

**Formal Verification of the PCI Local Bus: A Step Towards IP Core  
Based System-On-Chip Design Verification**

by

Dong Wang

Submitted to the  
Department of Electrical and Computer Engineering  
in partial fulfillment of the requirements for the degree of  
Master of Science in Electrical and Computer Engineering

at

CARNEGIE MELLON UNIVERSITY

May 1999

© Dong Wang, 1999. All rights reserved.

The author hereby grants to CMU permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .....  
Department of Electrical and Computer Engineering  
May 3, 1999

Certified by .....  
EDMUND M. CLARKE, Thesis Advisor  
FORE Systems Professor of Computer Science and Professor of Department of Electrical  
and Computer Engineering(C)

Certified by .....  
Randal E. Bryant, Thesis Reader  
President's Professor of Computer Science with courtesy appointment in Electrical and  
Computer Engineering

Accepted by .....  
Robert M. White  
University Professor and Head, Department of Electrical and Computer Engineering;  
Professor of Engineering and Public Policy

**Formal Verification of the PCI Local Bus: A Step Towards IP Core Based  
System-On-Chip Design Verification**

by

Dong Wang

Submitted to the Department of Electrical and Computer Engineering  
on May 3, 1999, in partial fulfillment of the  
requirements for the degree of  
Master of Science in Electrical and Computer Engineering

**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, temporal properties about the complete system are deduced. 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 specification that await confirmation of the PCI Special Interest Group(PCI-SIG).

Thesis Advisor: EDMUND M. CLARKE

Title: FORE Systems Professor of Computer Science and Professor of Department of Electrical and Computer Engineering(C)

Thesis Reader: Randal E. Bryant

Title: President's Professor of Computer Science with courtesy appointment in Electrical and Computer Engineering

## Acknowledgments

The preparation of this report has benefited greatly from my advisor, Ed Clarke. Ed got me started in this field and has opened my eyes to many challenging problems in model checking. His encouragement and help are among the most important reasons behind the completion of this report.

I would like to thank Pankajkumar Punjabrao Chauhan and Yuan Lu for working at the same project. Their contributions to this report are phenomenal. I especially enjoyed the counterless talks with Yuan, which has significantly improved the quality of this work.

I would like to thank Sergey Berezin for numerous insightful conversations on topics in formal verification. I would like to thank every member of our research group for their help.

Special thanks to Kenneth McMillan of Cadence Berkeley Labs for initially suggesting us the idea of verifying IP Core based system-on-chip designs.

# Contents

0.1	Introduction . . . . .	1
0.2	Symbolic Model Checking . . . . .	4
0.2.1	Binary Decision Diagram . . . . .	4
0.2.2	Temporal Logic Model Checking . . . . .	5
0.2.3	Implementation Relation between Models . . . . .	7
0.3	PCI Local Bus . . . . .	8
0.3.1	PCI Signals and Transactions . . . . .	8
0.3.2	PCI Bridge . . . . .	14
0.4	Verifying PCI Bus . . . . .	18
0.4.1	Single PCI Bus . . . . .	18
0.4.2	PCI Bus with Multiple Masters and Targets . . . . .	23
0.4.3	Verifying PCI Bridge . . . . .	26
0.5	Experimental Results . . . . .	28
0.6	Conclusions and Future Work . . . . .	32

## 0.1 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. This report outlines 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 0-1). 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.

1. Verify the interconnecting buses and bus bridges based on the abstract models developed

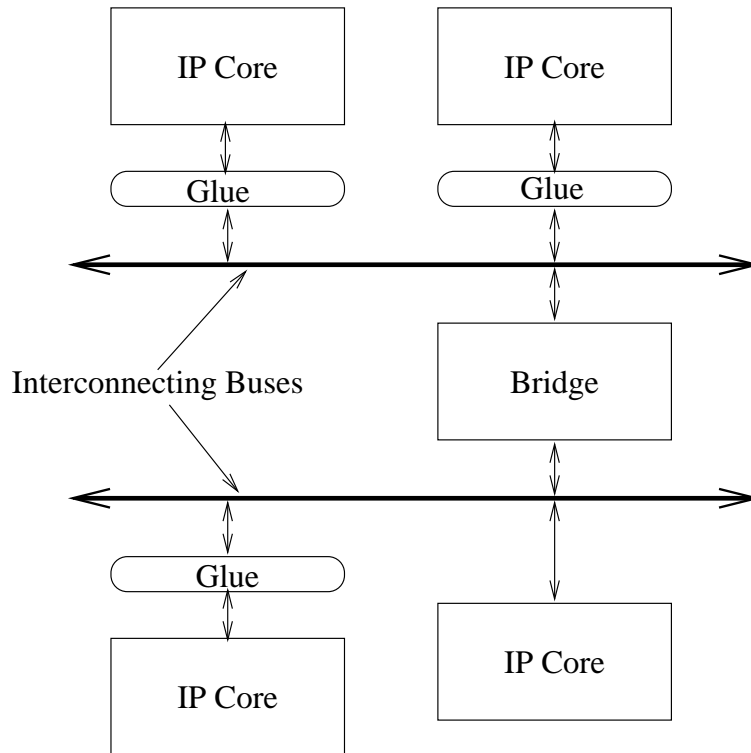


Figure 0-1: A typical IP Core based System

according to the bus specifications;

2. Verify the IP Core specific glue logic with respect to the pre-verified abstract glue models;
3. Given the verification guarantees of interconnecting buses and IP Cores, deduce properties about the actual system.

Since the bus protocol is standard, it needs to be verified once and for all. One of the major task in this process is the development of bus specific abstract models including master and target state machines, and master and target back-end models. These models should be general enough to incorporate all the behaviors of the interesting verification-related bus transactions. Temporal properties verified in this stage should be focused on the subsets that can be preserved by the second stage[15, 13, 11]. 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 [12]. Thus, we have reduced the complexity of verifying the glue to checking refinement. There is efficient algorithm that is both linear in the sizes of the state spaces of both implementation and specification machines[10]. Compositional reasoning techniques can be used to check refinement and provide abstract environments when checking part of the design[15]. When this is completed for all IP Cores and their glues, we can proceed to the third step, where temporal properties about the whole system can be easily deduced. The verified complete system should be free of deadlocks and livelocks[9]. We envision the following design language which facilitate the application of our verification methodology:

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 [18, 19, 20] is widely accepted by many microprocessor based systems (eg. Pentium and Alpha) and IP Core companies. Therefore, the report focuses 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 [6].

In many cases, bus protocols can be verified with current formal verification techniques as demonstrated by [5] and [8]. 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 [5]. In a recent paper [17], 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. ‘

## 0.2 Symbolic Model Checking

Model Checking[6] is an automatic technique to check an implementation described as a state transition system satisfies the specification expressed in temporal logic formulas or another state transition system. Symbolic Model Checking uses ROBDD[1] as the underlying representation, which makes it practical for many applications.

### 0.2.1 Binary Decision Diagram

Ordered binary decision diagrams (BDDs) are a canonical representation form for boolean formulas [1]. They are often substantially more compact than conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. BDD is a directed acyclic graph which can be obtained from the ordered decision tree by the following two steps:

- Combine any isomorphic subtrees into a single tree.
- Eliminate any nodes whose left and right children are isomorphic.

An efficient BDD package[2] normally include a unique table, an operation cache. Boolean operations can be implemented based on ITE operator. Suppose  $x$  is a boolean variable, then

$$ITE(I, T, E) = x \wedge ITE(I_x, T_x, E_x) \vee x' \wedge ITE(I_{x'}, T_{x'}, E_{x'})$$

The time complexity is  $O(|I| \cdot |T| \cdot |E|)$ .

OBDDs are used for obtaining concise representations of relations over finite domains. If  $R$  is  $n$ -ary relation over  $0,1$  then  $R$  can be represented by the OBDD of its characteristic function  $f_R(x_1, \dots, x_n) = 1, iff R(x_1, \dots, x_n)$ . The transition relation of a FSM can be represented by  $f(v, x, v')$ , where  $v$  is the current state variables,  $x$  is the input variables,  $v'$  is the next state variables. Image and Pre-image computation can be computed efficiently using BDDs. [14] gave an *AndExists* algorithm to efficiently compute  $\exists V'.(p \vee q)$ . By reducing 3SAT problem to *AndExists*



BDD computation, it is shown *AndExists* is at least NP-hard.

## 0.2.2 Temporal Logic Model Checking

The model of computation in temporal logic model checking is called Kripke structure  $M = \langle S, S_0, AP, R, L, H \rangle$ , where

- $S$  is the set of states,  $S_0$  is the initial state,
- $AP$  is the set of atomic propositions,
- $R \in S \times S$  is the transition relation, and
- $L : S \rightarrow P(AP)$  gives the set of atomic propositions true in each state.
- $H = h_1, \dots, h_n$  is a set of fairness constraints, where  $h_i \subseteq S$ .

Models such as Moore machine[13], Mealy machine commonly used in hardware description can be easily translated into Kripke structure.

In temporal logic, there are two branching time operators: **A** which stands for all execution paths from a given state and **E** which stands for some execution path. There are mainly four linear time operators(**F**, **G**, **X** or **U**):

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

computation tree logic(CTL) and linear temporal logic(LTL) are two kinds of temporal logic widely used as specification languages in model checking. In CTL each of the usual linear time operators must be directly preceded by a path quantifier. Examples of CTL formulas are:

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

Since  $\mathbf{AG}$ ,  $\mathbf{AU}$  and  $\mathbf{AX}$  can be rewritten in terms of  $\mathbf{EG}$ ,  $\mathbf{EU}$  and  $\mathbf{EX}$ , it is sufficient to only consider  $\mathbf{EX}f$ ,  $\mathbf{E}(f \mathbf{U} g)$  and  $\mathbf{EG}f$  in CTL model checking. The procedure to return the BDD representation of the states where the formula is true is called *Check* [4]:

1.  $CheckEX(f(\bar{v})) = \exists \bar{v}' [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$ .
2.  $CheckEU(f(\bar{v}), g(\bar{v})) = \mathbf{lfp} Z(\bar{v}) [g(\bar{v}) \vee (f(\bar{v}) \wedge CheckEX(Z(\bar{v})))]$ .
3.  $CheckEG(f(\bar{v})) = \mathbf{gfp} Z(\bar{v}) [f(\bar{v}) \wedge CheckEX(Z(\bar{v}))]$ .

Fairness constraints are needed for reactive concurrent systems. A path is said to be fair with respect to a set of fairness constraints if each constraint holds infinitely often along the path. CTL model checking with fairness can be described as:

1.  $\mathbf{EG}_{fair} f = \mathbf{gfp} S \left[ f \wedge \bigwedge_{k=1}^n \mathbf{EX}(\mathbf{E}(f \mathbf{U} S \wedge h_k)) \right]$ .
2.  $\mathbf{EX}_{fair} f = \mathbf{EX}(f \wedge \mathbf{EG}_{fair} true)$
3.  $\mathbf{E}_{fair}(f \mathbf{U} g) = \mathbf{E}(f \mathbf{U} (g \wedge \mathbf{EG}_{fair} true))$

Linear temporal logic(LTL) consists of formulas that have the form  $\mathbf{A}f$  where there is no branching time operators in  $f$ . [7] gave a LTL modeling checking algorithm based on CTL model checker SMV [14]. LTL formula  $\mathbf{A}f$  is converted to  $\neg\mathbf{E}\neg f$ . A tableau  $T = (S_T, R_T, L_T)$  with  $AP_f$  as its set of atomic propositions is constructed which has all the possible paths that satisfy  $f$ . Each state in the tableau is a set of elementary formulas  $el(f)$  obtained from  $f$ . New state variables are introduced for each occurrence of a temporal operator in  $f$  to encode the transition relation in  $T$ . A function  $sat(g)$  is defined for each subformula  $g$  of  $f$  to return a set of states in  $T$  where  $g$  is true. The following theorem establishes the conversion of LTL model checking to CTL model checking:

Theorem [7]:  $M, \sigma' \models \mathbf{E}f$  if and only if there is a state  $\sigma$  in  $T$  such that  $(\sigma, \sigma') \in sat(f)$  and  $P, (\sigma, \sigma') \models \mathbf{E}\mathbf{G} true$  under fairness constraints  $\{sat(\neg(g\mathbf{U}h) \vee h) \mid g\mathbf{U}h \text{ occurs in } f\}$ .

### 0.2.3 Implementation Relation between Models

In Model checking, both specification and implementation can be state transition systems, which describe behaviors at different level of abstraction. Typically, specification machine can be used as part of implementation machine when checking more abstract specifications. Model checking in this framework consists of checking the *implementation* relation between models, denoted by  $P \preceq Q$ , where  $P$  is the implementation machine and  $Q$  is the specification machine. There are two definitions of implementation[11], the first one is *trace-based implementation* or refinement, where every computation of the implementation is correlated to some computation of the specification. The second one is *tree-based implementation* or simulation, where every computation tree embodied in the implementation is correlated to some computation tree embodied in the specification. Simulation is a sufficient condition for refinement.

As far as temporal logic is concerned, refinement preserves LTL; while simulation preserves ACTL\* [13]. Refinement-checking problem is PSPACE-hard in the size of the implementation de-

scription and in the state space of the specification[10]. By adding *witness module* to make all the local variables of  $Q$  appear in  $P$ , the refinement checking problem can be reduced to transition-invariant checking, which is linear on the state spaces of both  $P$  and  $Q$ . Although simulation can be reduced to the problem of finding a simulation relation, with complexity linear in the state spaces of both  $P$  and  $Q$ , the simulation with fairness defined in [13] can not be checked efficiently(it is complete for PSPACE).

### 0.3 PCI Local Bus

The PCI Local Bus [18, 19, 20] 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.

#### 0.3.1 PCI Signals and Transactions

PCI bus signals(Figure 0-2)can be divided into the following categories according to their functionality. *Address and Data* lines are multiplexed and can be either 32 bit or 64 bit wide, and they also have a parity line for error correction. *Command* lines carry four bit commands at the start of each transaction, identifying the transaction type. *Interface Control* lines are used for handshaking between devices, device signalling, exclusive access and transaction termination. *Arbitration* lines are traditional request- and-grant type point-to-point lines between each device and an arbiter. PCI bus also supports four *Interrupt* lines and *IEEE JTAG* lines. *Error indicator* lines and system wide lines clock and reset are also required.

The following pins are important signals in PCI [21]:

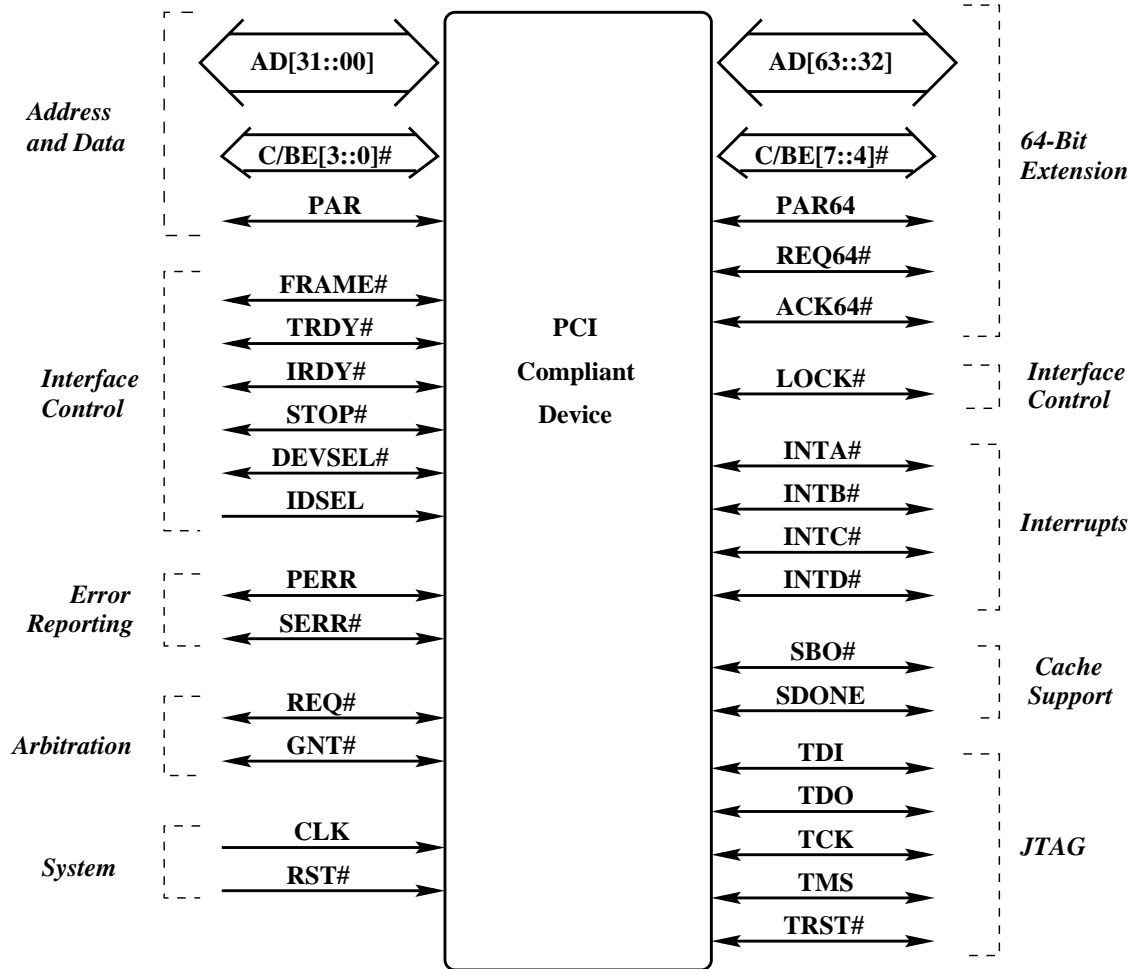


Figure 0-2: PCI signals

- **FRAME# (Frame):** This active-low signal identifies the beginning of a valid bus cycle. It identifies the address phase of the transfer and indicates that the AD and C/BE# signal lines contain valid address and command information. Once deasserted, the cycle is in the final data phase and completes when TRDY# and IRDY# are asserted in the same clock cycle to latch the final data transfer.
- **TRDY# (Target Rdy):** This active-low signal indicates that the accessed PCI resource (target) is ready to complete the bus cycle. This signal is used in conjunction with IRDY# such that when both are deasserted, the data phase is complete. For a read cycle, it indicates that valid

data is present on the bus. During a write cycle, it indicates that the target is ready to accept data.

- IRDY# (Initiator Rdy): This active-low signal indicates that, for write cycles, active data is valid on the bus driven by the bus master. During a read, it signifies that the master is ready to accept data.
- STOP# (Stop): This active-low signal indicates that the accessed PCI resource (target) wants to end the bus cycle without the bus master completing any or all accesses. The signal can be used if the target cannot accept or return more data or in the event of an error. Refer to the discussion on RETRY, DISCONNECT and TARGET ABORT.
- DEVSEL# (Device Select): This active-low signal indicates that a PCI resource (target) has decoded the address on the AD signal lines and claims the bus cycle. It is also used by other PCI resources to indicate another device has claimed ownership of the current transaction.
- AD[31::00] (Address/Data): These are 32-bit (AD31 is MSB) multiplexed lines used to convey address and data. They are valid on the rising edge of clock and are active-high lines. A typical bus cycle consists of one address phase and one or more data phases for burst reads and writes.
- C/BE[3::0]# (Command/Byte Enable): These active-low lines contain the command type for the bus cycle as well as the valid byte lanes of the AD signal lines. These lines are always valid.
- REQx# (Request): Each bus master has an individual and unique request line to the system arbiter. The master uses this line for output only, although the line is defined as bidirectional, to take advantage of a standard PCI buffer that has attributes of being in the 3-state mode. It

is an active-low line from master to arbiter and is used to ask for bus ownership.

- GNTx# (Grant): Each individual master has an individual and unique signal line allowing the system arbiter to communicate to a PCI resource that the master now owns the bus. Used only as an input by bus masters, Grant also is defined as bidirectional for the same reasons as for the Request signal.

A typical PCI bus transaction is demonstrated in Figure 0-3. 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.

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.

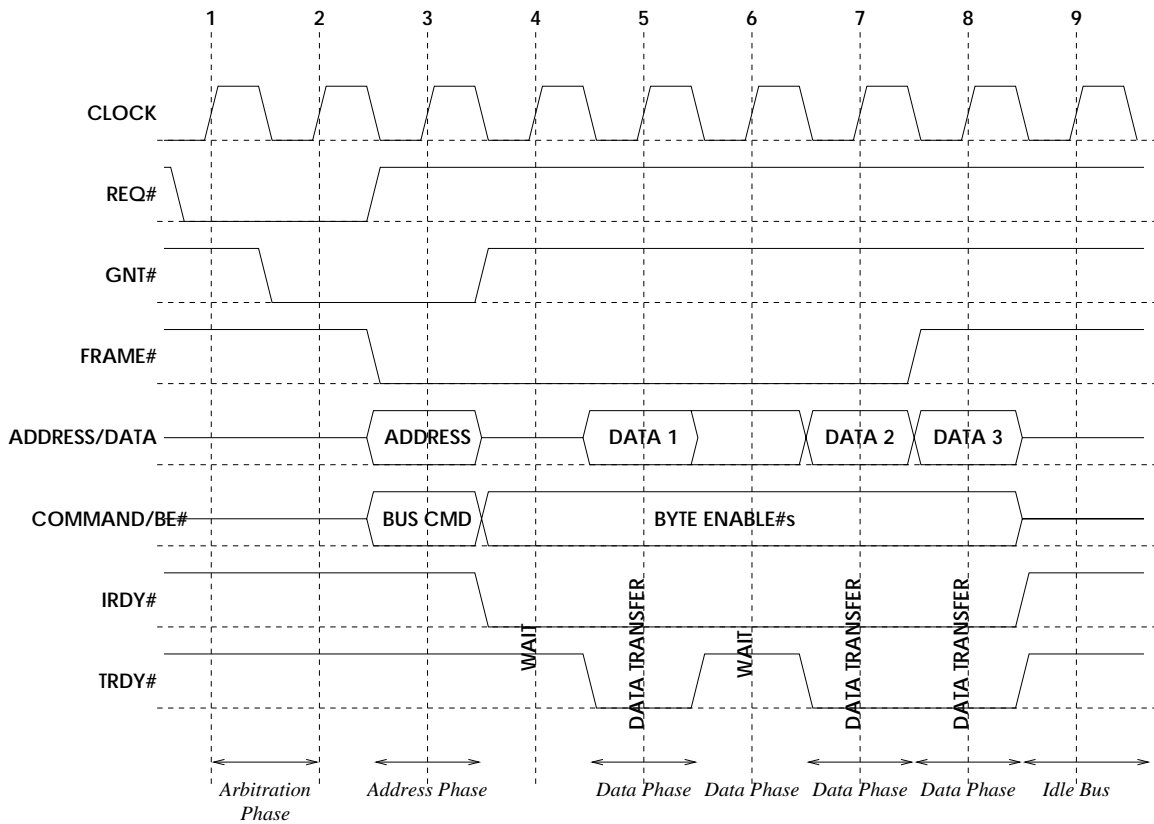


Figure 0-3: A typical PCI bus transaction

PCI local bus supports various kind of bus transactions, which are briefly described as follows. Apart from simple *memory and I/O read and writes*, we also have transactions for reading and writing configuration registers of devices on PCI bus. *Memory read multiple* command is intended to be used for bulk sequential data transfers where the memory system and the requesting master might gain some performance advantage by sequentially reading ahead one or more additional cache-line(s). *Memory read line* command is intended for reading complete cache line. *Memory write and invalidate* is used for master to write all bytes within the addressed cache-line in a single PCI transaction, and additional data transfer is also in terms of full cache-lines. PCI bus also supports fast back to back transaction (i.e. two transactions without idle states in between). All targets must implement fast back to back transactions, while masters may or may not implement them.

Delayed transactions and posting [18] are two important mechanism to improve system perfor-



mance in PCI. A delayed transaction progresses to completion in the three steps:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first step, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. The master of a Delayed Request is required to reissue the retried request until the request completes. During the second step, the target independently completes the request on the destination bus using the latched information from the Delayed Request. If the Delayed Request is a read, the target obtains the requested data and completion status. If the Delayed Request is a write, the target delivers the write data and obtains the completion status. The target stores the Delayed Completion until the master repeats the initial request. During the third step, the master successfully rearbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed. There are two types of devices that will use Delayed Transactions: I/O controllers and bridges.

Memory writes are allowed to be posted on the PCI bus. Posted transactions complete at the originating device before they reach their ultimate destination. The master will often proceed with other work, sometimes including other bus transactions, before the posted write reaches its ultimate destination. In essence, the intermediate agent of the access (e.g., a bridge) accepts the data on behalf of the actual target and assumes responsibility for ensuring that the access completes at the final destination.

### 0.3.2 PCI Bridge

A PCI-to-PCI bridge [19] provides a connection path between two independent PCI buses. The primary function of the bridge is to allow transactions to occur between a master on one PCI bus and a target on the other PCI bus. Thus a bridge has two PCI interfaces (Figure 0-4). The bridge functions as a target on the originating bus on behalf of the target that actually resides on the destination bus. Likewise, the bridge functions as a master on the destination bus on behalf of the originating master that actually resides on the originating bus.

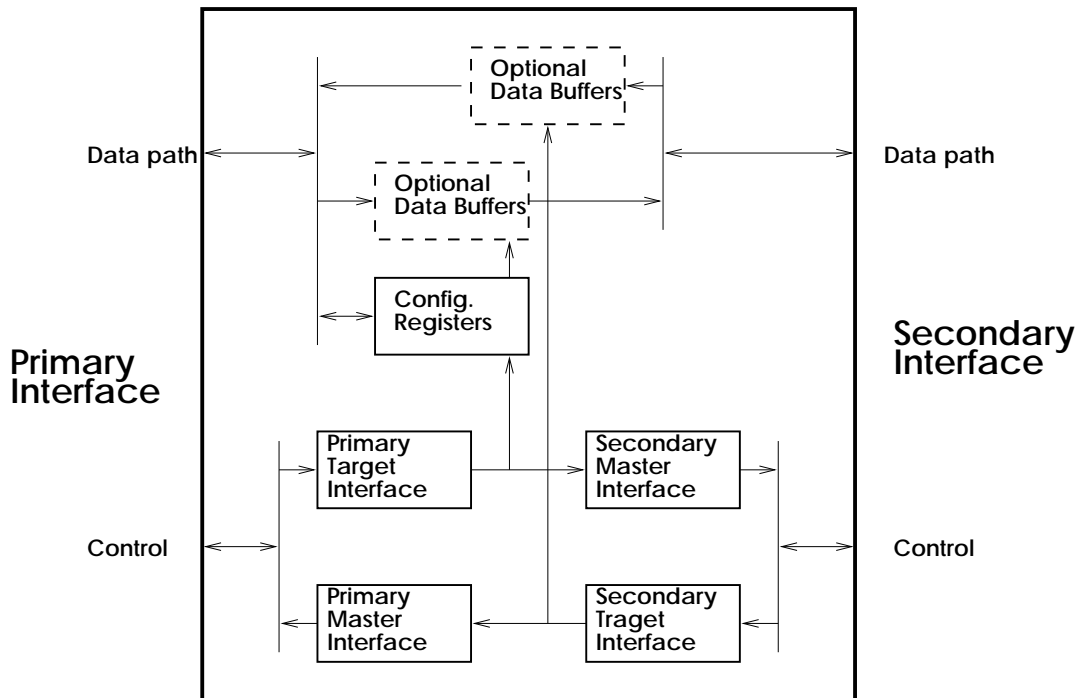


Figure 0-4: PCI Bridge Block Diagram

[19] requires a PCI bridge to support posting of memory write transactions and delayed transactions for non-posted transactions. *Posting and delayed* transaction can reduce the time for a master to hold the bus, but greatly increase the verification complexity. Transactions on PCI bus must satisfy the producer-consumer model. In this model, the producer produces the data and the

consumer consumes the data. The producer and consumer communicate between each other via a flag and a status element. The producer sets the flag when all the data has been written and then waits for a completion status code. 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, it clears it and the sequence repeats. If some of the producer's data writes were posted, then without buffer-flushing rules it might be possible for the consumer to see the flag set before the data writes had completed. The PCI ordering rules are written such that no matter which writes are posted, the consumer can never see the flag set and read the data until the data writes are finished. This model allows the data, the flag, the status element, the producer, and the consumer to reside anywhere in the system. Each of these can reside on different buses and the ordering rules maintain a consistent view of the data. Below are the requirements of PCI ordering rules:

1. Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus.
2. Write transactions crossing a bridge in opposite directions have no ordering relationship.
3. A read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.
4. A bridge can never make the acceptance(posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. Otherwise, a deadlock may occur.
5. A master must repeat any transaction terminated with Retry since the target may be using a

Delayed Transaction.

6. Once a Delayed Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, it is only a request and may be discarded at any time.
7. Delayed Requests and Delayed Completion are not required to be kept in their original order with respect to themselves or each other.
8. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.
9. A single master may have any number of outstanding requests terminated with Retry. However, if a master requires one transaction to be completed before the another, it cannot attempt the second one on PCI until the first one has completed.

PCI devices must follow certain allowable transaction orderings rules in order to satisfy the Producer-Consumer model. Table 0.1 summaries the PCI bridge transaction ordering rules.

Row pass Col.?	PMW (Col 2)	DRR (Col 3)	DWR (Col 4)	DRC (Col 5)	DWC (Col 6)
PMW (Row 1)	No	Yes	Yes	Yes	Yes
DRR (Row 2)	No	Yes/No	Yes/No	Yes/No	Yes/No
DWR (Row 3)	No	Yes/No	Yes/No	Yes/No	Yes/No
DRC (Row 4)	No	Yes	Yes	Yes/No	Yes/No
DWC (Row 4)	Yes/No	Yes	Yes	Yes/No	Yes/No

Table 0.1: Ordering rules for a bridge

**PMW** - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

**DRR** - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple commands.

**DWR** - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write command.

**DRC** - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The **DRC** contains the data requested by the master and the status of the target(normal, Master-Abort, Target-Abort, parity error, etc.).

**DWC** - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The **DWC** does not contain the data of the access, but only status of how it completed. The write data has been written to the specified target.

**No** - indicates that the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes found in column 2 prevent **PMW** data from being passed by other accesses and thereby maintain a consistent view of data in the system.

**Yes** - indicates that the subsequent transaction must be allowed to complete before the previous one or a deadlock occurs.

**Yes/No** - indicates that the bridge designer may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid.

Noted that there is no posted memory write completion, because the master completes its write

transaction before the data actually being written. Each entry in the table stands for whether the transaction in the row can bypass an earlier transaction in the column moving in the same direction.

Rule 1 - A subsequent **PMW** cannot pass a previously accepted **PMW**.(Col 2, Row 1)

Rule 2 - A read transaction must push posted write data to maintain ordering.(Col 2, Row 2)

Rule 3 - A non-postable write transaction must push posted write data to maintain ordering.(Col 2, Row 3)

Rule 4 - A read transaction must pull write data back to the originating bus of the read transaction.(Col 2, Row 4)

Rule 5 - A Posted Memory Write must be allowed to pass a Delayed Request(read or write) to avoid deadlocks.(Col 3 and Col 4, Row 1)

Rule 6 - A Delayed Completion(read or write) must be allowed to pass a Delayed Request(read or write) to avoid deadlocks.(Cols 3 and 4, Rows 4 and 5)

Rule 7 - A Posted Memory Write must be allowed to pass a Delayed Completion(read or write) to avoid deadlocks.(Col 5 and Col 6, Row 1).

## **0.4 Verifying PCI Bus**

The PCI bus protocol is verified in three steps. In the first step, we looked at properties related to the protocol without bridges. In the second step we verified properties for a single PCI bus with multiple masters and targets. In the last step, we modeled a PCI bridge and verified the producer-consumer model for bus transactions with bridges.

### **0.4.1 Single PCI Bus**

Figure 0-5 shows the configuration that we modeled to verify PCI bus properties. The arbiter is safe and fair. Masters and targets are modeled based on the Appendix of the *PCI Specification Revision 2.2* [18]. 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.

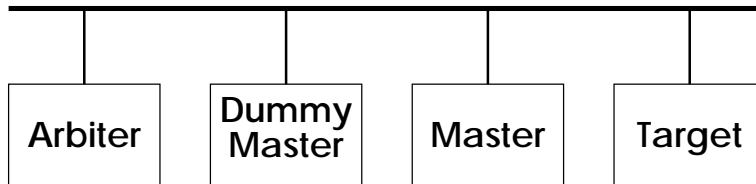
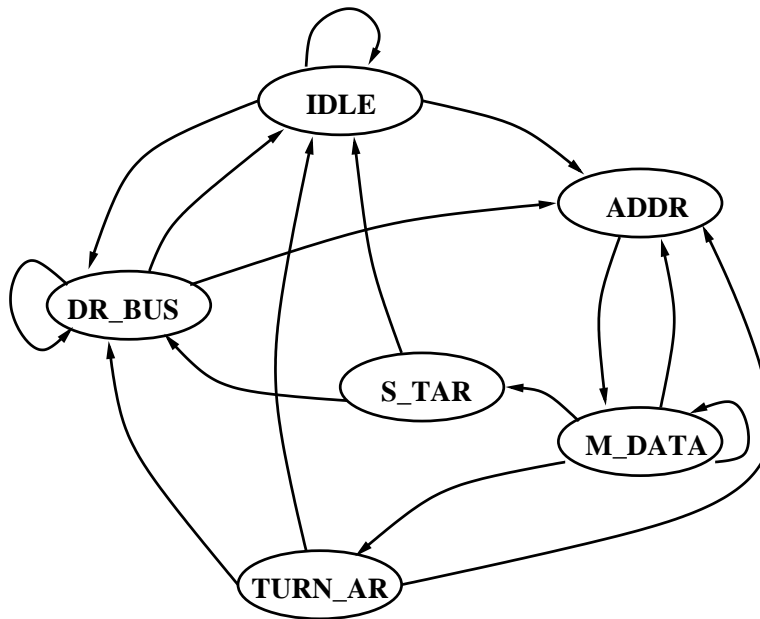


Figure 0-5: Configuration for verifying bus properties

The PCI master state machine (Figure 0-6) and PCI target state machine (Figure 0-7) are given in the PCI specification [18]. States in the master state machine are:

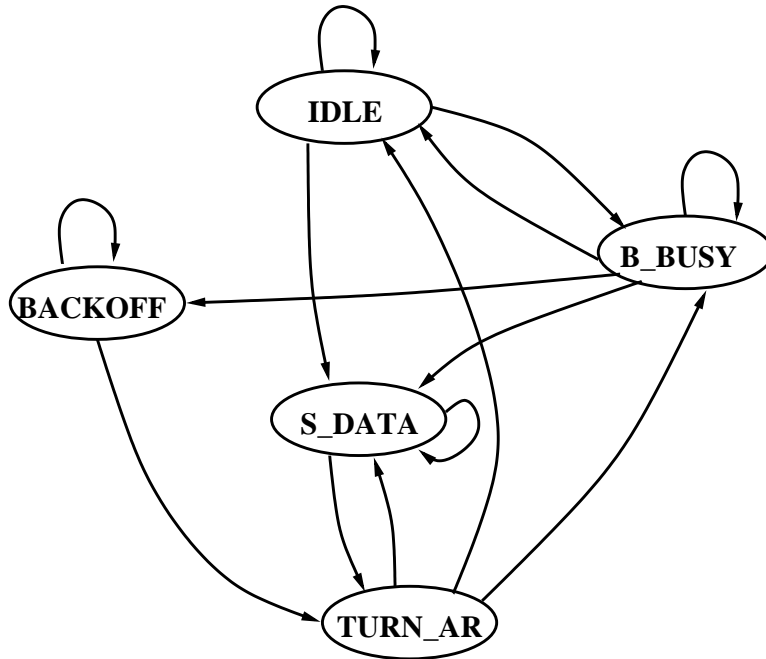


Master Sequencer Machine

Figure 0-6: PCI Master State Machine

- IDLE : waiting for transactions
- ADDR : address phase

- M\_DATA : data transaction phase
- TURN\_AR : turn around delay
- DR\_BUS : bus parking(master)
- S\_TAR : master termination



Target Sequencer Machine

Figure 0-7: PCI Target State Machine

States in the target state machine are:

- IDLE : waiting for transactions
- B\_BUSY : address decoding phase
- S\_DATA : data transaction phase
- TURN\_AR : turn around delay



- BACKOFF : target termination

We modeled the following PCI bus features:

- Bus Arbitration
- Burst Read and Write Transactions.
- Fast Back to Back Transaction.
- Master and Target Termination
  - Master Completion
  - Master Timeout
  - Target Retry
  - Target Disconnect
  - Target Abort
- Exclusive Access(LOCK)
- Parity Generation and Detection
- Latency
  - Target Decode Latency
  - Master Target Data Ready

Some abstraction is implemented in this model. We use 4-bits for the address line and 1-bit for data line. Initial data latency timer, subsequent data latency timer and master time out timer are all 8 cycles. Number of bus data phases is modeled by a counter, which controls the assertion of master termination. Target termination is modeled by the data latency timer and the counter for read/write

address, whenever the data latency timer exceeds 8 cycles or read/write address exceeds its address limit, target termination is asserted.

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:  $\mathbf{AG}(bus.FRAME\# \rightarrow \mathbf{AX}(\neg bus.FRAME\# \rightarrow bus.IRDY\#))$ . It says that when FRAME# is first asserted (active low), IRDY# should stay deasserted. SMV determined that this property is true. Following are some interesting properties we verified.

- Bus Arbitration is safe and fair.
- Bus Driving
  - AD can't be driven in the cycle following a read command.
  - Once asserted, STOP# can't be deasserted until FRAME# is deasserted.
  - Master must driven FRAME# until transaction finished.
  - FRAME# can only be deasserted with the assertion of IRDY#.
  - FRAME# will eventually be deasserted.
  - Once STOP# is asserted, FRAME# should be deasserted as soon as IRDY# is asserted
  - Once a target has asserted TRDY# or STOP#, it can't change DEVSEL#, TRDY#, or STOP# until the current data phase completes.
- Data Transferred
  - It is possible that Transaction succeeds.
  - When both master and target are ready, their read/write addresses are the same.
  - A successful read transaction transfer data correctly from target to master.

## 0.4.2 PCI Bus with Multiple Masters and Targets

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.

### Assume-Guarantee Reasoning

[13] defined a preorder between models which preserves ACTL and developed a framework to do compositional reasoning. The capability of the proposed framework is demonstrated by the following typical proof:

$$\frac{\begin{array}{l} P, \phi \models \psi \\ Q \models \phi \end{array}}{P \parallel Q \models \psi}$$

One major limitation in [13] is that usually the environment assumptions needed to verify interacting processes are interdependent. It can't handle the following proof rule:

$$\frac{\begin{array}{l} P \parallel Q' \models P' \\ P' \parallel Q \models Q' \end{array}}{P \parallel Q \models P' \parallel Q'}$$

[15] established a framework capable of doing circular composition reasoning, where circularity is resolved by induction over time. Define  $<_M$ , the dependency relation of machine  $M$ , be the set of pairs  $(\gamma, \sigma)$  such that  $M_\sigma$  is a gate (has zero delay) and  $\gamma$  is an input of  $M_\sigma$ .

Theorem[15]: Let  $P$  and  $Q$  be machines. For all  $\sigma \in S$ , let  $\varepsilon^\sigma$  be a machine such that:

- for all  $\sigma' \in \text{signals}$ :  $\varepsilon_{\sigma'}^\sigma = P_{\sigma'}$  or  $\varepsilon_{\sigma'}^\sigma = Q_{\sigma'}$ , and
- $\varepsilon_\sigma^\sigma = Q_\sigma$ .

Let  $<^*$  be the relation  $(<_P \cup <_Q)^*$ . If  $<^*$  is irreflexive then the following inference rule is sound:

$$\frac{\text{for all } \sigma : \varepsilon^\sigma \models P_\sigma}{Q \models P}$$

### Temporal Case Splitting

Although there may be large data structures in the model, but for a particular transaction, only part of the components are affected. By considering individually every path that a data item might take, we cut down the state space considerably. for example:

*layer spec:*

*if*(out.valid) out.data := bytes[out.idx]

*forall* (i in INDEX)

*subcase* spec\_case[i] of out.data//spec for out.idx = i;

In this example, specification *spec* requires that when out.valid is true, *our.data* should equals to the value *bytes[out.idx]*, where *bytes* is an array, *out.idx* is an index to that array. The *spec\_case[i]* represents the *spec* when *out.idx = i*. Although the array *bytes* can be very big, but when verifying *spec\_case[i]* only one entry of it is needed, while all the other entries are unspecified. The correctness of case splitting is captured by the following theorem:

Theorem[16]: If for all *i* in the range of variable *v*,  $\models \mathbf{G}(v = i \rightarrow \psi)$ , then  $\models \mathbf{G}\psi$ .

### Symmetry Reduction

Symmetry is captured by scalarsets in SMV and the only operation within a scalarset is comparison for equality and index to an array.

- *scalarset TAG 0..15*

- *typedef word array TAG of boolean*

The overall program semantics is not changed by exchanging any pair of elements of a scalarset type. Given a parameterized class of assertions to prove, SMV chooses a representative set of instances of the class, such that any instance can be reduced to one in the set by permuting scalarset values.

### **Verification Using CBL-SMV**

We developed a single bus model in CBL SMV [15] with 5 masters and 5 targets. In this model, there is symmetry within the masters, as well as the targets (An abstraction is done to abstract the address range of each target to be the index of the target in order to preserve symmetry). 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. The problem of this approach using symmetry and case analysis is the mismatch of the master and target, some of the bad behaviors when the inactive masters and targets are unconstrained are:

- A target can delay its active until the next transaction happens, so that two targets become active at the same time.
- An active target can remain active when a new transaction begins.
- The active master can assert and deassert its FRAME# signal multiple times during one transaction, so that other targets will get address hit.

In order to prove bus driving properties, we have to prove the following non-interference lemmas using the cyclic assume-guarantee rule:

- Inactive masters and targets can not drive the bus.

- Only one master and one target can be active at a time.
- Local properties about FRAME#, DEVSEL#, STOP#.
- 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 exerted from PCI specification [18]:

- 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.

### 0.4.3 Verifying PCI Bridge

PCI bridge transaction ordering rules are modeled based on the configuration given in Figure 0-8. We modeled the PCI bridge according to the PCI bridge specs [19]. The state machine of PCI bridge essentially is a composition of four state machines, the primary master and target and the

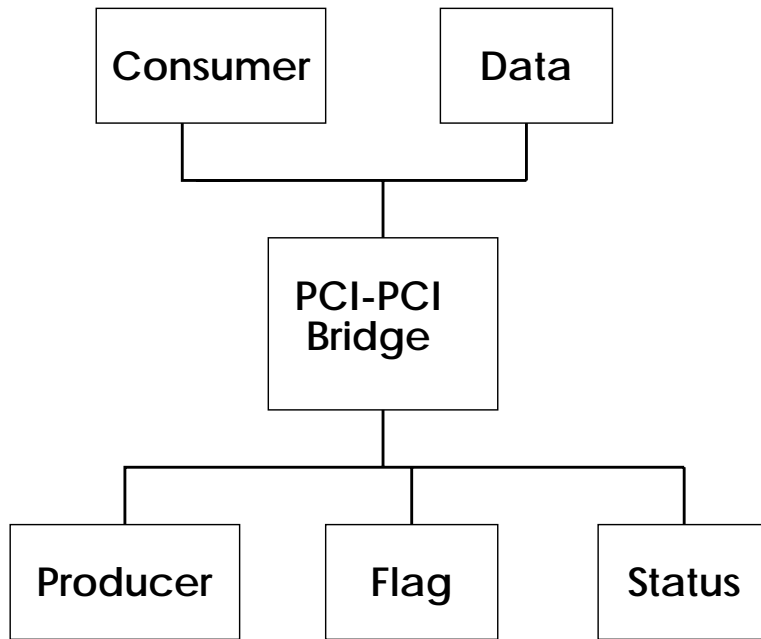


Figure 0-8: Producer-Consumer model

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, fast back to back transaction etc. We need to represent only three distinct values of data 0,1 and 2 for verifying this model [14], hence the data bus should be at least 2 bits wide. We abstracted the address range of each target to only one address, we changed the address decoding logic in each target and on both sides of the bridge. Since there are only 3 targets in our configuration, the address bus is 2 bits wide. Bridges also have finite data buffers for posting and delayed data.

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  $\text{flag} = 1$ , it should then receive  $x$ . The following three properties capture the correctness of this behavior ( $c$  denotes consumer).

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

## 0.5 Experimental Results

We developed the single PCI bus model with one master, one target, and one dummy master and a bus arbiter in CMU SMV. 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 Bus Specification [18] 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. The following equation from the target state machine in PCI specification [18] is the condition for the target to make a transition from B\_BUSY to IDLE(Figure 0-7):

$$FRAME\# * D\_done + FRAME\#\neg D\_done * \neg DEVSEL\#$$

According to this condition, the target makes a transition from B\_BUSY to IDLE *when* FRAME# is deasserted and D\_done (the signal driven by the decoding logic in each target to represent the



address decoding is finished) 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 0-9). This error is caught by the following CTL formula:  $\mathbf{AG}((m.req \wedge m.data\_cnt = 1) \rightarrow \mathbf{A}(m.req \wedge \neg t.ack \mathbf{U} m.timeout))$  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*

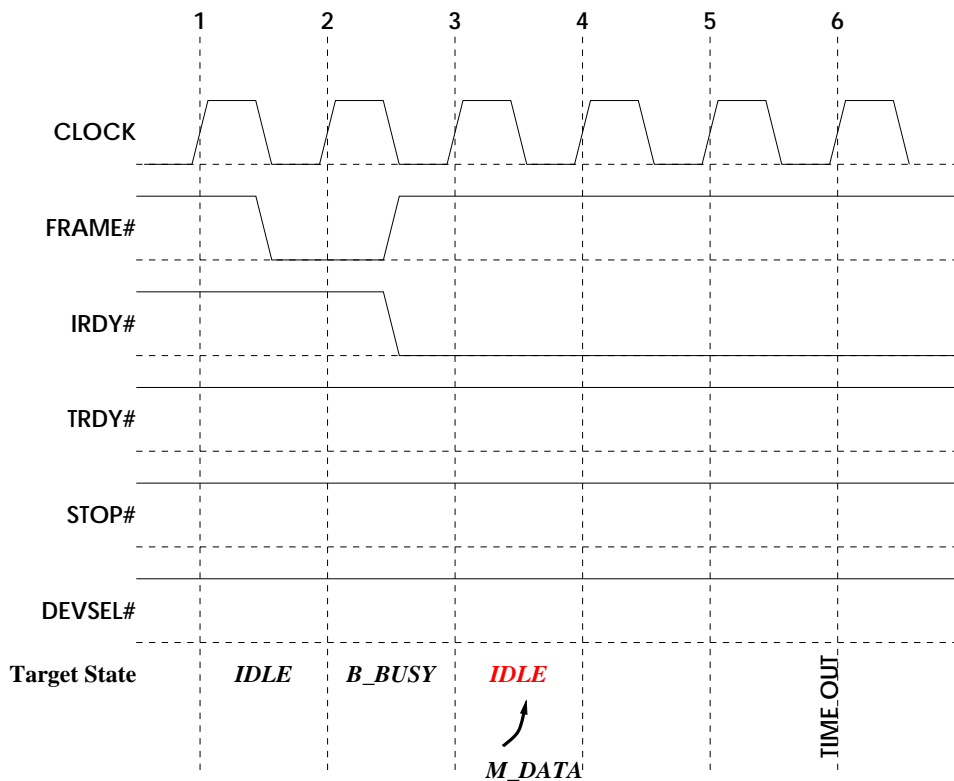


Figure 0-9: Illustration of the first bug

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.

$$FRAME\# = \neg(ADDR + M\_DATA * !Dev\_to * \{[!Comp * (!To + !GNT\#) * STOP\#] + !Ready\}) [18].$$

Let us assume that IRDY#, FRAME# are asserted, and GNT#, TRDY# are deasserted 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.

For the PCI bus model with multiple masters and targets, on an UltraSparc 248MHz machine, it took 46 minutes and 13M BDD nodes to verify all the lemmas and properties. As a comparison, after taking about 2Gb memory and 5 hours, the first property could not be verified without techniques mentioned in Section 0.4.

We modeled a multiple bus system with a bridge (see Figure 0-8) and verified the correctness of the producer-consumer model by checking the three major properties described in Section 0.4. In Table 0.2, we show our various statistics for two implementations of PCI bridges that have different numbers of data buffers.

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

Table 0.2: Experimental results for verifying PCI bridge

Although we haven't found any incident of the violation of the producer-consumer model because of the transaction ordering rules in this simple configuration. We indeed encountered some design errors by ourselves:

1. Since the **DRC**(Section 0.3) and **DWC** are moving in the opposite direction compared with the original **DRR** and **DWR**, so that they participate in the transaction ordering rule in the other direction through the bridge. An early design decision was to keep the **DRR** and **DWR** in the buffer until they become completed, then request to the other direction in the same bridge so that the completed delayed request can be placed in the buffers of the other direction, then when the original request is arrived again, it can be matched by the completed request in

the buffers of the other direction. The problem of this design is that it could be the case where there is no entry available in the other buffer unless the completed request finishes. Thus a deadlock happened. We fixed this problem by allowing the **DRC** and **DWC** to quit without having to be moved to the buffers in the other direction after checking the observance of the transaction ordering rules.

2. Another problem is due to the use of bursting for the consumer to set the completion status. The scenario is: consumer has read the newly written data by the producer, then it uses *burst*, *post* transaction to set the completion status. While consumer are posting to the bridge, bridge starts to write the latched data to status, so that the completion status is being set. Before the consumer finishes the posting, burst writing to the status by the bridge is being disconnected, then bridge reports to the consumer that posting to the bridge should be disconnected as well. Thus although the status is already being set, but the consumer consider it haven't set the status because of disconnection. Later, the producer checks the status and finds out that the consumer has set it to one. So it clears the status, writes the new data, sets the flag and waits for the status being set again. The consumer will resume its disconnected posting to the status, and successfully set the status. Then the producer sees the status is being set, then it will generate a new data, although the last data haven't been read by consumer. In reality, using burst and repeating from the original address of request after being disconnected is rare, thus the scenario may not happen. But it reveals some tricky aspects of the bridge transactions.

We expanded our bridge model by adding another bridge and bus. The model becomes very big. After more than 40 hours and 800Mb, none of the three major properties has been proven using a new version of SMV [22]. More techniques to avoid state explosion problem are needed to handle this model in the future.

## 0.6 Conclusions and Future Work

We have proposed a new methodology for verifying system-on-chip designs, which takes advantage of the modularity of the design and correlates the design process by using refinement checking. As the first example, we have verified the PCI Local Bus protocol using the symbolic model checker SMV. Two potential bugs in the standard PCI bus specification are presented. 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.

# Bibliography

- [1] R. E. Bryant. "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, C-35(8), 1986.
- [2] K. Brace, R. Rudell, R. E. Bryant. "Efficient Implementation of a BDD Package", *27th ACM/IEEE Design Automation Conference*, 1990.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] E. M. Clarke, O. Grumberg, H. Hiraishi. "Another Look at LTL Model Checking", *Formal Methods in System Design*, 10,47-71(1997).
- [8] 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.
- [9] Robert Allen, David Garlan. "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology* July 1997.
- [10] T. Henzinger, S. Qadeer, S. Rajamani. "You Assume, We Guarantee: Methodology and Case Studies", *CAV 1998, Lecture Notes in Computer Science 1427*, Springer-Verlag, 1998.

- [11] Orna Kupferman, Moshe Y. Vardi. "Verification of Fair Transition Systems", *CAV 1996*.
- [12] R. Kurshan. *Computer Aided Verification of Coordinating Processes*, Princeton, 1994.
- [13] D. E. Long. "Model Checking, Abstraction and Compositional Verification", *Carnegie Mellon University publication CMU-CS-93-178*, July 1993.
- [14] K. L. McMillan. "Symbolic Model Checking: An Approach to the State Explosion Problem", *Carnegie Mellon University publication CMU-CS-92-131*, May 1992.
- [15] K.L. McMillan. "A compositional rule for hardware design refinement", *Computer Aided Verification*, pp 24-35, June 1997.
- [16] K.L. McMillan. "Verification of Infinite State Systems by Compositional Model Checking", *Cadence Berkeley Labs Technical Report*, 1999.
- [17] A. Mokedem, R. Hosabetu, and G. Gopalakrishnan. "Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem", *FMCAD*, Nov. 98.
- [18] PCI Special Interest Group. *PCI Local Bus Specification Rev 2.2*, Dec. 1998.
- [19] PCI Special Interest Group. *PCI to PCI Bridge Architecture Specification Rev 1.1*, Dec. 1998.
- [20] E. Solari, G. Willse. *PCI Hardware and Software - Architecture and Design*, Annabooks, 1998.
- [21] <http://www.ti.com/sc/docs/asic/srga013/s1.htm>, Texas Instruments Incorporated.
- [22] B. Yang, R. Simmons, R. Bryant, and D. O'Hallaron. "Optimizing Symbolic Model Checking for Constraint-Rich Models", *CAV 1999* 1999.