Using formal techniques to Debug the AMBA System-on-Chip Bus Protocol

Abhik Roychoudhury Tulika Mitra S.R. Karri
School of Computing
National University of Singapore
Singapore 117543
{abhik,tulika,karrisid}@comp.nus.edu.sg

Abstract

System-on-chip (SoC) designs use bus protocols for high performance data transfer among the Intellectual Property (IP) cores. These protocols incorporate advanced features such as pipelining, burst and split transfers. In this paper, we describe a case study in formally verifying a widely used SoC bus protocol: the Advanced Micro-controller Bus Architecture (AMBA) protocol from ARM.

In particular, we develop a formal specification of the AMBA protocol. We then employ model checking, a state space exploration based formal verification technique, to verify crucial design invariants. The presence of pipelining and split transfer in the AMBA protocol gives rise to interesting corner cases, which are hard to detect via informal reasoning. Using the SMV model checker, we have detected a potential bus starvation scenario in the AMBA protocol. Such scenarios demonstrate the inherent intricacies in designing pipelined bus protocols.

1 Introduction

With the increase in size and complexity of system-on-chip (SoC) designs, functional verification has become important and more difficult to achieve. Currently an SoC design is typically bus based, that is, a number of heterogeneous functional modules are connected to a common bus (or a hierarchy of buses). Many of the modules connected to the bus are vendor provided Intellectual Property (IP) cores. The design for these cores are typically not available; however they are often pre-validated. Still, we would need to validate the interactions among the IP cores. In a bus based SoC design, this could be achieved in the following steps (see [4] for a similar methodology).

- Design/verify interfaces to connect IP cores to a bus.
- Verify the bus protocol.
- Check that the IP cores together with their interfaces conform to the specific bus protocol.

We do not address the issue of component interface design in this paper. Currently a huge body of work is devoted to this area with focus on interface modeling, verification and correct-by-construction synthesis. The interested reader could refer to [10] for example.

In this paper, we verify the interactions among various IP cores via the SoC bus, in the AMBA AHB protocol. Even today, these interactions are specified informally in design documents via timing diagrams and English descriptions. Such an informal descriptions makes it almost impossible to reason about the correctness of nontrivial protocols. We employ *formal verification* techniques in validating the bus protocol.

We first develop a formal specification of the protocol. The bus interface of each component is modeled as a finite state machine. The protocol is then defined to be the synchronous composition of these state machines. This finite state description of the protocol can be *formally* verified by a state space search technique called model checking [5]. Model checking is an automated verification technique for checking functionality properties of finite state concurrent systems. It allows verification of properties specified in a temporal logic, common forms of which include safety, liveness etc. In particular, we have used the Cadence SMV model checker [3] to automatically verify safety properties in the AMBA protocol.

Using the SMV model checker, we found a scenario which leads to bus master starvation: a master requesting the bus but being denied access forever. The scenario shows a subtle incompleteness in the protocol specification which would be very hard to detect without formal verification. More importantly, this starvation scenario demonstrates the intricacy involved in designing pipelined bus protocols. Pipelining is an important performance enhancing feature of many on-chip and off-chip protocols such as CoreConnect Bus, Intel Itanium Bus etc. The AMBA protocol is also pipelined, that is, transfer A_2 may start before the previous transfer A_1 has completed. Thus the protocol designer must be cautious about how to roll back transfer A_2 if transfer A_1 fails to complete. From our experience in understanding the AMBA protocol, we believe that the complication of pipelining makes it harder to detect bugs via human reasoning.

Contributions In summary, the contributions of this paper are:

- We exercise a practical verification methodology on a widely used system-on-chip bus protocol. The methodology involves formally verifying the protocol assuming that the individual component implementations conform to the protocol. Given the wide usage of AMBA, formal verification of this protocol is of utmost importance (which, to the best of our knowledge, has not been achieved so far). In fact, verification of pipelined bus protocols have only been studied very recently. In our work, the modeling of the protocol is done by hand whereas the verification is done automatically via model checking.
- Our efforts at model checking the AMBA protocol found a potential starvation scenario. This arises from an incompleteness in the specification. We believe it is important to document such corner cases for widely used protocols like AMBA. Furthermore, the starvation scenario that we found shows some inherent complications in designing pipelined bus protocols.

Organization The rest of this paper is organized as follows. In the next section, we discuss related work on run-time monitoring and bus protocol specification/verification. In Section 3, we give a brief overview of the AMBA bus protocol. In Section 4 we discuss our experience in model checking the protocol. In particular, we outline a subtle corner case detected by the model checker. Finally, we present our conclusions and possible future work in Section 5.

2 Related Work

Formal specification and verification of bus protocols (such as the PCI Local Bus) have been studied widely [1, 4, 8, 12]. However, these protocols do not involve pipelined data transfers. The work in [11, 12] promotes a specification style in which the bus protocol is described via an observer which raises error signals on violation of the protocol. This observer also detects the agent which is responsible for the error. Apart from finding protocol specification errors, such a specification style can aid the construction of a run-time monitor which checks protocol implementation.

The AMBA bus protocol involves advanced features such as pipelining and split transfers. This leads to interesting corner cases involving the splitting of a transfer whose subsequent transfers have been initiated. To the best of our knowledge, formal specification and verification of pipelined bus protocols has not been studied until recently. In particular, [11] formally specifies the pipelined Itanium bus protocol.

In [9], monitors are developed for watching the components (such as master, slave) connected to a bus system at run-time; this methodology is then applied to develop monitors for the AMBA AHB protocol. Note that this work does *not* correspond to a formal verification of the AMBA protocol itself. Instead, it performs functional verification of the circuit blocks participating in the protocol. Our work studies the interactions between components in the AMBA protocol via model checking.

3 The AMBA Bus Protocol

In this section, we outline the AMBA system-on-chip bus protocol. Many of the features of this protocol, such as pipelined transfers and split transfers are present in other SoC communication protocols such as CoreConnect [7].

Bus Architecture The architecture of the AMBA bus consists of a high-performance bus, called the AHB and a peripheral bus called the APB. The AHB and the APB are connected via a bus bridge. Several masters and slaves can be connected to the AHB, but at a time only one master is allowed access. The master to be allowed access is selected by an arbiter. Which slave services a transfer depends on the address being read/written. The AHB-APB bridge serves as a slave on the AHB, and the only master in the APB. The various low performance peripherals on the APB serve as the APB slaves.

Pipelining and Wait Cycles The AMBA protocol allows burst transfers by a master which has been granted bus access. The individual transfers within a burst are called as *beats*. The address and data of the different beats in a single burst are transferred in a *pipelined* fashion. A write burst which writes data D_1, D_2, D_3 to addresses A_1, A_2, A_3 respectively is shown in Figure 1. Note that data D_i and address A_{i+1} are transmitted in the *same* clock cycle on the *HADDR* and *HWDATA* lines. Thus the address and data phases of consecutive beats within a burst can overlap.

The protocol allows a slave to insert wait cycles by deasserting a HREADY signal if the slave is not ready to service a transfer. This extends the data phase of a transfer. Due to the pipelined nature of the bus, the address phase of the next transfer also has to be extended. Figure 2 shows

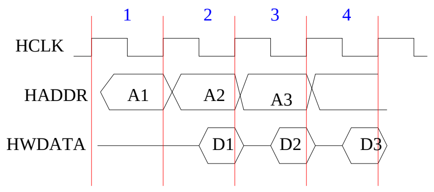

Figure 1. Pipelined transfer in AMBA

the writing of D_1, D_2, D_3 to addresses A_1, A_2, A_3 with the insertion of a single wait cycle in the transfer of D_2.

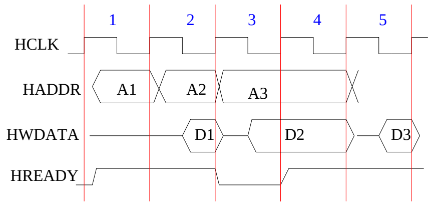

Figure 2. Pipelined transfer with wait states

Split and Retry Response In order to prevent an excessive number of wait cycles (and hence wastage of bus bandwidth) for a particular transfer, the protocol allows the release of bus access to the other masters. This is co-ordinated by the slave which either informs the arbiter of its temporary inability to service a master (a SPLIT response) or informs the master to retry the transfer (a RETRY response).

The provision of *split transfers*, that is, temporarily suspending a transfer and resuming it later when the slave is ready, raises many important questions. The pipelined nature of the AMBA bus further complicates the situation. Consider a burst transfer of addresses A_1, A_2 from a master m where the transfer for A_1 causes a split. By the time the slave's response is known, the address A_2 is already transmitted. However, since the transfer of A_1 has been temporarily suspended, the protocol must kill the transfer of A_2 as well. Furthermore, the SPLIT/RETRY responses can also cause a bus starvation scenario under certain circumstances. In the next section, we discuss how we can formally verify such design invariants by employing model checking.

4 Model Checking the Protocol

We used the Cadence SMV symbolic model checker [3] to formally verify the AMBA protocol. The aim of this exercise is to find subtle bugs in the informal protocol specification.

The input language of SMV allows us to describe each of the modules in the protocol (master, slave, arbiter, decoder) as a finite state machine. In particular, the user specifies the initial states and transition relation of each of the modules. From the description of each of the modules, SMV constructs a global state transition graph of the entire system. The transition relation and sets of states are viewed as boolean functions. These are represented efficiently by a compact data structure called Binary Decision Diagrams (BDD) [2] which involve structure sharing.

Modeling in SMV For the AMBA bus protocol, we first modeled the Advanced High Performance Bus (AHB). AHB consists of the following modules: (a) multiple masters (b) multiple slaves (c) an arbiter, (d) an address decoder, (e) a default master, and (f) a default slave. Masters request bus access from the arbiter. The arbiter is fair and it ensures non starvation of requests. A master which is granted bus access conducts a sequence of read transfers or a sequence of write transfers. The address of any transfer is passed on to the decoder, which selects the slave to service the transfer. Thus, the decoder has an implicit memory map. In case the memory map of the decoder is incomplete, the protocol has a default slave (which services addresses not mapped to any other slave). Similarly, there is a default master which is granted bus access when no master is requesting.

Next, we model the Advanced Peripheral Bus (APB). In particular, the AHB-APB bridge is modeled as another slave on the AHB; it is selected by the AHB decoder just like any other AHB slave. The bridge however has a reduced set of response signals as compared to other AHB slaves. This is because the APB slaves have a smaller response set, and the bridge can only echo these responses back to the AHB. In addition, we must model the additional control logic in the bridge which sets the APB bus to idle in the absence of any activity. Consequently, when there is a read/write request from AHB, there is a additional cycle for set-up. We can then verify simple properties of the AHB-APB interface such as: *Every read request from AHB to APB incurs at least one wait cycle*.

Logic for Specifying Properties The SMV Model checker allows us to specify properties of the protocol in Computation Tree Logic (CTL) [6]. Assume that φ is a state formula, that is, a property of states in the transition system we are verifying. Then, *φ is an invariant* is stated in CTL as $\mathsf{AG}\varphi$. *φ is true in all the next states (at least one next state)* is stated as $\mathsf{AX}\varphi$ ($\mathsf{EX}\varphi$). *φ is eventually true in all outgoing paths (at least one outgoing path)* is stated as $\mathsf{AF}\varphi$ ($\mathsf{EF}\varphi$).

The operators of CTL can be nested. Thus, the property $\mathsf{AG}(\varphi \Rightarrow \mathsf{AF}\psi)$ means that in all reachable states of the system either (a) φ is not true, or (b) φ holds and

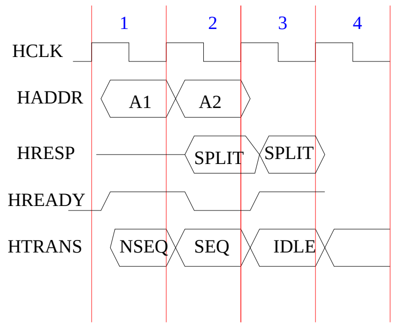

Figure 3. A possible Transfer Cancellation

along all outgoing paths ψ holds eventually. Such a property is useful for specifying "guaranteed response" or lack of starvation. For example in the AMBA protocol, we can specify and check the property AG$(HBUSREQ_m \Rightarrow$ AF $HGRANT_m)$. Here the master m requests bus access by asserting $HBUSREQ_m$ and the arbiter grants bus access to m by asserting the signal $HGRANT_m$. This property states that whenever master m requests bus access, it is always eventually granted bus access by the arbiter.

Transfer Cancellation Due to the pipelined nature of the AMBA protocol, a master initiates transfer $i+1$ before receiving the response for transfer i. Consequently, if transfer i resulted in a SPLIT or RETRY response from the slave, the master *must* cancel the transfer $i+1$. This is because transfer $i+1$ was initiated with the assumption that transfer i would result in an OKAY response from the slave. One such scenario (showing a SPLIT response from the slave and the cancellation of the second transfer in a burst) is captured by the timing diagram in Figure 3. The signal $HTRANS$ denotes the status of the current transfer; it is IDLE if transfer is not taking place, *NSEQ* for the first transfer in a burst and *SEQ* for the subsequent transfers.

In Figure 3, a master m is granted bus access and transfers in cycle 1. The slave issues a split response to this transfer in cycle 2. However, by this time m has driven the next address $A2$. So the slave issues another split response in cycle 3. Furthermore, the master also does not drive any new addresses on the address bus. This is shown as HTRANS = IDLE in cycle 3.

The current AMBA specification thus forces a slave to set HRESP = SPLIT for two cycles in case of a split response. This allows the cancellation of the transfer $A2$ in Figure 3 (which was started even before the response for $A1$ was obtained). This requirement is typical of a pipelined bus protocol. It captures the protocol's attempt to correctly roll-back transfers which were speculatively initiated ($A2$ was initiated based on the speculation that $A1$ will produce an OKAY response). We now discuss how subtle corner cases can occur in spite of such a requirement.

Checking for no-starvation The pipelined nature of the AMBA bus makes the cancellation (say due to a SPLIT) and resumption of transfers difficult. In the above, we have only considered the cancellation of transfers. A crucial design invariant in a bus protocol is the non-starvation of all bus masters, that is

$$\text{AG}\ (HBUSREQ_m \Rightarrow \text{AF}\ HGRANT_m)$$

Thus, if a master m requests the bus (by asserting the signal $HBUSREQ_m$) then it is eventually granted bus access. Verifying this property is important in AMBA, since there might be corner cases in the protocol which prevent the resumption of bus access by a master which has been split. Using the SMV model checker, we found a counterexample to the above property: a scenario in which starvation of a split master is possible in the AMBA protocol.

Before presenting the starvation scenario, let us explain some relevant features of our modeling. A bus master m which has been split by a slave s can suffer from starvation for many reasons:

- the slave s never informs the arbiter that it is now able to service m, or
- even after s has informed its ability to service m, the arbiter ignores the bus request from m forever.

However, the above situations are not caused by any error in the bus protocol. Instead they are caused by an implementation error in the slave or an unfair arbitration policy used in the arbiter. When we model check the AMBA protocol, our aim is to find subtle bugs in the protocol itself, assuming that the arbiter is fair and the masters/slaves conform to the protocol specification. To prove the no-starvation property AG $(HBUSREQ_m \Rightarrow$ AF $HGRANT_m)$ for any master m, we would like to instruct the model checker to assume certain properties of the arbiter/slave. This can be done in SMV by specifying

```
using fair, slave_live prove no_starve;
assume fair, slave_live;
```

where `no_starve` is the no-starvation property that we want to prove.

`fair` is an assertion stating fairness of the arbiter. In particular the arbiter maintains a list of masters which have been split (and have not recovered). Requests from such masters are *masked* until the slave informs the arbiter that it is able to service these masters. The arbiter maintains a $mask$ array; $mask_m$ is true if and only if the arbiter believes that m has been split but has not recovered. Thus the fairness property of the arbiter requires

$\mathsf{AG}(HBUSREQ_m \wedge \neg mask_m \Rightarrow \mathsf{AF}\ HGRANT_m)$ for any master m. In other words, if m is requesting the bus and the arbiter believes that m has not been split, then m is eventually granted bus access.

`slave_live` is an assertion which requires the slave to always eventually recover from a split. Any slave s maintains a list of masters that it has split so far. Thus, the slave maintains a *split* array; $split_m$ is true if the slave has split master m and is not yet ready to service m. When the slave is ready to service master m, it asserts the $HSPLIT_m$ signal. The `slave_live` property is $\mathsf{AG}(split_m \Rightarrow \mathsf{AF}\ HSPLIT_m)$.

Note that SMV does not prove the assertions `fair` and `slave_live`. Instead it uses them to prove the no-starvation property. This removes the need to model the specific details of the arbitration policy or the slave implementation in our SMV code. This feature is extremely useful for protocol verification where we want to verify the protocol by assuming certain desirable properties of the component implementations (which participate in the protocol).

Verification Statistics In a bus configuration with 2 masters and 1 slave, SMV finds a violation of no-starvation in 0.17 seconds. As far as the memory usage is concerned, the number of Binary Decision Diagram (BDD) nodes allocated during the verification run is 19149. The verification was conducted using a Linux version of Cadence SMV in a Pentium IV 1.3 GHz workstation with 1 GB of main memory.

The SMV description of a simplified version of AMBA AHB is available at `http://www.comp.nus.edu.sg/~abhik/software/amba/ahb.smv` This description does not model many features of AHB (such as burst transfers of various lengths $2, 4, \ldots$), but is detailed enough to demonstrate the no-starvation scenario presented in this paper. The SMV description contains only 144 lines of source code. The counter-example generated by Cadence SMV is also available online at `http://www.comp.nus.edu.sg/~abhik/software/amba/ahb.out`

A starvation scenario Figure 4 shows a bus starvation scenario in the AMBA protocol. The timing diagram captures the relevant part of the counter-example computed by the SMV model checker.

In Figure 4, master $m1$ transfers a two beat burst in cycles 1 and 2. In cycle 3, master $m2$ gains bus access and starts a burst. The HMASTER signal captures the master which currently has bus access. Thus, HMASTER = $m1$ in clock cycles 1 and 2, and HMASTER = $m2$ in cycle 3. The response for the address transmitted in cycle 2 (by master $m1$) comes in cycle 3 due to the pipelined nature of AMBA. If this is a split response (as is shown in Figure 4), then the

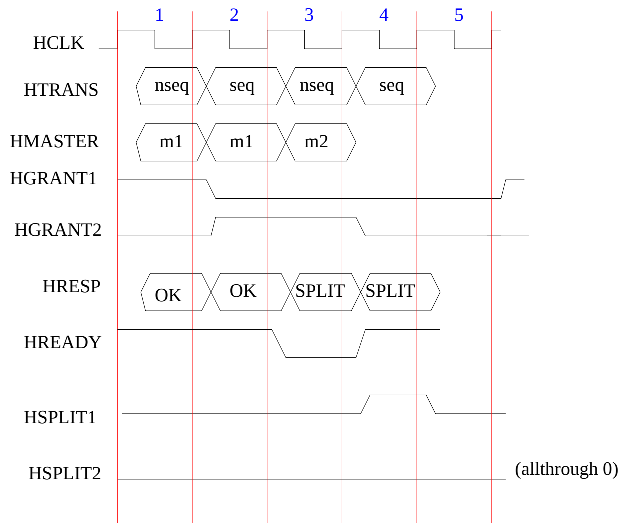

Figure 4. A Starvation scenario in AMBA

slave records the fact that master $m1$ has been split. Meanwhile the arbiter snoops on the split response sent by the slave in cycle 3. In the next clock cycle, that is in cycle 4, the arbiter however sets $mask_{m2}$ instead of $mask_{m1}$. This is because HMASTER changed to $m2$ in cycle 3. By snooping on the split response (from the slave) and checking the HMASTER signal, the arbiter believes that $m2$ has been split. This is the source of the problem, since the slave believes that $m1$ has been split. Subsequently, the slave becomes ready to service $m1$ and $m1$ accesses the bus. However, the arbiter never grants bus access to master $m2$ (even if $m2$ requests bus access), since it believes that $m2$ has been split. On the other hand, the slave has no record that $m2$ has been split, so it never asserts HSPLIT2 to inform the arbiter that it is ready to service $m2$. This leads to a starvation of the bus master $m2$, that is, a violation of the property $\mathsf{AG}(HBUSREQ_{m2} \Rightarrow \mathsf{AF}\ HGRANT_{m2})$.

After finding the above counter-example trace from SMV, we cross-checked it against the AMBA protocol specification. We found that this behavior is a potential bug arising from an incompleteness in the current AMBA specification. The error is introduced by the manner in which the arbiter checks for a split response. In Figure 4, the arbiter finds out in clock cycle 4 that a split response occurred in cycle 3. However, it is incorrect for the arbiter to consider the HMASTER of cycle 3 (which is m2). Note that the split response in cycle 3 is the slave's response to the address transferred in cycle 2 (due to the pipelined nature of the AMBA bus). Therefore, the arbiter should use the HMASTER of cycle 2 to infer that $m1$ has been split. In other words, the arbiter always needs to keep track of not only the current HMASTER, but also the HMASTER of previous cycles (whose response is yet to arrive). This needs to be explicitly clarified in the AMBA protocol specification to avoid the starvation scenario depicted in Figure 4.

Note that in a pipelined protocol, all communicating agents should always keep track of all incomplete transfers which have started. In the above scenario, the masters/slave keep track of the incomplete transfers in every cycle. However, in cycle 3 the arbiter failed to record that even though $m1$ has stopped transferring, the response for $m1$'s last transfer is yet to arrive. This resulted in the starvation scenario.

5 Discussions

In this paper, we presented our experience in verification of a system-on-chip bus protocol. Verification techniques such as model checking are useful in automatically detecting subtle corner cases in the protocol specification. This was indeed the case for the AMBA AHB protocol, where the starvation scenario found would be very hard to detect without automated formal verification. Our work shows the intricacies in designing pipelined bus protocols and the importance of formally verifying these protocols.

6 Acknowledgments

This research was partially supported by a National University of Singapore research project R-252-000-103-112.

References

[1] F. Aloul and K. Sakallah. Efficient verification of the PCI local bus using boolean satisfiability. In *International Workshop on Logic Synthesis (IWLS)*, 2000.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(9), 1986.

[3] Cadence Berkeley Laboratories, Free download from `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`, California, USA. *The SMV Model Checker*, 1999.

[4] P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-core based system-on-chip designs. In *IEEE ASIC SOC Conference*, 1999.

[5] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2), 1986.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] IBM. *White Paper on CoreConnect Bus Architecture*, 1999.

[8] A. Mokkedem, R. Hosabettu, M. Jones, and G. Gopalakrishnan. Formalization and analysis of a solution to the PCI 2.1 bus transaction ordering problem. *Formal Methods in System Design*, 16, 2000.

[9] M. Oliveira and A. Hu. High-level specification and automatic generation of IP interface monitors. In *ACM/IEEE Design Automation Conference*, 2002.

[10] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *IEEE Design Automation Conference (DAC)*, 1998.

[11] K. Shimuzu, D. Dill, and C.-T. Chou. A specification methodology by a collection of compact properties as applied to the Intel Itanium processor bus protocol. In *Correct Hardware Design and Verification Methods (CHARME), LNCS 2144*, 2001.

[12] K. Shimuzu, D. Dill, and A. Hu. Monitor-based formal specification of PCI. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000.