International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Assertion-Based Formal Verification of CPU-Cache Crossbar of OpenSPARC T1 Processor

P. Vishnu Vardhan Reddy[1], Rajendra M. Patrikar[2]

[1]Gayatri Vidya Parishad College of Engineering,
Visakhapatnam 530048, India
vishnunbkr@gmail.com

[2]Visvesvaraya National Institute of Technology, Nagpur,
Nagpur 440010, India
rmpatrikar@ece.vnit.ac.in

Abstract: ***Functional verification of complex designs, such as multi-core processors, is a challenging task in the entire verification cycle, because bugs which are not uncovered during this phase will carry on to the later design stages. The cost of fixing bugs is very high at later stages as compared to fixing them at the RTL implementation phase. Conventional verification methods like coverage-driven simulation techniques may not be able to uncover all the bugs due to their inability to exercise corner-case scenarios in a design. Formal methods like theorem proving, assertion-based verification are exhaustive and detect all corner-case bugs. This paper proposes an assertion-based formal approach for the verification of the CPU-Cache Crossbar module of the SPARC T1 processor, whose behavior is characterized by complex request patterns originating from the multiple cores to access shared resources such as the Level 2 cache memory banks, floating-point unit, and I/O Bridge – ideal candidates for an assertion based formal verification approach.***

Keywords: Arbiter, assertion-based verification, formal verification, multi-core processor.

1. Introduction

Functional verification is the task of verifying whether specifications are implemented correctly or not. As the design complexities are increasing, the task of verification poses new challenges to the verification engineers. Multi-core microprocessor designs having large blocks of parallel processing logic that share common resources pose unique challenges for functional verification. Besides parallel processing logic, these designs have more number of arbiters that ensures packet transfers among different sources and destinations. Validating multiple levels of arbitration is a difficult task. Multi-core processors have design blocks of replicated logic which reduces design effort, but increases verification complexity due to inherent asymmetry between threads [1].

Verification of this kind of complex designs takes majority of the resources (60-70%) — engineers, time, and money [2]. Even with such a significant effort, functional bugs are the main causes of silicon re-spin [3]. Once the specifications are implemented at Register Transfer Level (RTL) functional verification should guarantee that the design has no bugs.

If there are any bugs present in the RTL code the entire verification process needs to iterated, after isolating the bug and removing it from the design. After RTL implementation, design will undergo logic synthesis and backend stages involving physical layout synthesis where fixing any design functional bugs can be very expensive and time consuming. Hence, detection of all design implementation bugs and corner-case bugs exhaustively during functional verification is imperative. One interesting difference between simulation-based and Formal-based methods is, the former potentially demonstrates the presence of a bug whereas the later ensures the absence of a bug also [4].

The CPU-Cache Crossbar (CCX) module of the SPARC T1 processor which has more number of arbiters, manages complex request patterns from all cores to shared resources such as Level 2 cache (L2 cache) memory banks, floating-point unit (FPU), and I/O Bridge (IOB) and vice versa. Checking for fair arbitration for complex request patterns among multiple arbiters is a daunting task under all possible scenarios by simulation-based methods. Thus, assertion-based formal method is proposed for verification of CPU-Cache Crossbar (CCX).

This paper is organized as follows. Section 2 presents a brief introduction about an assertion-based verification. Section 3 presents SPARC T1 processor details and its various interfaces along with brief description of CCX. Section 4 presents Processor Cache Crossbar (PCX) arbiter. Section 5 presents a discussion on proposed verification methodology on CCX module, while section 6 concludes the discussion on proposed approach.

2. Assertion-Based Verification

In Assertion-based Verification (ABV) the design intent is captured by properties specified in one of the standard assertion languages like Property Specification Language (PSL) [5], SystemVerilog Assertions (SVA) [6]. Property specification (i.e., assertions, constraints, and functional coverage) is fundamental to assertion-based verification. Informally a property specification can be viewed as a composition of three distinct layers [7].

- The *Boolean layer*, which is comprised of Boolean expressions (e.g., Verilog or VHDL).
- The *temporal layer*, which describes the relationship of Boolean expressions over time.

- The *verification layer*, which describes how to use a property during verification.

Assertions developed from RTL specifications can be used either in simulation or formal verification. The simulation-based approach is called Dynamic ABV, since the properties are checked over simulation run — it captures only those behaviors that are encountered by simulation. In contrast assertion-based formal verification performs exhaustive checking of the design, i.e. for all possible behaviors under all possible input combinations [8].

3. SPARC T1 Processor

The OpenSPARC T1 processor consists of eight SPARC® processor cores and each core has full hardware support for four threads. These eight cores are connected to an on-chip L2 cache banks through a crossbar as shown in Figure 1 [9]. The four on-chip Dynamic random-access memory (DRAM) controllers directly interface to the Double data rate synchronous DRAM (DDR2 SDRAM). Further J-Bus controller interfaces between I/O subsystem and processor. All the eight cores, four L2 cache banks, IOB, and FPU are interfaced through CCX. CCX manages packet transfers among all these and its features are

- Each source can queues up to two packets per destination.
- Three stage pipeline— request, arbitrate, and transmit.
- Oldest request getting high priority.

CCX consists of two main blocks — Processor-Cache Crossbar (PCX) and Cache-Processor Crossbar (CPX) as shown in Figure 2 [9].

3.1 Processor-Cache Crossbar

PCX accepts packets from any of the eight cores and delivers to any one of four L2 cache banks, IOB, or FPU. As L2 cache banks and IOB can process only limited number of packets, destination sends a stall signal to PCX after its maximum limit, but FPU cannot stall PCX. PCX contains five arbiters (Figure 2) corresponding to six destinations.

3.2 Cache-Processor Crossbar

CPX accepts packets from any of the four L2 cache banks, IOB, and FPU and delivers to any of the eight cores. Since each core has an efficient mechanism to drain the buffer that stores packets, CPX does not receive any stall signal. CPX contains eight arbiters (Figure 2) corresponding to eight cores.

4. PCX Arbiter

4.1 PCX Arbiter Control Flow Logic

The PCX arbiter has eight FIFO queues for control flow logic which are sixteen entries deep as shown in Figure 3 [9]. When the arbiter corresponds to the particular destination receives a packet from one only one source at particular clock cycle then it is processed in same cycle before it receives a packet in next cycle when there is no stall from destination. When multiple sources send a packet to one destination in same cycle, the arbiter will decide the priority depends on direction bit.

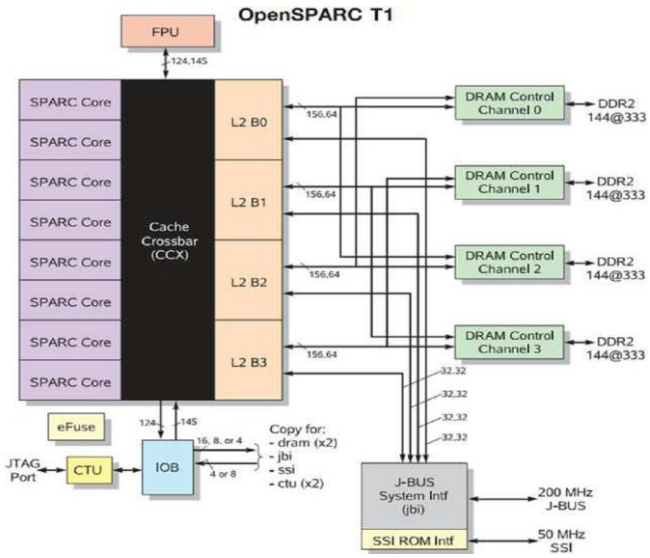

Figure 1: OpenSPARC T1 processor block diagram.

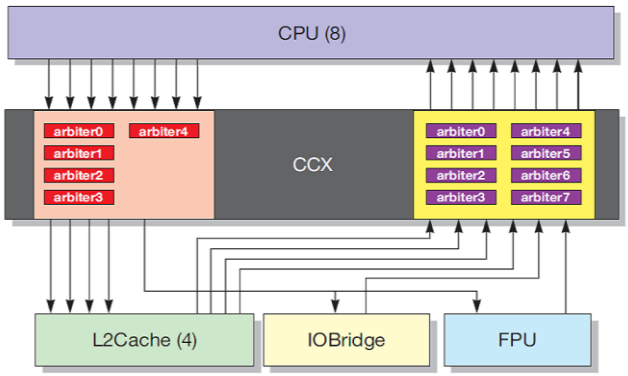

Figure 2: PCX and CPX internals block diagram.

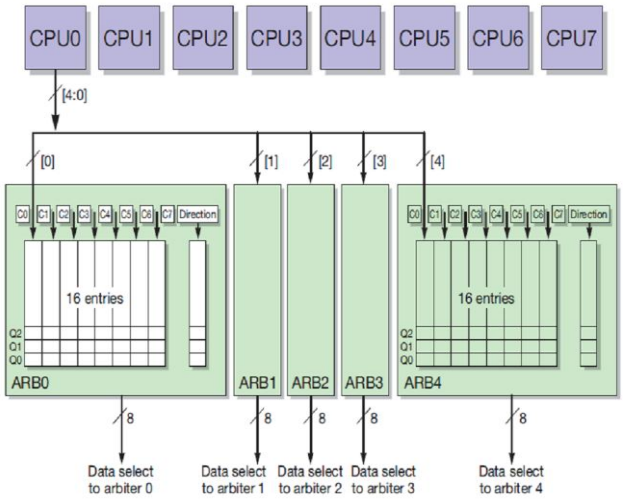

Figure 3: PCX Arbiter control flow block diagram.

If the direction is high the priority is from CPU0 to CPU7 otherwise it is from CPU7 to CPU0. Arbiter will generate 8-bit signal which is one hot to data FIFO multiplexers (Figure 4) and it also sends acknowledgement to source and data ready to the destination.

4.2 PCX Arbiter Data Flow Logic

Similar to the control flow it has eight FIFO queues for data flow logic which are two entries deep as shown in Figure 4 [9]. The PCX receives data packets of 124-bits wide and delivers to the destination without any modifications.

The CPX arbiters are similar to PCX arbiters except that packets to CPX are 145-bits wide and it does not receive any stall signal

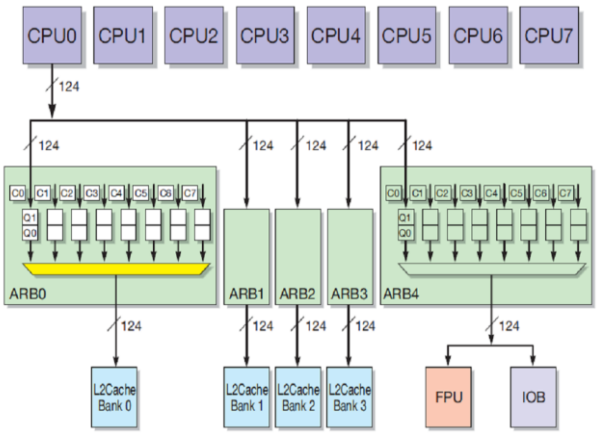

Figure 4: PCX Arbiter data flow block diagram.

5. Verification Methodology and Discussions

The CPU-Cache Crossbar (CCX) is verified using Incisive® Formal Verifier (IFV) tool from Cadence [10]. SPARC T1 processor Verilog RTL code is available as an open-source [9]. CCX RTL consists of approximately 31,290 D flip-flop/latches. CCX design hierarchy summary by IFV tool is shown in Table 1. Cadence IFV supports most of the industry standard assertion languages for specifying properties. In this paper PSL [5] is used for properties specification. Specification of the environment in which the design is embedded called constraints, is the key to property verification. In IFV we can specify these constraints as PSL properties.

Table 1: CCX Design Hierarchy Summary

Parameter	*Instances*	*Unique*
Modules	3277	97
Registers	2939	34
Scalar wires	40642	-
Expanded wires	109175	2591
Always blocks	2763	10
Continuous assignments	7640	788
Pseudo assignments	4067	266

In the following sub-sections we describe a few properties that are specified to verify requests originating from the eight processor cores (as Masters) for access to one of the four L2 cache banks, the FPU port or the IOB port (as Slaves), mediated through the PCX block. Similar kinds of properties are specified for transactions originating from the slaves to the masters mediated through the CPX block.

The following are the few kinds of properties that are verified on CCX using IFV. The status or result of property specifications (i.e., pass, fail, or explored) are discussed along with property description.

5.1 PCX Verification

5.1.1 One CPU to anyone of the four L2 cache banks

a) Without stall from any destination

For example, CPU0 sends a request to write a data packet to L2 Cache bank0 through ARB0, ARB0 ensures that CPU0 should be acknowledged with a grant, and a data ready, packet transfer to L2 cache bank0 when there is no stall signal from L2 cache bank0. These kinds of properties are proved by putting constraints on other CPU requests and stall signals.

b) With stall from destination:

For example, CPU7 sends a request to write a data packet to L2 Cache bank3 through ARB3, ARB3 ensures that CPU7 should not be acknowledged with grant.

5.1.2 Two or More CPUs to anyone of the four L2 cache banks

For example, CPU0 and CPU1 concurrently send a request to write a data packet to L2 cache bank0 through ARB0, ARB0 arbitrates multiple requests based on status of direction signal. CPU0 should get grant first if direction is high otherwise CPU1 is given the grant. Since the direction signal toggles in every clock cycle, setting up an environmental constraint to check for the correctness of the functional behavior for a given set of concurrent requests, for a given direction as set by the value of the direction bit can be extremely difficult. This leads IFV to fail any property very easily by throwing up counter examples when the verification has to be carried out exhaustively. For exhaustiveness, all possible complex request patterns need to be specified in the antecedent part of a PSL property, while the behavior consistent with each different request patterns needs to be captured over multiple clock cycles in the consequent part of the property for the behavior to be validated.

One way to overcome this is by capturing the behavior of the PCX arbitration logic in the PSL modeling layer and checking, behavior of the RTL implementation of the PCX arbitration logic against it over all clocks cycles against all possible concurrent requests originating from multiple CPUs. This is tantamount to replicating the PCX arbitration logic and incurs the cost of additional flip-flops or register elements needed in the modeling layer version of the PCX arbitration logic. A much simpler approach is to write a cover for the expected behavior for a pre-defined input request pattern. However, this approach does not guarantee the correctness over all possible input behaviors.

A manual analysis of the counter example generated by IFV can also reveal interesting details of the possible correctness of behavior for a particular input request pattern. A counter example could point to a genuine bug in the design or an

error in the specification and/or an error in setting up environmental constraints.

If the property specification is correct then we can directly go to original source of bug and fix it. The counter example based analysis even if done manually, can be much faster in pin-pointing the source of bug in either the design, or in the specification, as compared to inferring the same from an analysis of the simulation based traces.

5.1.3 Two or more CPUs to different destinations at same time

CPU7 and CPU1 send a request, data packet to L2 cache0 and FPU through ARB0 and ARB4 respectively, then ARB0 ensures that CPU7 should be acknowledged with grant, and data ready, packet transfer to L2 cache0 when there is no stall signal from L2 cache0. Whereas ARB4 ensures that CPU1 should be acknowledged with grant, and data ready, packet transfer to FPU.

5.1.4 Checking of mutually exclusive grants

If two or more CPUs send request to same destination (e.g., L2 cache3) at same time, then arbiter (ARB3) should not generate two grants in the one cycle. These kinds of properties are specified by using never in temporal layer of PSL structure.

5.1.5 Checking for correct grant sequence

When all eight CPUs send request at same time, the correct grant sequence is verified by visual inspection of two counter-examples. For the property specified CPU0 to CPU7 as grant sequence, the counter example showed CPU7 to CPU0 as grant sequence as one counter example. The second counter example showed vice versa.

5.2 CPX Verification

5.2.1 Any L2 cache bank to any CPU

For example, L2 cache bank0 send request and data packet to CPU3 through ARB3, ARB3 ensures that L2 cache bank0 should be acknowledged with the grant, and a data ready, packet transfer to CPU3.

5.2.2 FPU/IOB to any CPU

For example, FPU/IOB send request and data packet to CPU1 through ARB1, ARB1 ensures that data ready, packet transfer to CPU3 and grant to IOB, however FPU does not receive grant.

The other properties like checking for mutually exclusive grants and others are verified in similar to PCX. Besides all these properties we have verified few corner cases like arbiter generating grants when no requests at all and when stall signal is constrained arbiter generating grants in future. Table2 gives a snapshot of formal verification results by IFV for PCX and CPX blocks. The explored properties are due to limitations of tool, which is usually because of deep FIFOs.

Table 2: Assertion Summary

Block	*Properties*	*Proved*	*Explored*
PCX	624	484	140
CPX	674	554	120

6. Conclusion

CCX module of SPARC T1 processor is verified exhaustively by using an assertion-based formal approach. The advantage of proposed method is verification engineer starts developing properties parallel to the RTL design engineer, thereby reducing verification time, and hence overall time to market. But, these formal-based methods are not mature to handle end to end formal verification because of state explosion problem. Some solutions to state explosion problems are design abstractions and assume-guarantee verification. In practice formal methods does exhaustive checking for specified properties over all possible behaviors of implementation, but does not guarantee that the specified properties are sufficient for full design intent coverage. However, these formal methods are ideal for control logic dominant designs.

References

[1] B. Turumella, and M. Sharma, “Assertion-based verification of a 32 thread SPARC™ CMT microprocessor,” In Proc. 45th ACM/IEEE, DAC Jun. 2008, pp. 256-261.
[2] A. Lungu, and Daniel J. Sorin, “Verification-Aware Microprocessor Design,” In Proc. 16th International Conference on Parallel Architecture and Compilation Techniques, IEEE Computer Society, Sept. 2007, pp. 83-93.
[3] Alok sanghavi, “What is formal verification?” [Online]. Available:http://www.eetasia.com/STATIC/PDF/201005/EEOL_2010MAY21_EDA_TA_01.pdf
[4] Douglas L. Perry, and H. Foster, “Introduction to formal techniques” in Applied Formal Verification, McGraw-Hill, Inc., 2005, pp. 39-65.
[5] IEEE Standard for Property Specification Language (PSL), IEEE Std 1850-2010 (Revision of IEEE Std1850-2005) – Redline, pp. 1-188, Apr. 2010.
[6] IEEE Standard for SystemVerilog, “Unified Hardware Design, Specification, and Verification Language”, IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) – Redline, pp. 1-1346, Dec. 2009.
[7] R. Drechsler, “Assertion-Based Verification” in Advanced Formal Verification, Kluwer Academic Publishers, 2004, pp. 167-202.
[8] P. Dasgupta, “A roadmap for formal property verification”, Springer Netherlands, 2006.
[9] SPARC T1 Microarchitecture and RTL code [online]. Available:http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html
[10] Cadence Incisive Formal verifier [online]. Available: http://www.cadence.com/products/ld/formal_verifier/pages/default.aspx