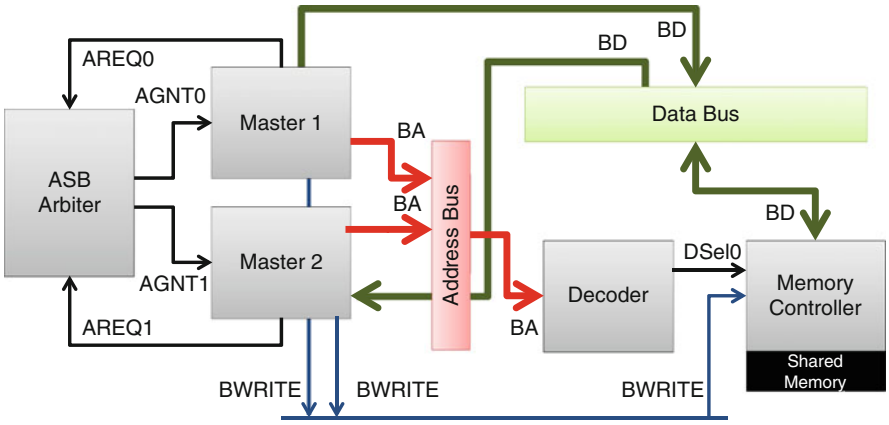



**Fig. 2.2** A simple ASB system

1. *Master*: A master device, usually connected to a high-throughput system bus, is an IP which is capable of initiating communications on the bus. Figure 2.2 contains two such masters: *Master 1* and *Master 2*.
2. *Arbiter*: As an SoC may contain multiple masters, an appropriate arbitration algorithm [Fly97] is needed. In the ASB system as shown in Fig. 2.2, a central arbiter is available, which connects to all masters in the system directly, and can provide bus access to them using an appropriate scheduling scheme. The central arbitration scheme can be round-robin, prioritized, etc.
3. *Slave*: A slave device responds to communication requests from a master. While peripheral devices, memory controllers, and network controllers are categorized as slaves, even processing elements such as DSPs may act as slaves. In Fig. 2.2, the SoC contains a single memory controller slave.
4. *Decoder*: Masters use the address bus of the interconnect to invoke slaves. A decoder decodes the address (or a part of it) into relevant control signals to enable slaves. For example, in Fig. 2.2, the decoder reads the address issued by the masters, and translates it into a slave select signal `DSel0` for the memory controller slave.
5. *Bridge*: AMBA provides the Advanced Peripheral Bus which is used to connect multiple peripheral IPs to a SoC. As shown in Fig. 2.1, a mobile phone SoC can contain a number of such peripherals which provide key functionalities to the SoC. Peripheral buses are slower and than system buses like ASB, and provide different communication protocols to the IPs connected to them. A bridge allows a master on the system bus to communicate with a peripheral device even though they do no share the same communication protocol.

In addition to the above IP-types, an SoC contains the following mechanisms for inter-IP communications:

- *Control signals*: AMBA buses provide dedicated point-to-point lines for IPs to communicate meaningful control information using control signals. Examples include signals for arbitration (such as $AREQ0$, $AGNT0$ in Fig. 2.2), write/read signals (such as $BWRITE$ in Fig. 2.2), slave select signals (such as $DSel0$ in Fig. 2.2), etc.
- *Data bus*: A data bus is used by IPs to write and read data. It is generally of a fixed width, and in AMBA data buses are 32-bits. Each master and slave in an SoC can read or write data through the data bus, as shown in Fig. 2.2. However, only one device is allowed to use the data bus at any one time in order to maintain data consistency.
- *Address bus*: An address bus allows IPs to specify physical addresses. AMBA address buses are 32-bits typically. Each IP in the system may connect to the address bus, but only one IP is allowed to read/write at any stage. In addition to specifying physical addresses, the address bus is read by the bus decoder to generate slave-select signals (like $DSel0$ in Fig. 2.2), which allows masters to invoke specific slaves.

The communications between IPs in an AMBA SoC happen as a series of *transactions*. A transaction is invoked by a master, and consists of the following steps: securing bus access, initiating read/write operation, and completing read/write operation. Figure 2.3 presents one possible transaction that can happen in the simple SoC shown in Fig. 2.2. It is shown as a message sequence chart [ITR96], and consists of the following steps. First, master $M1$ (and then master $M2$) requests bus access from the bus arbiter by asserting $AREQ0$ ($AREQ1$) control signal. The arbiter allows $M1$ access to the bus by asserting the $AGNT0$ control signal. Master $M2$ has to wait for $M1$ to relinquish the bus access after completing the current transaction. $M1$ then places a 32-bit address on the bus (also available to read by the slave), and a 32-bit data packet on the data bus. It also asserts the $BWRITE$ control signal, which means that it is initiating a write transaction. The decoder reads the address bus and then generates the slave select control signal $DSel0$ to invoke the slave. The slave then reads the data on the data bus and completes the write transaction.

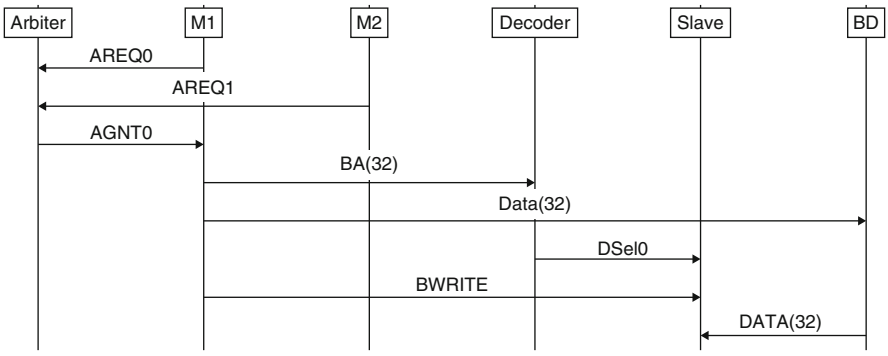



**Fig. 2.3** Example of a transaction in the ASB system shown in Fig. 2.2

AMBA based buses provide a variety of transactions. A transaction may be read (a master reading from a slave) or write (a slave reading data written by the master). Similarly, a transaction may be a *single transfer*, where a single data packet is exchanged (read/written), or a *burst transfer*, which allows multiple packets to be exchanged in the same transaction. Burst transfers are useful in high band-width data operations such as cache-line fills. In addition, AMBA provides mechanisms for error reporting (e.g. when a slave fails to correctly read or write). Each AMBA bus has different capabilities in terms of the transactions it can support, and the communication protocol may also differ from other AMBA buses.

### *2.1.2 AMBA Buses*

AMBA, first introduced in 1996, describes a number of buses and interfaces for on-chip communications. The first version of AMBA contained only two buses—a system bus and a peripheral bus. The current version, AMBA 4, contains a wide collection of high performance buses and interfaces. In this section, we look at some key AMBA buses, based mostly on the AMBA 3 standard [ARM].

#### Advanced High Performance Bus

The AHB bus was first proposed in AMBA 2, and was upgraded later in AMBA 3. A simple AHB system consists of one or more masters connected to slave devices (such as memory controllers), as shown in Fig. 2.4. This arrangement is known as a *layer*. A layer contains a unique decoder, which resolves master requests into slave activation signals. Additionally, if there are multiple masters, an arbiter is included,

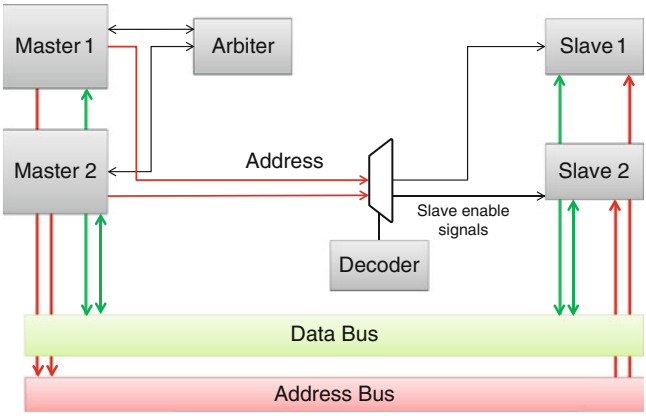



**Fig. 2.4** Single-layer AHB

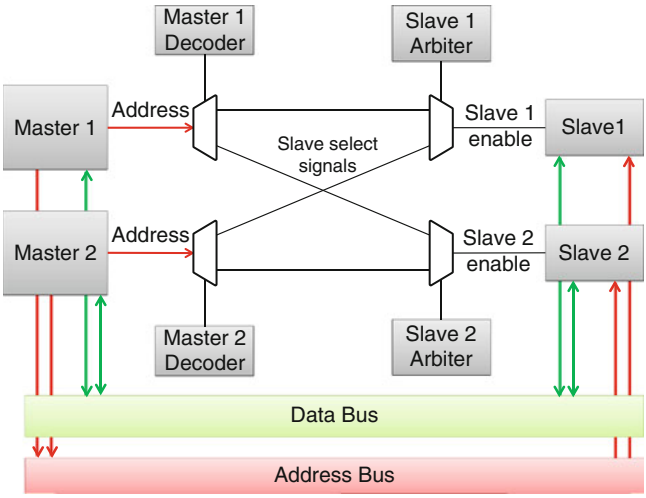



**Fig. 2.5** Multi-layer AHB

which decides which master gets to access the bus at any time. AHB masters can perform burst transfers where multiple data elements are read/written from/to a slave in a single transaction. A number of variants of AHB exist, as discussed below.

A multi-later AHB, as shown in Fig. 2.5 contains one layer (decoder and slave select signals) for each connected master in the AHB. Each slave has a slave arbiter, which decides which layer (master) is given access in case two masters request access to the slave at the same time. A multi-layer arrangement therefore allows more than one master to be actively talking to a (unique) slave in the SoC at the same time. The arbitration scheme for every slave-arbiter can be different, for example, round robin, or fixed priority. Of course, as the numbers of masters and slaves increase, the interconnect matrix becomes complex. This problem can be minimized by making some slaves *local* to a layer (and hence removing the need for separate arbiters for them), allowing multiple slaves to be accessed as a single device (reducing the number of arbiters), using multi-port slaves which do not need arbiters, and/or allowing multiple masters on a single layer (reducing the number of decoders).

The AHB-lite protocol is a high-bandwidth system that supports a single master connected to multiple slaves (shown in Fig. 2.6). This protocol does not need an arbiter due to the presence of a single master, and is therefore simpler to synthesize. A multi-multiplexer is used to ensure that only one slave writes to or reads from the data bus at any time. Multiple AHB-lite layers can be connected together using the multilayer interconnect, allowing for the synthesis of complex SoCs.

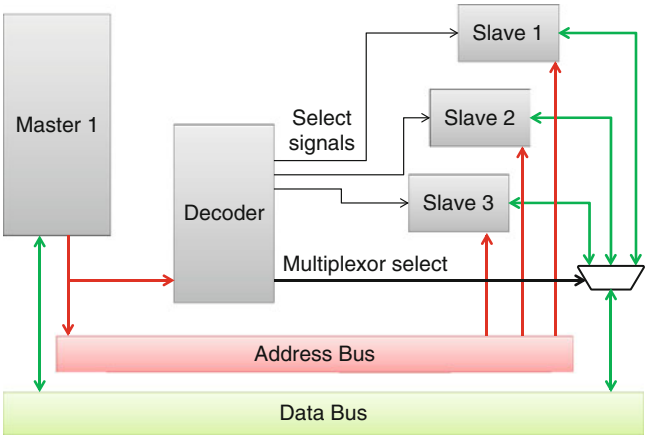



**Fig. 2.6** AHB-lite

### Advanced System Bus

Introduced in AMBA version 2.0, the ASB is a high performance synchronous bus. Multiple bus masters connect to an arbiter, which grants access to a master using the implemented arbitration scheme. ASB offers a pipelined protocol, where arbitration, address transfers and data transfers can occur simultaneously.

Figure 2.2 shows a simple ASB-based SoC. It contains two masters. Master 1 writes data to a shared memory via a memory controller slave, while master 2 reads data from this shared memory. We assume that the shared memory is simply a circular buffer, which reads or write data without needing an address. Only one master may access the slave at any time. This arbitration is achieved through the ASB arbiter, which reads requests from the two masters through the `AREQ0` and `AREQ1` signals. The arbiter then grants access to either master 1 or master 2 by asserting the signal `AGNT0` or `AGNT1`, respectively. The master granted access is called the *current bus master*, or simple the bus master.

If master 1 is chosen as the bus master, it drives a write cycle as follows. It asserts the control signal `BWRITE` which indicates a write operation. It then places an address on the 32-bit address bus. The decoder reads this address and generates the control signal `DSel0`, which activates the memory controller slave. At the same time, master 1 also writes 32-bit data `BD` onto the data bus. The slave then reads the data on the data bus into the shared memory. Note that in ASB systems, slaves can read some lines of the address bus to determine where to store the data being saved onto memory. However, as stated earlier, we assume that in our simple system, the shared memory is simply a circular buffer, and the new data is saved onto the next available location.

If master 2 is chosen as the bus master, it drives a *read* cycle. It first de-asserts the control signal `BWRITE` to indicate a read operation. It then places an address on the 32-bit address bus. The decoder reads this address and generates the control signal

DSel0, which activates the memory controller slave. When the slave is activated (and BWRITE is de-asserted), it places 32-bits of data from the shared memory onto the data bus. Master 2 can then read this data and complete the transaction.

#### Advanced Peripheral Bus

The APB bus is a low-cost non-pipelined bus used to connect low-bandwidth peripherals of an SoC. The APB is connected to a system-bus (such as AHB and AXI) using a bridge. The bridge allows masters on the AHB to address the APB system as a single slave. However, a bridge does not ensure that the interactions between masters and slaves are *correct*, i.e., as intended by the designer. APB transfers (read and write) take a minimum of 2 clock cycles with further wait states added for slower peripherals. Each transfer involves the exchange of 1 unit of data between the invoking master and the responding slave. Error responses for both reads and writes are allowed to provide extra functionality.

#### Other Buses

The advanced trace bus (ATB) provides functionality for on-chip debugging and trace analysis for AMBA-based SoCs. Each IP in the SoC that has trace capabilities is connected to the ATB. In the context of ATB, master interfaces write trace data on the bus, while slave interfaces receive the trace data from the ATB. The advanced eXtensible interface (AXI) is a high-speed, burst-only interface introduced first in AMBA 3 specifications. The AXI interface is backward compatible to AHB and APB, and hence can be used with legacy components. This generic interface is extensible to be used in a range of settings ranging from basic single-address bus single-data bus system to multilayer systems with multiple address and data buses.

## 2.2 Formal Modelling of AMBA SoCs

As discussed in the previous chapter, our aim is to use mathematically sound techniques to analyse an SoC for the satisfaction of functional specifications. In this chapter, we illustrate how to formally model AMBA systems. A formal model allows the use of mathematical reasoning to analyse a system. A formal model must be *abstract* so that it is easier to analyze and does not carry redundant details, but at the same time, it must capture adequate details about the system to be meaningful. In our context, the following details must be captured by the formal model:

1. *Control flow*: The model must be able to capture the sending and receiving of control signals of the IPs of a SoC.

2. *Data flow*: The model must be able to capture the reading and writing of data by the IPs of a SoC.
3. *Timing*: The model must be able to capture the synchronous nature of AMBA systems, where execution proceeds with respect to clocks. Note that different peripherals may use different clocks, but that these clocks are generally obtained from the same clock source.
4. *Compositionality*: The model must be modular and compositional, such that individual IPs can be modelled, and the SoC can be rendered as a composition of its IPs.

We now expand on each of the above criteria, and illustrate how these can be modelled.

### 2.2.1 Control Flow

In AMBA systems, control flow consists of the exchange of control signals between the bus components, masters and slaves. The control signals used in every AMBA bus are well defined by the AMBA specifications. For example, in the AMBA ASB (advanced system bus), a master may request bus access by sustaining the `AREQ` signal, and the bus arbiter responds using the `AGNT` signal when access is granted (as shown in Fig. 2.2).

The behaviour of a single IP can be captured using a state machine which has a finite set of states $S$ with a unique initial state $s_0$. States have transitions between them, which describe possible control flow between them. For example, consider master 1 of the AMBA ASB example shown in Fig. 2.2. This master writes data to a shared memory via the memory controller slave. We can capture the control flow in the execution of master 1 as follows. Firstly, we can identify all control states that this IP can be in at any time. These states are:

1. *Init*: The initial state of the IP upon start up.
2. *Request*: The master has requested for bus access and is waiting for bus access to be granted by the arbiter.
3. *Active*: Access has been granted, and the write operation has been started.
4. *Complete*: The write operation has been completed, and the master can relinquish bus access.

The control flow within master 1 can be described using transitions between the above states, as shown in Fig. 2.7a. Starting from the *Init* state, the IP enters the *Request* state, and waits there until the arbiter allows bus access. It then enters the *Active* state to start the transfer. A transition to *Complete* state is taken when the transfer is complete. The IP then takes a transition back to *Init* to repeat the above control flow infinitely often.

Note that, as shown in Fig. 2.7, states and transitions can have *labels*. State labels may refer to state names (as in the above figure), or to the values of the variables of

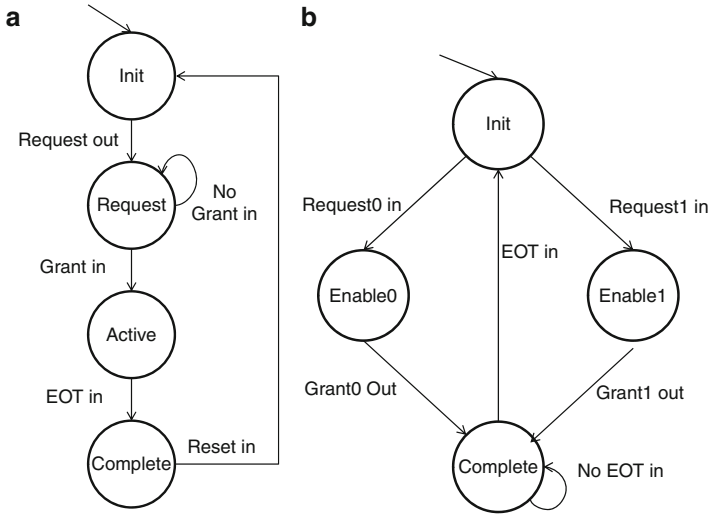



**Fig. 2.7** Modelling control flow using finite state machines. (**a**) Modelling control flow of a master IP using finite state machine. (**b**) Control flow model of the bus arbiter

an IP when it is in a specific state. Similarly, transition labels may refer to the inputs and outputs generated by the IP when a transition is taken. For example, when a transition is taken from state *Init* to state *Request*, the transition label `Request Out` indicates the emission of a request signal by the IP (to the arbiter).

The level of detail captured in each state and transition of the above model can be increased or decreased by using a denser or sparser set of propositions. Consider Fig. 2.7b, which provides a control flow model of the bus arbiter shown in Fig. 2.2. This model reads access requests from two masters, and grants access on a first-come first-served basis. When access is granted to a master, the arbiter waits in the *Complete* state until it receives the control signal `EOT` signalling an end-of-transaction. A more detailed model of this arbiter can carry actual signal names of the AMBA ASB, such as $AREQ0$ and $AGNT0$ (as shown in Fig. 2.2) instead of the more abstract labels $Request1\,in$ and $Grant0\,out$.

### 2.2.2 Data Flow

A typical single-layer AMBA system uses two main data flow channels: the address bus and the data bus. The address bus is used by the current bus master to invoke a slave. The data bus is used to read/write the actual data produced by the slave or master during the transaction. A formal model must capture all relevant data flow information and must leave out any redundant information.

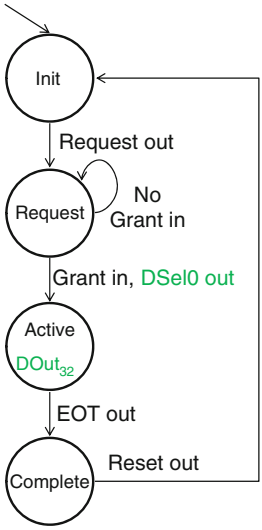



**Fig. 2.8** Modelling data flow using finite state machine

A naive way to model the above data flow is to precisely enumerate all possible addresses and data that buses can contain. Given that each of these buses is usually 32-bits in size, there are $2^{32}$ possible unique values. Capturing all these in the formal model can result in state explosion. On the other hand, we can ignore data flow all together, or in other words, we can remove all data elements from the formal model. However, such abstraction will limit the effectiveness of the model significantly, and it may become useless in most cases.

We can also take a third, more balanced approach. For the address bus, we can abstract all data out and replace them with pre-interpreted extra control signals. For example, the master may refer to a slave $S$ using the range of addresses $Ox00000000$ to $Ox000000FF$. We can simply replace the emission of any address in this range by a control signal $Invoke_S$, which can act as an additional output of the state machine representing the master IP, and will be included as an additional control input of the model of the slave $S$. The extra control signals can be captured simply as inputs and outputs of the state machine. Figure 2.8 extends Fig. 2.7, by capturing the data flow in master 1 (shown in Fig. 2.2). Note that the label of the transition from state *Request* to state *Active* contains the `DSel0 out`, which abstracts the actual address (on the address bus) issued by the master to invoke Slave0 (via the decoder) to a control signal.

For data transfers, we can restrict our model such that it only shows the *number of bits* written to/read from the data bus by a slave/master during a transaction. An IP may partially use the data bus. For example, an 8-bit serial UART IP may only read/write 8 bits from/to the 32-bit data bus. While this approach does not enumerate all possible data exchanged between IPs (and at the same time reduces state explosion), it can be useful in ensuring that all data generated by one IP is read by other IP(s), or that no data-corruption due to overflows (when existing data on

the bus is overwritten) or underflows (when an IP tries to read non-existent data). As shown in Fig. 2.8, the state label $DOut_{32}$ indicates that the IP outputs 32 bits of data to the data bus every time state *Active* is reached.

### 2.2.3 Timing

AMBA has synchronous buses, which enforce connected IPs to execute with respect to a common clock signal. Different buses in a system might use different clocks. For example, the APB will usually have a slower bus than the ASB. Usually, these different clocks are *rational*, or are obtained by dividing the system clock by different integer values.

We can capture timing aspects by simply ensuring that the control and data flow of the system is divided into discrete steps (or states) such that each step represents the state of the IP at a different clock edge, or tick. For example, we can assume that the state machine describing the control-data flow of master 1 (Fig. 2.2) as illustrated in Fig. 2.8 executes synchronously. In other words, only one transition is taken at every tick of the clock.

### 2.2.4 Compositionality

We require that the formal model be capable of not only describing individual IPs, but also partial and complete SoCs. For this, we require a *composition operator* to combine individual models into a single model. Given that our model is based on finite state machines that execute synchronously, we can use existing operators such as the synchronous parallel [BCE+03] operator.

As discussed above, the state machine shown in Fig. 2.8 executes synchronously with the common bus clock of the SoC shown in Fig. 2.2. Moreover, all other IPs connected to the same bus (master 2, arbiter, the memory controller slave) also execute with respect to the same clock, and therefore will *synchronize* their execution with master 1 at each clock tick. In other words, we can describe the SoC as a *composition* of its connected IP protocols. More details about compositionality and how we compose IP protocols will appear later in the book.

## 2.3 Conclusions

This chapter provides a detailed look at ARM-based SoC architectures. We explain the architecture of a typical mobile phone SoC. We discuss the AMBA family of SoC buses, and show how different interconnects can be used to implement different parts of an SoC. As AMBA is an open standard that can be used royalty-free, it is currently the de facto standard for SoC design.

Once the SoC architecture is introduced, we need to discuss how the on-chip communication protocols of the IPs are described. We introduce the concept of a finite state machine (FSM)-based representation of the on-chip communication protocols of SoCs. This model captures the control flow, data flow, and timing characteristics of these protocols. A formal representation of these FSMs is presented in Chap. 4 as synchronous Kripke structures (SKS). SKS allow us to formally capture key details of IPs and SoCs, which can be used to analyze them statically using techniques such as model checking/module checking. The next chapter provides such background material i.e., formal modelling and analysis using model/module checking.



http://www.springer.com/978-1-4614-7863-8

Correct-by-Construction Approaches for SoC Design

Sinha, R.; Roop, P.; Basu, S.

2014, XXI, 144 p., Hardcover
ISBN: 978-1-4614-7863-8