Verifying IP-Core based System-On-Chip Designs *

Pankaj Chauhan, Edmund M. Clarke, Yuan Lu and Dong Wang
Carnegie Mellon University, Pittsburgh, PA 15213
{pchauhan, emc, yuanlu, dongw}+@cs.cmu.edu

June 20, 1999

Abstract

We describe a methodology for verifying system-on-chip designs. In our methodology, the problem of verifying system-on-chip designs is decomposed into three tasks. First, we verify, once and for all, the standard bus interconnecting IP Cores in the system . The next task is to verify the *glue* logic, which connects the IP Cores to the buses. Finally, using the verified bus protocols and the IP core designs, the complete system is verified. To illustrate our methodology, we verify the PCI Local Bus, a widely used bus protocol in system-on-chip designs. We demonstrate various modeling and verification techniques for buses by modeling the PCI Local Bus with the symbolic model checker SMV. We have found two potential bugs in the PCI bus protocol specification that await confirmation of the PCI Special Interest Group(PCI-SIG).

1 Introduction

Hardware designs have reached a mammoth scale today, with over ten million transistors integrated on a single chip. This breakthrough in technology has, in fact, reached the point, where it is hard to design a complete system from scratch. Industry has already started designing ASICs from a large repertoire of Intellectual Property Components or IP Cores sold by many vendors. System-on-chip designs usually involve the integration of *heterogeneous* components on a standard bus. These components may require different protocols or have different timing requirements. Moreover, designers often do not have complete knowledge of the implementation details of each component. For example, vendors may want to protect their IP Cores by only providing interface specifications. Consequently, the validation of such designs is becoming more and more challenging. In this paper, we outline a new methodology for formally verifying IP Core based, system-on-chip designs.

An IP Core based system can be viewed as a collection of various IP cores, with interconnecting buses running among them (see Figure 1). Since the cores are obtained from different vendors, there is a need for standard buses to connect them. We also envision some kind of interface logic, which we call *glue*, to connect IP Cores to the standard buses. In some cases IP Cores are designed to be compliant to a standard bus protocol and can be connected directly to the bus without glue. Bridges are used to extend such systems in a hierarchical fashion by connecting buses.

IP Cores are often pre-validated. This increases the confidence of system designer in third party IP Cores. The validation of IP Cores must be part of the IP Core design itself. So in this scenario, where we have a) pre-verified IP Cores with certain guarantees and confidence, b) a standard bus protocol, and c) IP Core specific glue to connect cores to the bus, we can decompose the task of verifying system-on-chip designs into three parts as follows.

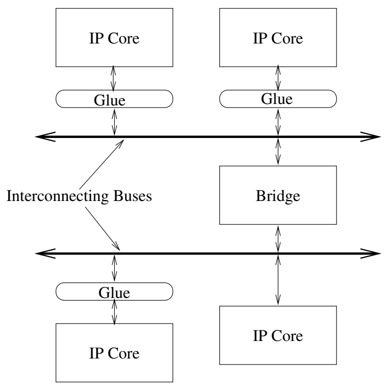

Figure 1: A typical IP Core based System

1. Verify the interconnecting buses and bus bridges;
2. Verify the IP Core specific glue logic;
3. Given the verification guarantees of interconnecting buses and IP Cores, verify the complete system.

Since the bus protocol is standard, it needs to be verified once and for all. Glue logic is IP Core specific. If we have a collection of protocols for IP Cores, then we can design an abstraction of the glue between the standard bus and each IP Core protocol. This abstract model is designed once. Then, we intend to check if the actual glue implementation *refines* the abstract model of the glue [7]. Thus, we have reduced the complexity of verifying the glue to checking refinement. When this is completed for all IP Cores and their glues, we can proceed to the third step.

Experience in industry with IP Core based ASIC designs shows that most of the bugs are found in the bus or glue logic. To our knowledge, there is still no agreement on a standard bus protocol for system-on-chip designs. However, the PCI Local Bus protocol [12, 13, 14] is widely accepted by many

*This research is sponsered by the Gigascale Silicon Research Center (GSRC). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the GSRC.

microprocessor based systems (eg. Pentium and Alpha) and IP Core companies. Therefore, we focus in this paper on verifying system-on-chip designs using the PCI Local Bus. This will provide insight into questions like, what basic functionality is required of the buses, what kind of standard interfaces are needed for IP Core based designs, and how glue logic may be designed and verified for heterogeneous IP Cores. We have formally verified the correctness of the PCI bus protocol using symbolic model checking [5].

In many cases, bus protocols can be verified with current formal verification techniques as demonstrated by [4] and [6]. We concentrate more on the functional properties of the PCI local bus and bridges rather than performance issues. A formal treatment of PCI bus performance is given by Campos, et al. in [4]. In a recent paper [11], theorem proving techniques have been used to validate a proposed solution for a bug in the PCI bus protocol, but this approach requires considerable expertise in modeling the bus and is not easily automated.

The rest of the paper is organized as follows: In Section 2, we introduce Computation Tree Logic and the symbolic model checker SMV. In Section 3, we provide an overview of the PCI local bus protocol and systems based on it. In Section 4, we illustrate a systematic approach to verify the PCI bus and bus bridges. We describe modeling and specification techniques, which can be applied to verify other bus protocols as well. In Section 5, we present our experimental results, notably the description of two bugs we have found in this widely used bus protocol. Finally in Section 6, we summarize our experience and present a few thoughts on how to reach our ultimate goal of verification of complete system-on-chip designs.

2 Symbolic Model Checking

Symbolic model checking is a powerful technique for formally verifying finite state concurrent systems automatically [9]. The task of symbolic model checking can be broken down in three phases *Modelling, Specification* and *Verification*. The first task is to represent system under consideration in a precise model that can be accepted by a model checking tool. Often we need to use abstraction to eliminate irrelevant or unimportant details from the design due to limits on time and memory. We used SMV [9], a temporal logic model checker based on binary decision diagrams (BDDs) [1]. SMV has its own built-in dataflow-oriented hardware description language for modeling and accepts specifications expressed in the CTL temporal logic [5]. SMV extracts a finite-state model as a state transition graph from an SMV program. BDDs are used to represent transition relations and to manipulate them. This representation has been the major contribution to the success of symbolic model checking [2, 9].

In the specification phase, we write the properties in a branching-time temporal logic called CTL ("Computation Tree Logic") [5]. Formulas in CTL are built from three components: atomic propositions, boolean connectives, and *temporal operators*. Atomic propositions refer to the values of individual state variables. The boolean connectives are conjunction, disjunction and negation ($\wedge$, $\vee$, $\neg$). Each temporal operator consists of two parts: a path quantifier ($\mathbf{A}$ or $\mathbf{E}$) and a temporal modality ($\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$ or $\mathbf{U}$). The quantifier indicates whether the operator denotes a property that should be true of all execution paths from a given state or whether the property need only hold on some path. The modalities describe the ordering of events in time along an execution path and have the following intuitive meanings:

1. $\mathbf{F}\,\varphi$ ("φ holds sometime in the future") is true of a path if there exists a state on the path for which the formula φ is true.
2. $\mathbf{G}\,\varphi$ ("φ holds globally") means that φ is true at every state on the path.
3. $\mathbf{X}\,\varphi$ ("φ holds in the next state") means that φ is true in the second state on the path.
4. $\varphi\,\mathbf{U}\,\psi$ ("φ holds until ψ holds") means that there exists some state on the path for which ψ is true, and for all states preceding this one, φ is true.

Each formula of the logic is either true or false in a given state. An atomic proposition is true in a state if the state variable that it refers to has the appropriate value. The truth of a formula built from boolean connectives depends on the truth of its subformulas in the usual way. A formula whose top level operator is a temporal operator with a universal (existential) path quantifier is true whenever all paths (some path) starting at the state have the property required by the operator's modality. A formula is true of a system if it is true for all the initial states of the system.The following examples illustrate the expressive power of the logic.

1. $\mathbf{AG}(\mathit{Req} \rightarrow \mathbf{AF}\,\mathit{Ack})$: it is always the case that if the signal *Req* is true, then eventually *Ack* will also be true.
2. $\mathbf{AG}\,\mathbf{AF}\,\mathit{DeviceEnabled}$: *DeviceEnabled* holds infinitely often on every computation path.
3. $\mathbf{AG}\,\mathbf{EF}\,\mathit{Restart}$: from any state, it is possible to get to the *Restart* state.
4. $\mathbf{AG}\big(\mathit{Send} \rightarrow \mathbf{A}(\mathit{Send}\,\mathbf{U}\,\mathit{Recv})\big)$: if *Send* holds, then eventually *Recv* is true, and until that time, *Send* remains true.

The final task of verification is an automatic process. Every CTL formula has a fixed point characterization that can be used to find the set of states satisfying the formula [3].

3 PCI Local Bus

The PCI Local Bus [12, 13, 14] is a high performance, synchronous bus architecture that can transfer 32-bit or 64-bit data. Its primary goal is to establish an industry standard and optimize for direct silicon (component) interconnection with minimum glue logic required. It supports most processor designs and connects various types of devices on a chip. Bridges are used to extend the PCI bus based systems.

A typical PCI bus transaction is demonstrated in Figure 2. The request for a transaction starts when a subsystem asserts its request line REQ#. It then waits until being granted the bus by the arbiter by asserting the corresponding GNT# line. This phase is known as the *arbitration phase*. The transaction begins when signal FRAME# is asserted. In the first clock after asserting FRAME#, address is put on the data/address multiplexed lines in the *address phase* and the command lines carry the transaction-type. All target devices listen to this address and if the address maps to their address space, they assert their DEVSEL# lines, indicating they are present on the bus. The master then asserts the signal IRDY#, meaning that it is ready for data transfer. The bus target asserts its TRDY# signal to indicate that the target is ready for data transfer. Data transfer occurs when both IRDY# and TRDY# are asserted, which is known as one *data phase*. A transaction can have more than one *data phase*, and wait cycles can be inserted between data phases by the master (target) by deasserting the IRDY# (TRDY#) signal. One clock cycle before the end of the data transfer phase, the FRAME# signal is deasserted. In the next cycle both IRDY# and TRDY# are deasserted, and the bus goes back to the idle state.

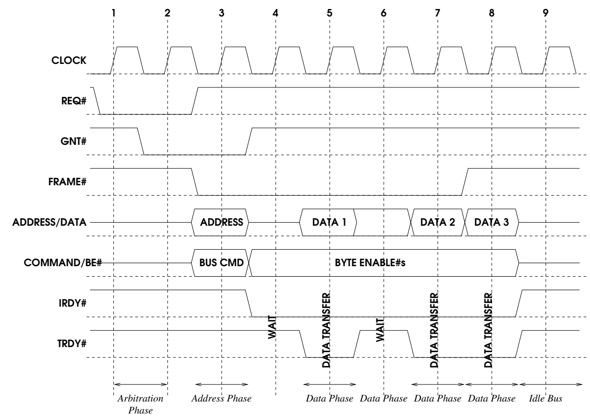

Figure 2: A typical PCI bus transaction

The PCI bus requires a *fair* central arbiter, which implies that every master should be served. Note that apart from this requirement, the arbitration algorithm is not part of the PCI bus specification. The arbiter may *park* the bus at some selected master or allow the bus to float. Since the PCI bus is a high performance bus, there are strict timing requirements on various events, like number of wait states, latency for a target asserting its selection line, arbitration latency, etc.

PCI bridges are used to connect two PCI buses in a transparent manner. PCI devices must follow certain allowable transaction orderings in order to satisfy the Producer-Consumer model discussed in the next section. Bridges can *post* certain transactions in order to improve performance, meaning certain write transactions can be buffered in the bridges without being completed at the target, but the master is not required to wait until it completes. In order to observe the producer-consumer model, there are certain restrictions on bridges regarding transaction-posting. Bridges may also attempt to convert one or more transactions into a large transaction for improved performance by *combining, merging or collapsing* [12].

4 Verifying PCI-bus

We verified the PCI bus protocol in two steps. In the first step, we looked at properties related to the protocol without bridges. In the second step, we modeled a PCI bridge and verified a producer-consumer model for bus transactions with bridges. Figure 3 shows one configuration that we modeled to verify some bus properties. The arbiter is safe and fair. The dummy-master is an abstracted master which has very restricted functionality and is used only for checking arbitration properties. The master and the target are capable of carrying out PCI sequences, e.g. fast back to back transactions, bus parking, burst transactions, latency requirements, transaction termination, etc. The state machines of PCI master and PCI target are given in the PCI specification [12]. In order to handle multiple masters and targets on a single bus and avoid the state explosion problem, we used some techniques including abstraction, assume-guarantee reasoning, symmetry and case analysis. We developed a single bus model in CBL SMV [10] with 5 masters and 5 targets. In this model, there is symmetry within the masters, as well as the targets. For a bus driving property, we perform a case analysis on the actual master and the target that are active on the bus, then use symmetry to reduce the number of proof obligations. In order to prove these properties, we first proved the following non-interference lemmas.

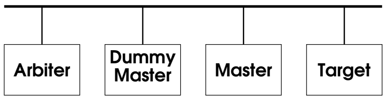

Figure 3: Configuration for verifying bus properties

- Inactive masters and targets can not drive the bus.
- Only one master and one target can be active at a time.
- An active master must stay active until one target becomes active.
- An inactive target can not become active until it has an address hit.
- If a master and a target are active, then the master must remain active until the target becomes inactive.
- If one master starts a transaction with address k, then it will remain active until the target with address k becomes active.

Using these lemmas, we have proved the following bus driving properties:

- Once STOP# is asserted, FRAME# should be deasserted as soon as IRDY# is asserted.
- If not already deasserted, TRDY#, STOP#, and DEVSEL# must be deasserted the clock following the completion of the last data phase and must be tri-stated in the next clock.
- The target cannot drive data on the bus in the cycle immediately after a read transaction begins.
- A master or a target should drive the data bus correctly in a data phase.

We verified properties about transaction termination, arbitration, latency requirements, etc. These properties primarily followed various *must* hold statements from the specification. For example consider the following property: $\mathbf{AG}(\mathit{bus.FRAME\#} \rightarrow \mathbf{AX}(\neg \mathit{bus.FRAME\#} \rightarrow \mathit{bus.IRDY\#}))$. It says that when FRAME# is first asserted (active low), IRDY# should stay deasserted. SMV determined that this property is true.

The configuration given in Figure 4 illustrates the *Producer-Consumer* model. In this model, the Producer writes all data, sets the flag and waits for completion status; while the Consumer waits until it finds the flag set, then it resets the flag, consumes the data, and writes the completion status code. When the Producer finds the completion status code, it resets the code and the sequence repeats. Since bridges can post write transactions, it is important to see that this model is satisfied, i.e. the Consumer never sees that the flag is set when the data is not written. The bridge obeys all PCI ordering rules. We will demonstrate that it satisfies the Producer-Consumer model as well. We modeled a PCI bridge according to the PCI bridge specs [13]. The state machine of PCI bridge essentially is a composition of four state machines, the primary master and target and the secondary master and target (each bridge has a primary bus and a secondary bus). Since we have already proved many properties about basic bus protocol, we have abstracted away some of the functionality of the PCI masters and targets in this configuration, e.g. bus parking, burst transactions, target delay, etc. We need to represent only three distinct values of data 0,1 and 2 for verifying this model [9], hence the data bus should be at least 2 bits wide. The address bus is 2 bits wide. Bridges also have finite data buffers for posting data.

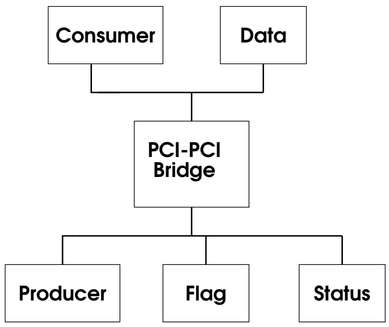

Figure 4: Producer-Consumer model

We use a simple abstraction to establish the correctness of the PCI bridge transaction ordering rules. We show that if a symbolic data value x is generated by the producer, then a unique copy of x will be received by the consumer. The variable *flag* in the producer-consumer model indicates the status of the data generated by the producer: flag = 0 means the producer has not written new data; flag = 1 means that the producer has written the data value x; flag = 2 means the producer has written a value different from x. In our model, the producer only generates one instance of the data value x. The producer-consumer model requires that whenever the consumer sees flag = 1, it should then receive x. The following three properties capture the correctness of this behavior (c denotes consumer).

1. $\mathbf{AG}((\text{c.chkFlag} \wedge \text{flag=1}) \rightarrow \mathbf{A}[(\text{!c.statusFlag} \wedge \text{data} \neq \text{x})\mathbf{U}(\text{c.readData} \wedge \text{data=x})])$: The consumer receives x after seeing flag = 1.
2. $\mathbf{AF}(\text{c.chkFlag} \wedge \text{flag=1})$: The consumer eventually sees flag = 1.
3. $\mathbf{AG}((\text{c.chkFlag} \wedge \text{flag=1}) \rightarrow \mathbf{AX}\,\mathbf{AG}(\text{c.chkFlag} \rightarrow \neg(\text{flag=1})))$: The consumer sees flag = 1 at most once.

5 Experimental Results

We verified the PCI bus protocol in two steps a) a single PCI bus without bridges and b) a bus with bridges. The single bus model includes a master, a target, a dummy master and a bus arbiter. The dummy master is used to check arbitration properties. Masters and targets are modeled based on the respective state machines given in the Appendix of the *PCI Specification Revision 2.2* [12]. The model is about 1000 lines of SMV code and 111 BDD variables. We verified twenty-three major properties using SMV for this setup, which took about three and half hours on a Pentium-Pro 200MHz machine with 1G memory. During the process we discovered two potential bugs in the PCI protocol specification and wait for confirmation from the PCI SIG group. Both errors are due to the inconsistencies between the specification and given state machines in the Appendix.

The first error occurs because the target transition condition is set incorrectly: in the target state machine, the target makes a transition from B_BUSY to IDLE *when* FRAME# is deasserted and D_done is asserted. However, when the master starts a transaction with single data phase, target goes from B_BUSY to IDLE instead of S_DATA in *all* cases (see Figure 5). This error is caught by the following CTL formula: $\mathbf{AG}((\mathit{m.req} \wedge \mathit{m.data\_cnt=1}) \rightarrow \mathbf{A}(\mathit{m.req} \wedge \neg \mathit{t.ack}\,\mathbf{U}\,\mathit{m.timeout}))$ This formula means that whenever the master requests a transaction with single data phase, the target never acknowledges before the master times out, i.e. the target never goes to the S_DATA state. We verified that this formula is true, which is inconsistent with the standard. In the second error, the Specification requires that *once a master has asserted IRDY#, it*

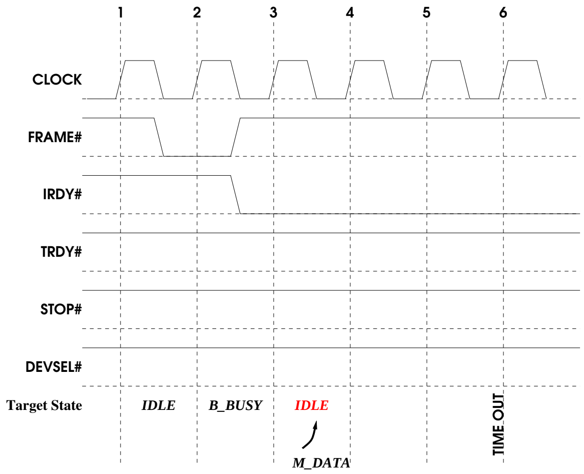

Figure 5: Illustration of the first bug

can't change IRDY# or FRAME# until the current data phase completes. But the implementation of FRAME# in the state machine doesn't satisfy this requirement in the following scenario. Let us assume that IRDY#, FRAME# are asserted, and GNT#, TRDY# are disasserted in the current cycle. If the master timer expires in the next cycle, then FRAME# is always deasserted , even though IRDY# is still asserted. This clearly conflicts with the specification. SMV came up with a counterexample trace to illustrate this inconsistency.

On an UltraSparc 248MHz machine, it took 46 minutes and 13M BDD nodes to verify all the lemmas and properties for multiple masters/targets. As a comparison, after taking about 2Gb memory and 5 hours, the first property could not be verified without techniques mentioned in Section 4.

In the next step, we modeled a multiple bus system with a bridge (see Figure 4) and verified the correctness of the producer-consumer model by checking the three major properties described in Section 4. In Table 5, we show our various statistics for two implementations of PCI bus with bridges that have different numbers of buffers in bridge.

# Buffers	BDD Var	Time	BDD Nodes
4	120	982 s	12,075,532
6	142	8.8 h	15,080,273

Table 1: Experimental results for verifying PCI bridge

6 Conclusions and Future Work

In this paper, we have proposed a new methodology for verifying system-on-chip designs. As the first example, we have verified the PCI Local Bus protocol using the symbolic model checker SMV. Significantly, we have found two potential bugs in the standard PCI bus specification. In our experience, formally verifying the functionality of bus protocols is feasible using current model checking techniques.

In order to achieve our ultimate goal of system-on-chip verification, we will focus in the future on verifying the glue logic involved in such designs. We also intend to verify more complex industrial bus designs using the methodology we have proposed in this paper. Finally, we are interested in high-level specification of bus protocols for verification purposes.

Acknowledgements

We are especially thankful to Kenneth McMillan of Cadence Berkeley Labs for initially suggesting us the idea of verifying IP Core based system-on-chip designs.

References

[1] R. E. Bryant. “Graph-based algorithms for boolean function manipulation”, *IEEE Transactions on Computers*, C-35(8), 1986.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, J. Hwang. “Symbolic model checking: 10^{20} states and beyond”, *Proc. 5th Ann. Symp. on Logic in Compu. Sci.,* June 1990.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, “Sequential circuit verification using symbolic model checking”, *Proc. 27th ACM/IEEE Design Automation Conf.,* June 1990.

[4] S. Campos, E. Clarke, W. Marrero, M. Minea. “Verifying the performance of the PCI Local Bus using Symbolic Techniques”, *IEEE Intl. Conf. in Comp. Design*, Oct. 1995.

[5] E. M. Clarke, E. A. Emerson, A. P. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”, *ACM Transactions on Programming Languages and Systems,* 8(2):244-263, 1986.

[6] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness. “Verification of the Futurebus+ Cache Coherence Protocol”, *Carnegie Mellon University Technical Report no. CMU-CS-92-206,* Oct. 1992.

[7] R. Kurshan. *Computer Aided Verification*, Kluwer Academic Publishers, March 1993.

[8] D. E. Long. “Model Checking, Abstraction and Compositional Verification”, *Carnegie Mellon University publication CMU-CS-93-178,* July 1993.

[9] K. L. McMillan. “Symbolic Model Checking: An Approach to the State Explosion Problem”, *Carnegie Mellon University publication CMU-CS-92-131,* May 1992.

[10] K.L. McMillan. “A compositional rule for hardware design refinement“, *Computer Aided Verification,* pp 24-35, June 1997.

[11] A. Mokkedem, R. Hosabettu, and G. Gopalakrishnan. “Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem”,*FMCAD, Nov. 98.*

[12] PCI Special Interest Group. *PCI Local Bus Specification Rev 2.2,* Dec. 1998.

[13] PCI Special Interest Group. *PCI to PCI Bridge Architecture Specification Rev 1.1,* Dec. 1998.

[14] E. Solari, G. Willse. *PCI Hardware and Software - Architecture and Design,* Annabooks, 1998.