ABSTRACT

ZHANG, XU. Verification Strategy of Cache Coherence for OpenSPARC T2 Multi-processor Systems (Under the direction of Dr. Rhett Davis).

A general procedure of verification is presented. Problems associated with verification of cache coherence are presented. Solutions of these problems are presented. More specifically, a global finite state machine approach of defining coverage is presented. A stimulus generation algorithm is presented to efficiently reach coverage goal. Necessity of such algorithm is discussed. Moreover, these two techniques are turned into detailed execution plan that fit into the test environment for OpenSparc T2 systems. Two checker schemes are presented. Advantages and disadvantages of these schemes are discussed. And efforts have been made to incorporate checkers into test environment.

Verification Strategy of Cache Coherence for OpenSPARC T2 Multi-processor Systems

by
Xu Zhang

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2013

APPROVED BY:

_____	_____
Dr. Rhett Davis Committee Chair	Dr. Eric Rotenberg
_____	
Dr. Yan Solihin	

DEDICATION

To my parents

BIOGRAPHY

Xu Zhang was born on July 21, 1989 in Taiyuan, China. In September 2007, he entered into Shanghai Jiaotong University and received the Bachelor of Science degree in School of Electronics and Electrical Engineering in July 2011. In August, he entered to North Carolina State University as a Master student.

ACKNOWLEDGMENTS

First I'd like to thank Dr. R. Davis, my supervisor, for his continuous help and encouragement. In the past 2 years I received guidance and suggestions which are far more than I could ever expect. Working with him is a real pleasure. Also, I want to thank his taking lots of time out of his busy life to help with this thesis. Without him this thesis would never reach its present form, let alone in such short time.

Second I want to extend my gratitude towards my committee member, Dr. Eric Rotenberg and Dr. Yan Solihin for many helpful discussions that I have with this. And I have to say this to Dr. Rotenberg, as his student in his computer architecture class, that you have the power to present pipeline structure in a way that's more magnificent and beautiful than Hunt Library.

Also I have to thank all my friends and colleagues, without whom my life here would be black and white.

At last, I have to thank my parents, which goes totally without saying, for being my spiritual backbone. They have proven that love and caring can go just as fast as the speed of light.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation and goal of the thesis

Functional verification of digital chips is becoming very difficult, due to increasing silicon complexity. Most designs today incorporate large portions of designs from other teams and even other companies scattered around the world, making designs extremely error-prone. Also, with rising fabrication costs, the cost of bugs in a chip is ever increasing. This thesis presents a verification plan for cache-coherency in multi-processor chips, which is fast becoming a necessary part of the systems-on-chip in nearly all consumer electronics.

Unfortunately, information on how to create a successful test plan for cache coherency is surprisingly scattered. The Application-Specific Integrated Circuit (ASIC) Verification Course [1] in our department (ECE 745) provides information on System-Verilog data types and object-oriented programming (OOP) concepts, which are the building blocks of any test environment, but provides no other guidance that is helpful for verifying cache coherency. Also, the Planning Session of Mentor Graphics' Verification Academy [6] gives us great planning advice from a high-level view to day-to-day execution. This session offers a nice skeleton, however it does not offer any specifics as to how to turn this into a fully-fledged execution plan. The literature of verifying cache coherency also provides fragmented guidance. For example, Rodrigues [2] et al. proposed a verification methodology of distributed memory by adding a sentry-cache to monitor every bus transaction. This paper, like many from the academic world, focuses on one of the many aspects of the environment but does not adequately show how its findings fit into the larger task of verifying cache coherency. This thesis synthesizes the ideas from the ASIC Verification course [1], Verification Academy [6], and two good references from the literature on verifying cache

coherency [3,5] in order to describe a more complete picture of what is required to verify cache-coherency in a multi-processor system.

This thesis presents a global finite state-machine- (FSM) based methodology for verifying cache coherency, embedded within a general procedure of verification. Furthermore, this thesis presents an example of how this procedure would be followed and how a detailed test plan would be created for an OpenSPARC multi-core processor to verify its coherence protocol. The central idea of this methodology is an approach to define and quantify test-coverage for cache coherency. This work also describes how to author the stimulus-generation and checker portions of verification environment which both depend heavily on the coverage definition. Ultimately, this work hopes to provide guidance to a new verification engineer who "doesn't know where to begin", so that he or she may avoid the missteps made by the author.

1.2 Proposed General Verification Procedure

Every test environment consists of three parts: coverage, stimulus, and checker. The steps below detail the approach to completing these three parts of the test environment and introduce the remaining chapters.

Step 1) Establish Verification goals

Remember that the end-goal is to verify as much as possible in the allotted time. Verification is really a job with no endings. For simplicity, our goal in this thesis is to verify cache coherence on OpenSPARC processor.

Step 2) DUT study

The next step in any verification project is a detailed study of the design under test (DUT), which is presented in Chapter 2. This study focuses on the cache-coherence subset of functionality. This study extracts a coherence protocol, signals, and timing information from the OpenSPARC specification

documents.

Step 3) Determine Granularity of the verification

The granularity is the level of abstraction we want to use when verifying the design. The highest level of abstraction is what the academic world calls formal verification. Formal verification is usually used to verify the specification itself. It has the advantage of being a simpler test environment. The lowest level of abstraction would be testing the design in a cycle accurate manner.

Since the design we want to verify was developed years ago and is unlikely to have cycle-to-cycle bugs, we would like to focus on the high-level and verify the cache-coherency protocol specification itself. At this stage, we would like to assume that the level-1 (L1) cache is ideal and focus the test environment to make the level-2 (L2) cache controller deal with coherence related issues.

Due to the time limit, in this document we build a test plan to verify the coherence protocol implementation assuming atomic memory operations. At the next level of granularity, we would normally build coverage for more realistic, non-atomic memory operations. Given sufficient time, test cases should be built to test interactions of sub-blocks, completeness of pipelining functionality, arbitration processes and even deeper operation corners. Some ideas are presented about how to approach verification of non-atomic memory access, but these are less fully developed. The background information presented in Chapter 2 introduces these ideas.

Step 4) Coverage

After the granularity is set, we can determine the coverage of the test environment. Chapter 3 presents our methodology's approach to define and quantify test-coverage, focusing on atomic memory operations, with some details about the approach would be expanded for non-atomic memory operations. Coverage is machine-understandable description of our verification goal. We can setup the coverage first, because coverage only depends on the information we gathered, but the stimulus and checker parts of the environment depend more or less on the chosen coverage groups.

There is bit of trouble translating a general goal like "verify that the cache operates coherently under

all circumstances" into pins and temporal logic that a machine can understand. In chapter 3, we present a general methodology that can be applied to generate coverage points for all complex systems. We also trim this methodology to create coverage for the OpenSPARC L2 cache controller.

The basic idea of this coverage generation methodology is simple and comes from [3]. First, we idealize next level subsystems into finite state machines and create a global state set consisting of all possible combinations of states in the hardware state machines. For example, the global states of a 2 core system with a Modified-Shared-Invalid (MSI) protocol can be represented with following table. We can name these global states as according to the combination of the states of each core.

Table 1-1 Build Global States using states of components

Global State	M	S	I
M	MM(error)	MS(error)	MI
S	SM(error)	SS	SI
I	IM	IS	II

Each of the combinations is called a Global State. Next we connect these states with edges that represent legal transitions between these states. In this context, we define coverage as travelling through each edge, forward and backward. Following the approach presented in [5], we further propose how this approach could be expanded for the transient states inherent in non-atomic memory operations.

Step 5) Stimulus Generation

Chapter 4 presents the stimulus generation approach in our methodology. Good stimulus generation

allows for closing coverage quickly. Therefore, this step can only be executed after the structure of coverage is settled. We would like to traverse the global state machine while limiting redundant visits to each edge to as few as possible.

Usually a combination of random test cases will do the trick. But for some systems, random test generation wouldn't be able to create efficient test stimulus. In this case we need to look into the design under test and try to come up with an algorithm to generate more efficient test cases.

In our case, for reasons will be introduced in Chapter 4, a random test is unsuitable for testing coherent shared memory. We adopt the technique presented in [3] that traverses the coverage structure on an Euler tour.

Step 6) Checker design

Chapter 5 presents the approach to designing the checker in our methodology. The checker structure can be developed in parallel with the stimulus generation strategy, after the coverage structure has been settled. The main reason behind building coverage first is to keep it simple and just check things that appear inside the coverage structure.

Since we used an FSM to create coverage, it makes sense to use the same FSM to check correctness of operation.

1.3 Graphic View of test environment

Figure 1-1 shows the test environment that is envisioned. Please refer to later chapters for further information regarding each part of this environment.

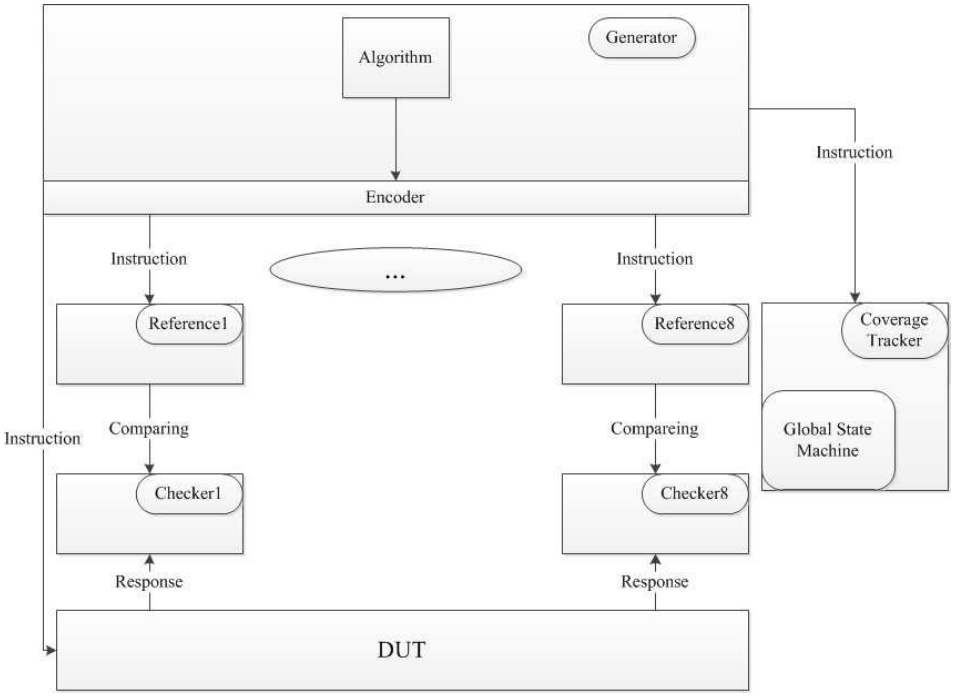

Figure 1-1 Test Environment Structure

1.4 Outline

The rest of this document is arranged as follows. Chapter 2 describes the design that is about to get tested. Important signals and setup information are studied and provided. In chapter 3 we develop coverage groups as goal of our verification effort. In chapter 4 we introduce the stimulus generation technique that will efficiently close the coverage. In chapter 5, we decide what checker structure should look like in our environment. We conclude in chapter 6.

Chapter 2

Background and Design under test study

This chapter begins with some useful background information on multi-core systems, and follows with the important details about the OpenSPARC T2 example system.

2.1 Background

2.1.1 Multi-threaded system

With the advancing of CPU clock frequency, delay of memory operations has become the bottleneck of the overall system performance. Caching of data has become an important technology to solve this problem. Among many other techniques which also try to solve this problem, multithreading is known for its effectiveness. Multithreaded systems, including traditional multiprocessors, chip multiprocessors and simultaneously multithreaded processors have come to dominate the commodity computing market. The most brilliant thing about a multithreaded system is that it can hide memory delay better than single threaded system. Single threaded systems can also execute more than 1 instruction per cycle. But as a cache miss often takes hundreds of cycles to resolve, execution lanes are often unable to find any independent instructions and the pipeline gets stalled anyway. With multithreaded systems, instructions from other thread are all independent from this unresolved thread and can execute freely. Figure 2-1 illustrates how individual threads will frequently stall while waiting for a memory access. Instruction-level parallelism (ILP) will frequently leave the processor idle, while thread-level parallelism (TLP) can allow the processor to execute useful instructions more frequently [4].

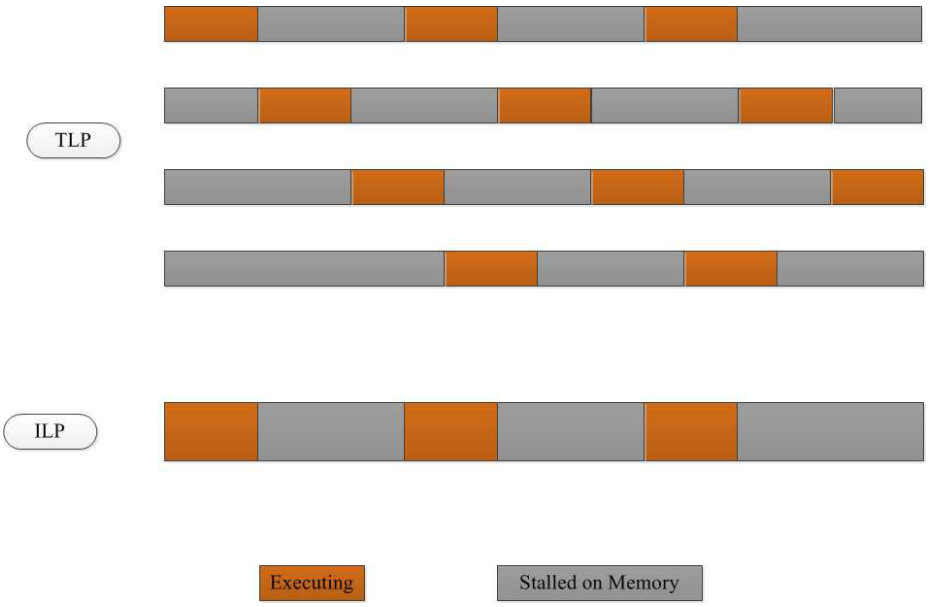

Figure 2-1 Comparison of effect on hiding memory delay between TLP and ILP system [4]

2.1.2 Introduction to cache coherence

A typical memory system for a multithreaded multiprocessor has a private L1 cache for each core and shared public L2 cache, as illustrated in Figure 2-2. This organization introduces the coherence problem.

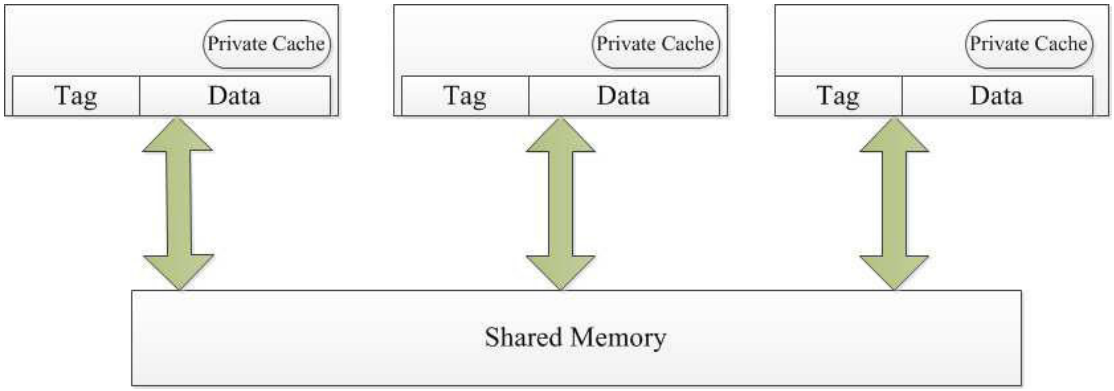

Figure 2-2 Memory system with distributed caches

For example, say core1 loads data from main memory. Then a core, say core2, stores to the same location. At this time core1 must be notified in some way, by either main memory or core1, that its data is

no longer usable.

The system has to be designed to provide the illusion that there is only one centralized memory and memory operations take no time to commit. There is a memory controller in charge of maintaining data coherency, which does its job by following a coherence protocol specification. Therefore verifying the coherence protocol usually means verifying this memory controller.

2.1.3 Verifying Cache Coherence

Cache coherence protocols are notoriously difficult to design and verify. A protocol description may specify only a few states, but implementation quickly become very complicated as states are added to allow hardware optimization and implement protocol optimizations. The complexity increases the possibility of subtle errors in the specification and low level implementation.

Figure 2-3 shows a simple MOESI FSM taken from [5], where the states for a cache line are Modified, Owned, Exclusive, Shared, and Invalid. Each cache line is initially invalid. Different memory operation causes transitions to other states. When a clean, read-only copy is present, the state is Shared. When a clean copy is present in only that cache, the state is Exclusive. If a dirty copy is present, the state is Modified. In this state, the data may be read or written by the processor. Finally, the Owned state allows cache-to-cache transfers of dirty data without updating memory. A simplified version of this state machine is described in more detail in Chapter 3.

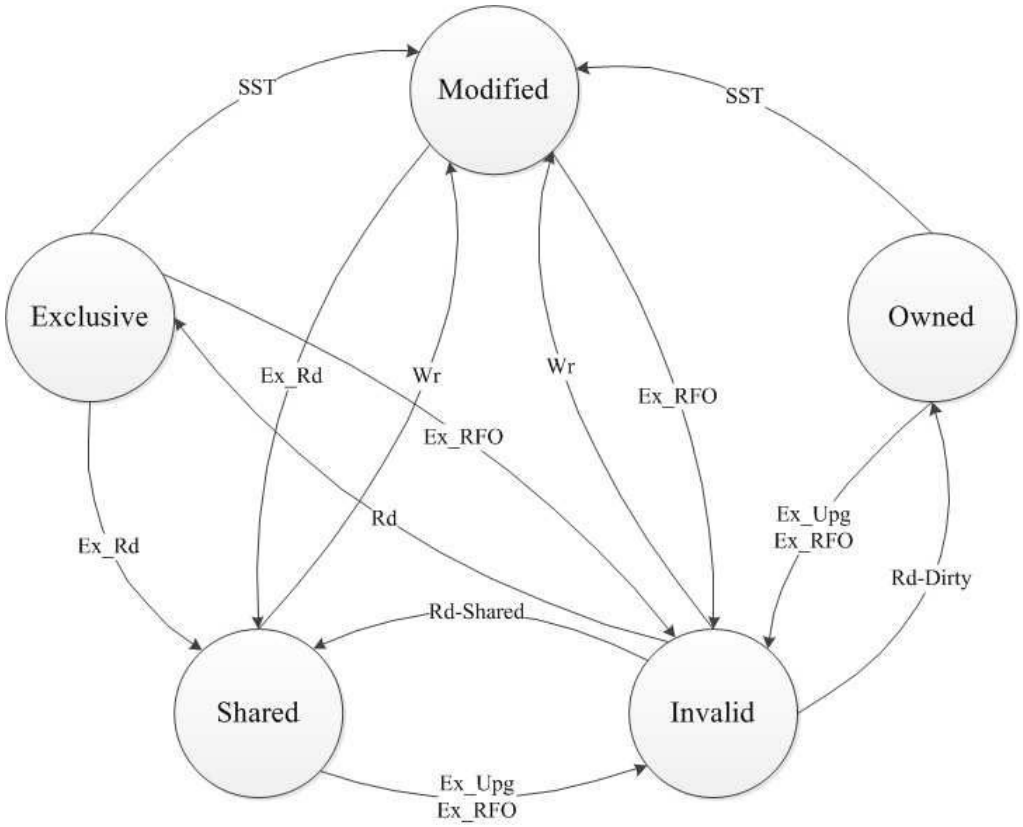

Figure 2-3 State diagram of an example MOESI protocol [5]

State transition diagrams such as the one shown in Figure 2-3 provide a nice, simple, high-level view of the protocol with atomic transitions between states. However, atomic memory operations severely limit the performance of a memory system. Most hardware-optimized systems rely heavily on non-atomic memory operations, which require many additional transient states to represent concurrency and make it so much harder to verify an implementation protocol. Figure 2-4 shows an example implementation of the MOESI protocol [5].

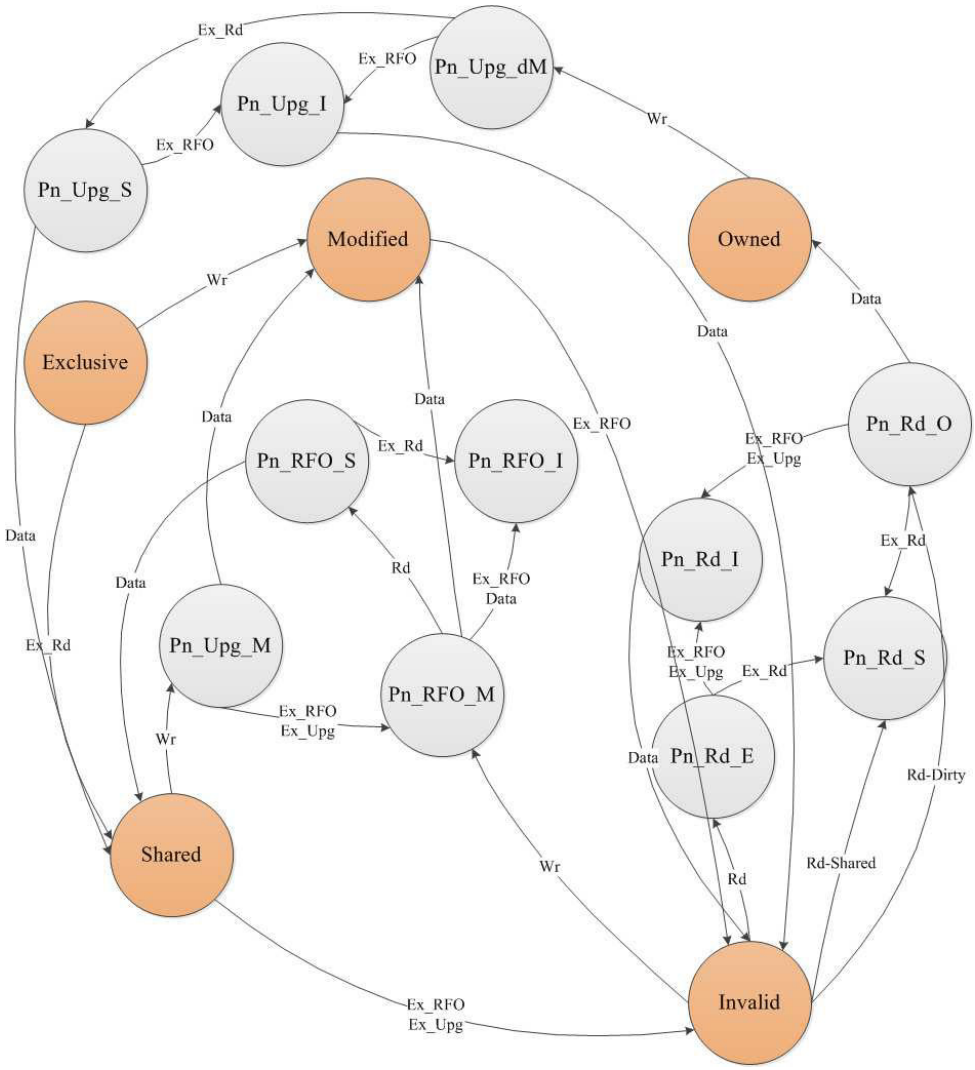

Figure 2-4 State diagram of implementation protocol of MOESI [5]

These additional transient states exponentially increase the state space. Light circles in the graph represent transient states of the system. Arcs between stable states and transient states mean incomplete instructions. For example, say a load request coming down from processing core and the cache line isn't there. Then this cache line is transferred into the transient state "Pn_Rd_E". After the data is fed from lower memory hierarchy, this transient state is then transferred into the stable state "Exclusive". This state diagram shows how the number of transient states goes up as coherence protocol gets more complicated. Also, their combination, the global states, goes up exponentially. With 16 transient states, an 8 core system would have 4,294,967,296 global states in theory. Even with 5 stable states, number of global states is

390,625. An Efficient stimulus generation algorithm would be helpful to complete simulation quicker, in other words, to verify more functionality in the allotted time.

2.2 DUT Study

The work applies the cache coherency verification approach to the freely available OpenSPARC T2 multi-processor design. This section presents the details of the OpenSPARC T2 system that are relevant verification of cache coherence, including the coherence protocol and its related input/output signals. This information is used in the remainder of the thesis to drive the level of abstraction chosen for the environment and the memory operations that need to be verified.

2.2.1 Protocol study

As illustrated in Figure 2-5 from the OpenSPARC T2 specification [4], each SPARC Core has its own L1 cache and these L1 caches are connected with a shared L2 cache through a cache crossbar which sends back and forward data and instructions between L1 and L2 cache. L2 cache consists of 8 identical banks and they are separated by address.

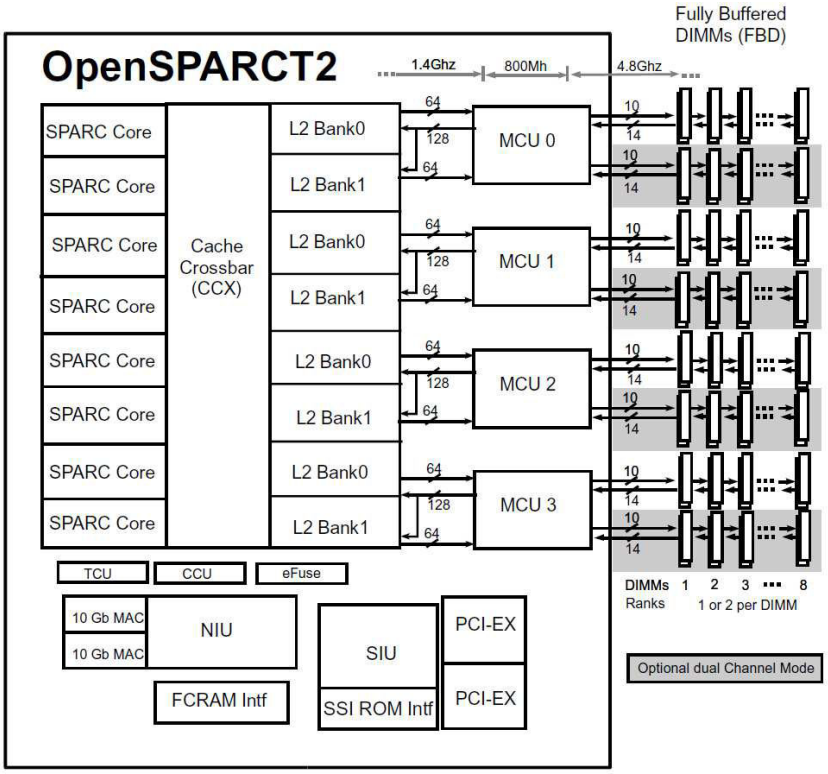

Figure 2-5 System diagram of OpenSPARC System on Chip [4]

The L2 cache controller has a copy of L1 tags from all cores inside a block called the directory. This directory also ensures that the same line is not resident in both the instruction cache (I-Cache) and data cache (D-Cache). This directory is written, according to the document, when a Load or a Store is performed. On data accesses, the directory is checked to determine whether the data is resident in L1 cache. The result of these checking operations is a set of matching bits that are encoded to create an invalidation vector to be sent back to SPARC cores to invalidate L1 lines.

On the other end, there are 2 sets of bits that indicate the states of an L1 cache line: dirty bits and valid bits. The simplest way to extract this into a state transition diagram is with the well-known MSI model. Table 2-1 depicts how the combination of these two bits constructs the MSI protocol (M is for “modified”, I is for “Invalid”, S is for “Shared”).

Table 2-1 Mapping actual state bits to protocol states

	Valid bit value	
Dirty bit value	I	V
D	I	M
~D	I	S

Figure 2-6 depicts the relationship between these states, simplified from a similar diagram from [3] to remove arcs back to the same state. In the next chapter we'll use this to construct a global state machine for this design and then coverage structure. The following terms are used in the graph:

SLD: Self Load, load instruction or data for the L1's own processing unit

SST: Self Store, store instruction or data for the L1's own processing unit

Evict: Line evicted from L1 Cache

OLD: Other Load, load issued by another processing unit

OST: Other Store. Store issued by another processing unit

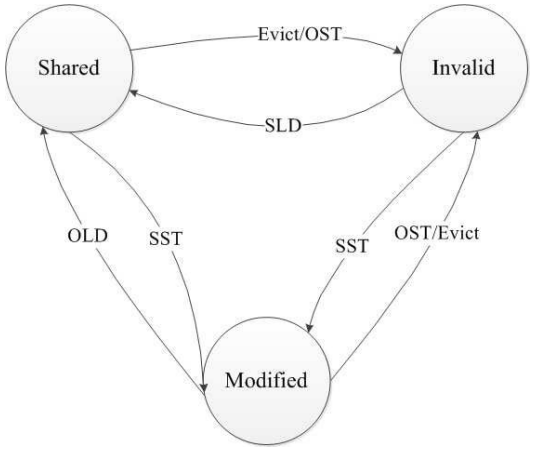

Figure 2-6 Coherence protocol of DUT

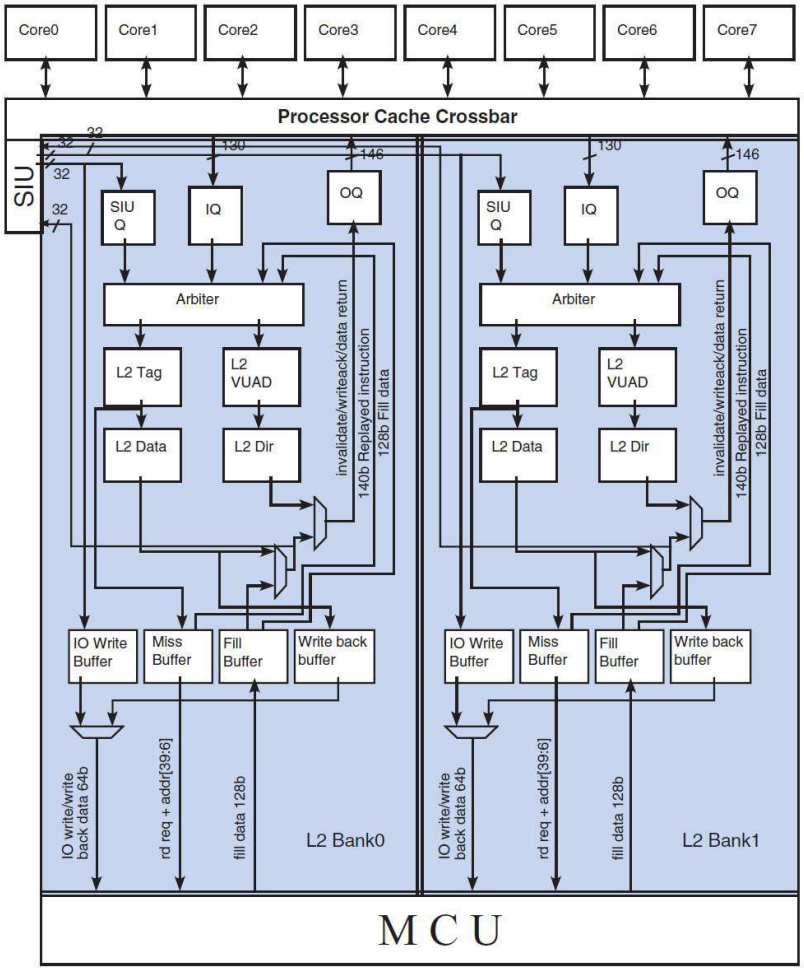

Figure 2-7 Sub-blocks of L2 cache controller [4]

Figure 2-7 illustrates the L2 cache logic in more detail. It is very tempting to look at the internal structure of the L2 cache controller and to study these buffers and arbitration units. However, studying this logic will cause lots of unnecessary pain. The important details are the timing of input and output signals.

2.2.2 Input Output signals to be checked

Figure 2-8 illustrates the protocol signaling between the L1 and L2 cache controllers. The signals that the L1 sends to the L2 cache controller are pcx_l2t_data_px2 and pcx_l2t_data_rdy_px1. The L2 input interface receives the data valid signal (pcx_l2t_data_rdy_px1) followed by the data (pcx_l2t_data_px2) in the next cycle. The request thus received is decoded into address, data and instruction fields in the PX2 pipeline stage and is forwarded to the arbiter logic to request access to the L2 cache memories to process the request.

Figure 2-8 also illustrates the coherence-protocol signaling between the L2 cache controller and the L2 cache memories. The L2 controller sends output signals L2t_cpx_req_cq, cpx_l2t_gnt_cx, and l2t_cpx_data_ca. These signals include invalidation signals and return data. The L2 controller sends a request (l2t_cpx_req_cq) out if it has a packet to be dispatched. The packet may be return data for load requests, acknowledgments for stores and invalidates for evictions and stores. Output data from the L2 memories is carried on the signal called l2t_cpx_data_ca. If the packet is consumed by the crossbar, an acknowledgement is received in the next cycle on the signal cpx_l2t_gnt_cx.

The signals described here are the main connection points for a cache-coherency test environment. Traditional verification methods will verify that the L1 and L2 cache controllers each adhere to the coherence-protocol specification. This thesis is concerned with the harder question of whether or not the L2 cache controller can adequately guarantee coherency, even when the protocol is perfectly observed.

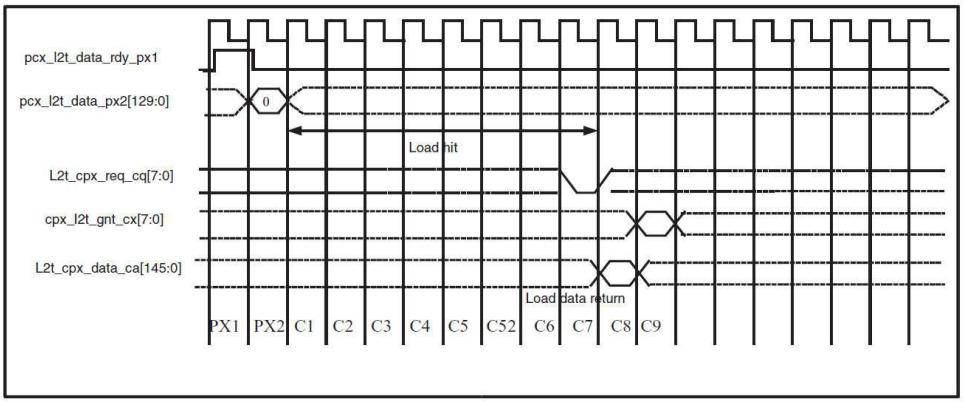

Figure 2-8 Timing information for signals between test environment and L2 controller [4]

Chapter 3

Coverage Design

3.1 Introduction

Coverage design is the process of setting a goal for verification effort. Designing verification coverage should be the first concern of every verification team. Carefully designed coverage groups should represent all circumstances that the DUT is going to face. This is not going to be easy, especially when verification people have to translate the unclear term “all circumstances” into machine understandable terms, instruction streams, temporal constraints, et al. Nevertheless, this is a definition that is better to be defined along with process of development of the DUT, otherwise it will be too much work to be and small operating corners are easily omitted.

Designing coverage groups for a coherence protocol is particularly hard, because the protocol operates in a distributed system. There is no centralized control unit or traditional pipeline structure, both of which have well-known approaches for defining thorough coverage groups.

Extended from the simple idea to represent a distributed cache in the form of an FSM, the global state can be represented in the form of elements of the Cartesian product of each individual cache’s states. Furthermore, we can define complete coverage as “the system is doing what the spec says it supposed to be doing”, “at any global state”, “under any instruction”.

This chapter is structured as follows. The 2nd subsection introduces the global state structure and the coverage derived from it. It should be noted that this coverage is introduced under assumption of atomic state transition. The 3rd subsection deals with the more realistic case where non-atomic memory accesses can occur, leading to a finite state machine that includes transient states that occur frequently.

3.2 Transition Coverage Model

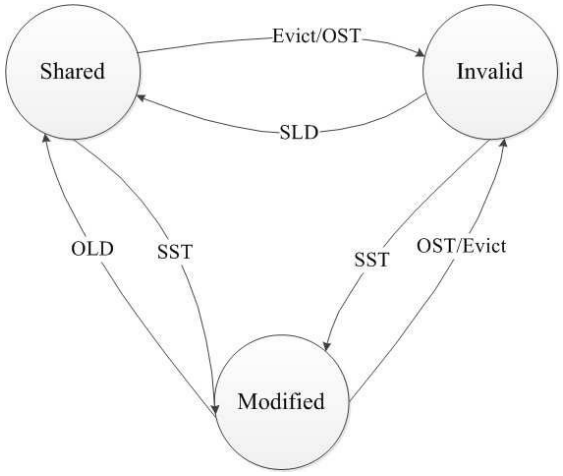

Figure 3-1 State transitions of single private cache (simplified from [3])

As described in the previous chapter, the OpenSPARC T2 uses an MSI protocol across 8 cores. Figure 3-1 above is repeated from chapter 2 and shows the relationships between states of single cache line. To get a clearer view of relationships of states between different cache lines, the so called Global States, the following graph introduces the coverage with 3 cores, which can be represented as a 3 dimensional graph [3].

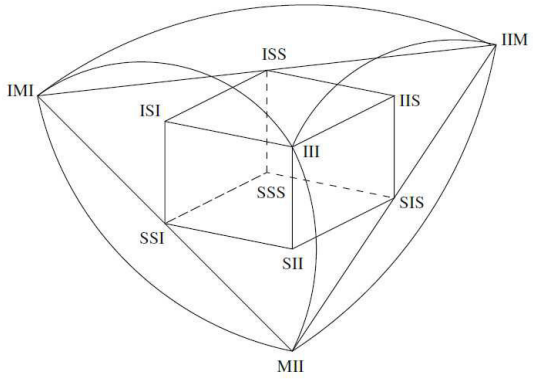

Figure 3-2 Global State structure of 3 core cache system [3]

Figure 3-2 represents all possible global states for a 3 private cache system. Each point in the graph represents a global state. Take state point SSS as an example, it means all 3 private L1 caches have allocated the same L2 cache line, and all are in the shared state. Here we illustrate how the global state evolves as instructions are executed. First of all, the initial state should be III, implying that this L2 cache line is invalid for all L1 caches throughout the entire system. Then as we start our simulation, the first instruction is either a load or a store. In the case of a load, one of the cache line states should be changed into S, so the global state should be changed to among SII, ISI, or IIS. Therefore, there should be 3 new global states SII, ISI, and IIS and they should all connect to the initial state III. In case the first instruction is a store, then one of the cache line states should become Modified and others should remain Invalid. Hence the global state should become one of the states IIM, MII, or IMI, and they all should be connected to the initial state III. Note in the graph that transitions into these three global modified states have been omitted for simplicity if there is no corresponding transition back into the originating state. It is assumed that a global modified state can be reached from any other state when a store operation is reached.

Say the second instruction is a load from a different core. The global system state becomes SSI, ISS, or SIS. There we should add 3 new states called SSI, ISS, and SIS, which should be connected to the previous 6 states (IMI, MII, IIM, SII, IIS, ISI).

Finally if the third core executes a load operation, then the global state becomes SSS. After examining all possible combinations of memory operations, we find that this graph defines all the legal global states and legal transitions between these states.

Naturally the next step is to check for completeness of the coverage structure. There is one way to do this. Start from an arbitrary global state and execute any legally permitted instruction. There should be an existing edge representing this transition or there should be no transition at all. This way of checking completeness of coverage structure is somewhat intuitive. Under this definition, the coverage structure described above is complete. Coverage can be defined to cover all the edges of the graph, forward and backward.

The state-transition diagram in Figure 3-2 must be extended to 8 cores in order to model the OpenSPARC T2 design. Table 3-1 lists the relationships between each global state and a graph

representation. Figure 3-3 provides a simple illustration of the state-transition diagram. A simple explanation of notions used in the table and figure is provided below the table.

Table 3-1 Global state structure of OpenSPARC cache system

	Load	Store	Evict
GI: (I[8])	GS	GM	N/A / Error
GS: (S[n=1~8], I[8-n])	GS	GM	GS/GI
GM: (M, I[7])	GM/GS	GM	GI / Error

GI: Global Invalid State, this state means this cache line is invalid for all 8 cores.

GS: Global Shared State, this state means that 1 or more private L1 caches share this cache line, others either don't have that line or that line is in invalid state.

GM: Global Modified State, this state means 1 and only 1 cache has a copy in the Modified State, others either don't have that line or that line is in the invalid state.

Note that GS and GM stand for a category of global states. The cache-line can be called "in the GS state" (or "state category") when it is in any of the shared global states. Likewise, if the cache line is in any one of the modified global states, the system is "in the GM state" (or "state category"). This also means the system may go through global state changes and still remain in the same category. This means that many of the functions carried out by checkers and coverage trackers won't show up on the diagram.

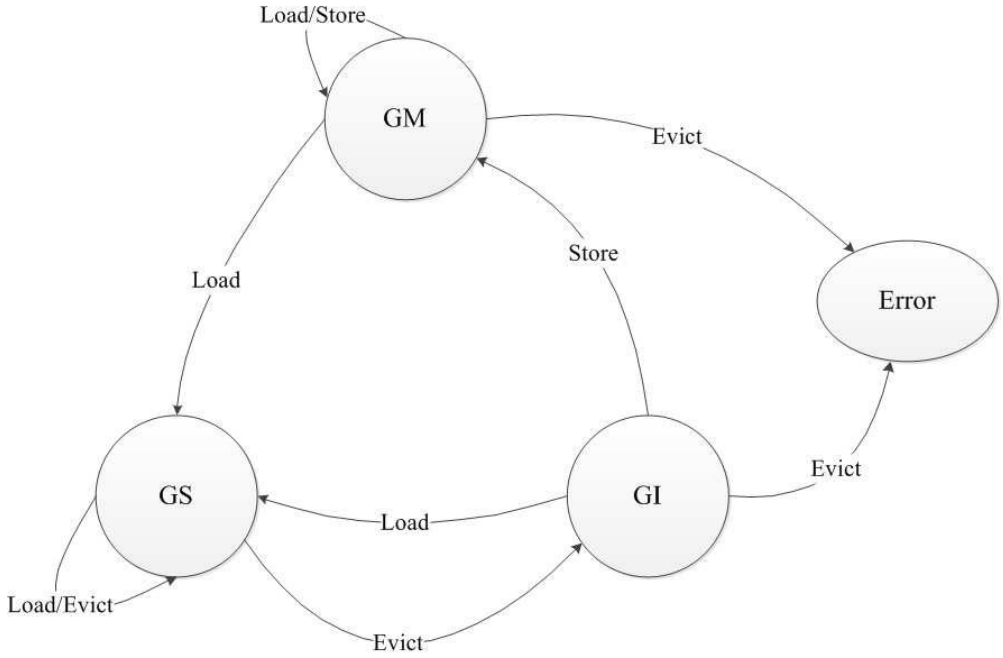

Figure 3-3 Global state structure of OpenSPARC cache system

3.3 Transient states of this design

Using the global state definitions above, it is possible to derive a stimulus generator and checker to achieve 100% coverage for atomic memory operations. These will be explained in the later chapters. But what about non-atomic memory operations, which are much more the normal in multi-processors? What do we do when more than one instruction are flying inside the system? How does the system cope with this? Following the approach in [5], several transient states are added to the single private L1 FSM in Figure 3-1 to adapt it to real life verification needs.

Figure 3-4 illustrates the modified FSM. First, a transient state is added to describe the state that a private cache is waiting for its data, called Pd_Rd_S. This state is the most likely to be visited as a transient state. Every load miss of L1 private cache will cause cache line to transfer from Invalid to this state. This cache line will stay at this state until either data comes from the lower memory hierarchy, or another L1 private cache writes to the same location, in which case this state transits to a new transient state and waits for an invalidation vector to transition to an Invalid state. Bear in mind, that (stale) data is

still needed for processing although it never commits to the data cache.

In case a store happens later with that same core, data still needs to feed to the processing unit, but this is going to happen through another data bypass path. Accordingly, data commits to both the L1 private cache and the L2 shared cache. An invalidation vector is generated to invalidate other copies of this line. The L1's own cache line enters the Modified state. So there isn't going to be a transient state between Pd_Rd_S and the Modified state.

When a cache line is in the Modified state, there is the possibility that another core writes the same cache line before receiving notification that its cache line has been invalidated. To handle this case in the coverage model, we want to create a new transient state for that "waiting to be invalidated" case and setup all the transitions towards other stable states. We call this state Pd_Rd_I.

When a cache requests data that currently resides in another L1 private cache, a feed through request is generated by the L2 controller. After the data arrives at the L2 cache, the L2 cache then feeds the data back to the requesting core. Also, an acknowledgement is generated to the core that provided the data, and its state changes from Modified to Shared. A third transient state Pd_ack is added for this case.

Table 3-2 below lists every transient state, its name, use, and relationships with stable states and other transient states.

Table 3-2 State transitions between states of OpenSPARC system

(Initiative)					
Stable states	SST	OST	SLD	OLD	EVICT
Modified	Modified	Pd_Rd_I	Modified	Pd_ack	Invalid
Shared	Modified	Pd_Rd_I	Shared	Shared	Invalid
Invalid	Modified	Invalid	Pd_Rd_S	Invalid	N/A
Transient states					
Pd_Rd_S	Modified	Pd_Rd_I	Pd_Rd_S	Pd_Rd_S	N/A
Pd_Rd_I	Modified	Pd_Rd_I	Pd_Rd_S	Pd_Rd_S	Invalid

Table 3-2 Continued

Pd_ack	Modified	Pd_Rd_I	Pd_ack	Pd_ack	N/A
(Responsive)					
Transient states	Data	Inv_vec	ack		
Pd_Rd_S	Shared				
Pd_Rd_I		Invalid			
Pd_ack			Shared		

Here we define the terms used in Table 3-2:

Initiative transitions (transitions originating from the L1 cache-initiated operations):

SLD: Self Load, load instruction or data for the L1's own processing unit

SST: Self Store, store instruction or data for the L1's own processing unit

Evict: Line evicted from L1 Cache

OLD: Other Load, load issued by another processing unit

OST: Other Store. Store issued by another processing unit

Responsive transitions (transitions originating from the L2 memory array response)

Inv_vec: Invalidation vector generated by L2 controller to invalidate cache lines to maintain coherency

Data: Data returned from L2 cache

Note that even with only these three basic transient states added to the system, it is already showing a sign of exploding state space. It is quite hard to explore every combination of states across all cores. In this case, a much more practical way to define coverage would be to cover each transition between private L1 states, ignoring transitions between global states. Assuming that all atomic memory operations are verified with the global FSM coverage definition in the previous section, it may be reasonable to ignore global transient transitions, given the symmetry in the architecture and, hopefully, the lack of possible bugs

between global transient states.

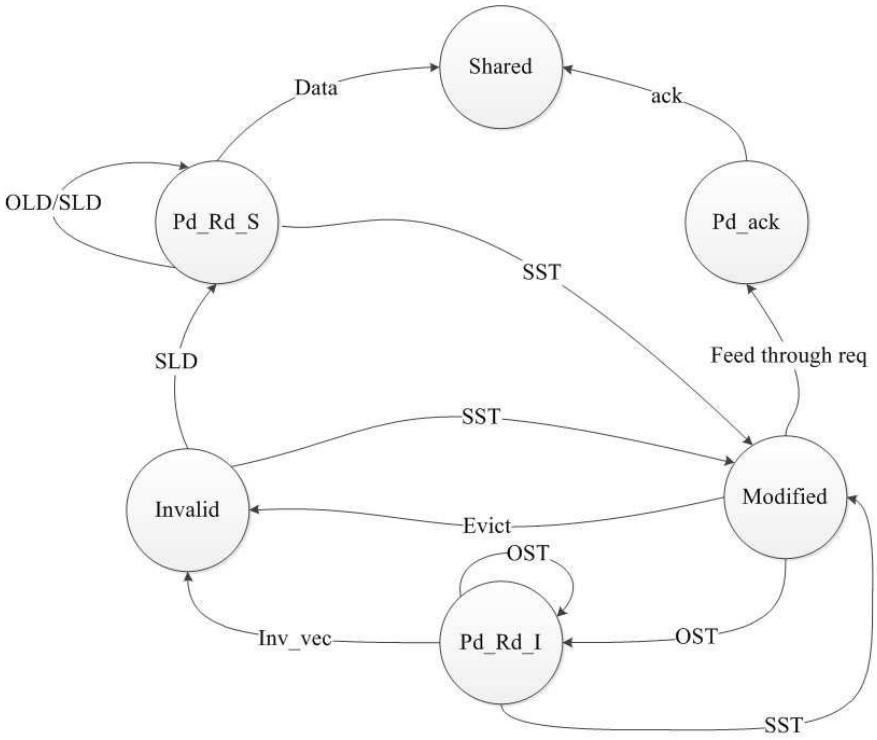

Figure 3-4 Global state machine with transient states

Chapter 4

Stimulus Generation

4.1 Introduction

The constrained random approach to stimulus generation is still the most popular technique in practice. This approach fills the gap of efficiency between random tests and directed test cases. For this reason, constrained random test generation should be tried first. If its performance is acceptable, then there is little incentive to seek a better method. Unfortunately, constrained random stimulus generation is not a particularly good idea when it comes to verifying a distributed memory system for following reasons:

a) Instructions will splash all over the memory address space. That makes it too hard to have a cache line stay in more than 2 private caches simultaneously with each one of them easily kicked off by the very next instruction. This limitation will lead to the need for a huge number of test vectors to achieve a small amount of coverage. The goal of checking coherence is essentially to see if there is a stale value, or to see if a cache line gets invalidated quickly enough to avoid errors. These situations will not occur if a cache line is evicted prematurely.

b) Using the coverage definition from the previous chapter, the goal of verification is to cover all the global state transitions. The tree structure nature of these states will cause random stimulus to hit early states way too much, while states hidden deep in the tree will be almost entirely avoided. This limitation will also lead to the need for a huge number of test vectors.

Given these limitations, we need algorithms to generate stimulus more efficiently and close coverage quicker. To give a more clear view on how random stimulus closes coverage. We did a little experiment using random stimulus to try to cover MSI protocol for 8 cores, which is just like the coverage structure.

Figure 4-1 depicts the result of this experiment. Source codes are appended at the end of this paper. Note that total state transitions of MSI coverage structure is 5256. From this figure we can see that random stimulus costs on average 32 instructions per state transition to cover the global state machine. Moreover, efficiency of random stimulus gets lower as coverage gets close to 1. This tells us the following: Efficiency of random stimulus will get unacceptably low with state machine with more states or more cores. At some point, direct test cases will cost engineers less time trying to close such coverage structure even with some extra coding time.

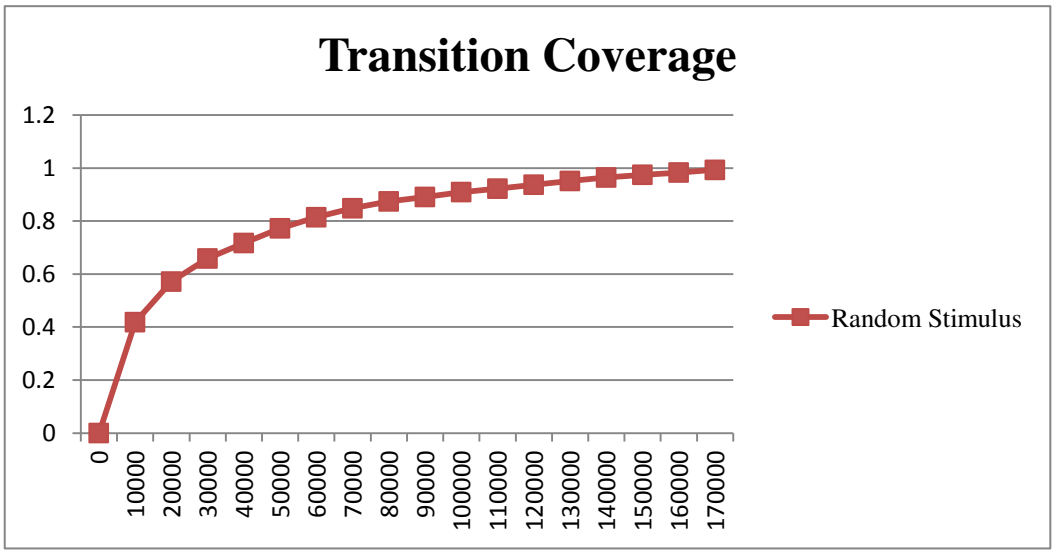

Figure 4-1 Coverage vs. Instruction count using random stimulus

4.2 Test Generation for Transition Coverage (TGTC)

Fortunately, Qin and Mishra, who formulated the coverage definition described in Chapter 3, also presented an algorithm for stimulus generation [3] that can

a) Take an Euler tour through the whole FSM, and

b) Generate test cases on the fly

The second point is important, because storing stimulus is not wise. Simulation often takes an extremely long time to run, and preparing instructions beforehand requires huge memory space to store

them. The test generation for transition coverage of cache coherence protocols presented in [3] is motivated by a basic breadth-first-search in the state space of a global FSM. Given the FSM description of any cache coherence protocol, it is possible to compose a test case that can reach all states and transitions using two steps. First, from the current states of the system, find a path to reach a state that has not yet been reached. Second, apply the instruction sequence to the machine and see if the system behaves as expected.

However, such intuitive method suffers inefficiency due to the following reasons. First, the initial state and states close to initial state are visited way too many times. Simulation time is wasted revisiting them repeatedly. Secondly, for states that are deep inside the coverage structure, a routine that finds path to it requires significant amount of time and memory, which is not the most productive way of using these resources.

Fortunately, Qin and Mishra noticed that the cubic structure of the global FSM has good symmetry and designed an algorithm to exploit that symmetry. The algorithm is divided into two parts. First an algorithm that traverses a simple shared-invalid (SI) protocol is introduced. Then a similar algorithm that traverses MSI protocol coverage structure is introduced. The MSI algorithm and the SI algorithm share the same basic idea of recursive execution. The MSI algorithm calls the SI algorithm during its execution process to make things easier for programmers and readers of the code. Moreover, this structure makes presenting this algorithm easier.

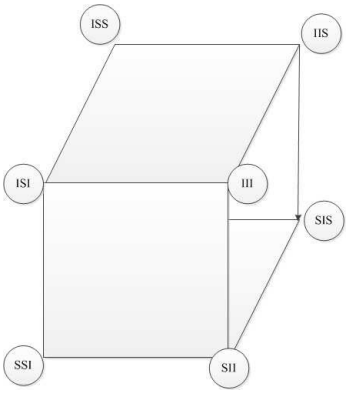

Figure 4-2 Global State machine of SI protocol

Algorithm1: Test Generation for SI protocol with n cores

CreateTestsSI(n)

```
CreateTestsSI(n){
        for(int i =0; i < n; i++) {
                load(i);
                VisitHypercube(1, n-1, i);
                evict(i);
        }
return;
}
```

VisitHypercube(id, m, shift)

```
VisitHypercube(id, m, shift){
int power = 1;
        for(int i = 1; i <= m; i++) {
                power = power + power;
                int newid = id + power;
                int p = (i + shift) % n;
                load(p);
                if(i > 1) VisitHypercube(newid, i-1, shift);
                evict(p);
        }
return;
}
```

Figure 4-3 TGTC for SI protocol

The SI protocol is not a realistic protocol for the fact that cores are not allowed to perform store operations. In other words, the data is read only. Legitimate global states include any number of private caches in Shared state and all others in Invalid State. So for a system with N cores, the total number of legal global states are 2^N. Since each global states can be transferred into N other states by toggling the state of one of the cores. Each global state should have N connections with its direct neighbors, essentially forming an N-dimensional hypercube. Figure 4-2 illustrates this FSM for the case of a simple three processor system, redrawn from a figure in [3]. Each edge of the hypercube is bidirectional. Test cases need to traverse the graph from both directions to achieve full coverage. Moreover, we want repeated visits to the same edge to be as few as possible.

Figure 4-3 shows Algorithm 1 from [3], rewritten using the C language syntax. Algorithm 1is the TGTC algorithm for the SI protocol, which performs an Euler tour on an n-dimensional hypercube. Here, load(p)/evict(p) means that the p-th core performs a load/evict operation in a particular cycle, while all other cores remain idle.

The whole algorithm adopts a Hanoi tower trick. The initial state is III, which is the global invalid

state. In the main function CreateTestsSI(), the algorithm sets the first core into Shared state. The algorithm then calls VisitHypercube() to traverse the rest of the hypercube with that core remaining in the Shared state. Similarly, the function VisitHypercube() recursively calls itself with the task of traversing a smaller portion of the hypercube. For example, we present a sample state transition sequence with one call of the function VisitHypercube() with N=3 cores. III-ISI-SSI-ISI-ISS-SSS-ISS-ISI-III is a sequence of global state with the VisitHypercube() function called with shift=1. Notice that second core remains Shared while several states of other 2 cores are traversed, and then finally the second core gets reset to Invalid.

The next step is to traverse the hypercube structure of MSI protocol. Here we repeat the figure from Chapter 3 for convenience.

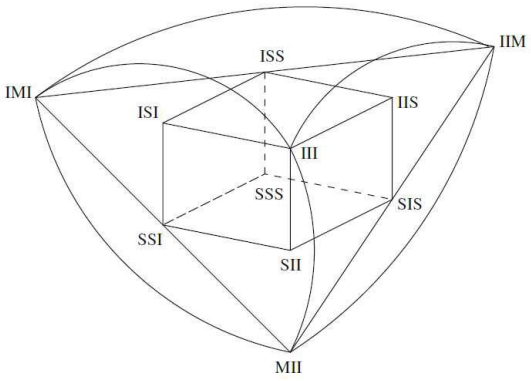

Figure 4-4 Global state machine of 3 core MSI protocol

Algorithm2: Test Generation for MSI protocol with n cores

CreateTestsMSI(n)

```
CreateTestsMSI(n){
        CreateTestsSI(n);
        VisitClique(0);
        for(int i =0; i < n; i++) {
                load(i); // Visit each GS
                for(int j =0; j < n; j++) {
                        store(j);
                        // Find a path to current state
                }
        }
return;
}
```

VisitClique(p)

```
VisitClique(p) {
        store(p);
// Reach all reachable GSs
        for(int i = p+1; i < n; i++) {
                store(i);
                if(i == p+1) VisitClique(i);
                store(p);
        }
return;
}
```

Figure 4-5 TGTC for MSI Stimulus

Algorithm 2 from [3] generates stimulus for the MSI protocol, as shown in Figure 4-5, again translated into a C-language syntax. Algorithm 2 first calls function CreateTestsSI() to traverse the global shared states, which is a subset of the coverage structure. In addition, Algorithm 2 has to traverse state transitions that involve Global Modified states. These transitions are grouped into 3 categories. The first category includes transitions from one Global Modified state to another. The second category includes transitions that are bidirectional between Global Shared states and Modified states, an example being transitions between SII and IMI. The third category includes transitions that are one way towards Modified states. For example, system can go from IIS to IIM by performing a store operation but cannot go back from IIM to IIS in one operation.

The algorithm is broken down into two parts as well. First, the main function calls function VisitClique(). This function starts by visiting all transitions that are bidirectional between Modified states and reachable shared states, and finishes by visiting all transitions between Modified states. After

VisitClique() returns to the main function, all transitions that are left unvisited are one way transitions. Then the algorithm goes on and traverses those transitions. This is done by visiting every Global Shared states then performing a store.

The next step is to sequence the stream of memory operations produced by the algorithm in order and send them out in a timely manner that can be controlled. Moreover, some trivial work needs to be done to convert these abstract memory operations into toggling the pins listed in section 2.2.2. This portion of the structure will be shows in Chapter 6.

4.3 Fake atomic operation

Algorithm 2 described above generates stimulus suitable for verifying atomic memory operations, but most memory accesses in the OpenSPARC system are not atomic. We can still use Algorithm 2 to generate stimulus for the OpenSPARC system, provided that we assume “fake atomic operation.” The basic idea is that we give each operation enough time to complete, and then execute the next operation. In order to do that, the test environment needs to know how long it takes before each and every instruction commits. To address this issue, we studied the pipeline behavior of Load / Store instructions. Particularly we want to know the number of cycles between a Load request and (if there is a hit) the arrival of its data at the interface or (if there is a miss), the forwarding of the request to the Main memory interface. [4]

TABLE 2-1 Pipeline Diagram: Load Hit

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag, VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen, xmit VUAD ECC check	way sel xmit in l2d	data array read cyc1 FB data read cycle Xmit inputs to directory	data array read cyc2 stage FB data D$ directory write I$ directory CAM VUAD write	data array read cyc3 4:1 mux mux with FB data	data xmit cycle gen inval. vector	request to the dest cpx queue check ECC on data	Mux Data/Inval. Vector data return to dest. cpx

Figure 4-6 Pipeline delay of a hit [4]

TABLE 2-5 Timing Diagram: Miss

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read tag compare MB CAM perform store data ECC (for store) VUAD bypass FB CAM WBB CAM	way sel logic, check VUAD ECC	write MB tag set MB valid bit					request to the dest cpx queue (ack for write in case it is a store miss but hits in one or more MB store miss entries)	

Figure 4-7 Pipeline delay of a miss [4]

Luckily, for verification of a coherence protocol, we don't have worry about the details of L2 misses or forwarding data (other than fake data), because what we want to verify is correct operation amid conflicts between multiple copies of a same memory address. As shown in the pipeline diagrams for L2 hits and misses, shown in Figure 4-6 and 4-7, there are 9 pipeline stages (labeled C1 to C8, note that the C5 stage has 2 cycles) , plus the PX2 stage from crossbar interface (not shown). There are no responsive

interfaces or arbitration within this pipeline, so stalls are not a concern. Therefore, it is a safe bet that any instruction will finish within 10 cycles, and the stimulus generator should be designed to execute operations with this frequency.

Chapter 5

Checker design

5.1 Introduction

Checkers are often an insignificant part of the test environment for processing units. Checkers are responsible for checking correct operation, which is simple because for processing units correctness can be easily checked by an instruction-set simulator. But for checking correctness of coherence, the situation is slightly different. This chapter is structured as follows: first we introduce FSM checkers, and then we introduce a simple checker. In both of these sections we not only introduce the structure and principles of these techniques but also their pros and cons in order to guide decisions about how the test environment should be structured.

5.2 FSM coherence checker

Every cache line can be modeled as an FSM, so it is intuitive to check base on a simple assumption that each L1 private cache is a FSM. The primary aspect of an FSM checker is that no data is actually generated or changes hands (between private L1 caches or between private L1 caches and the shared L2 cache), which leads to the following advantages and disadvantages:

a) Advantage: greatly reduced traffic going through interfaces, leading to a simpler test environment.

b) Advantage: this checker can detect subtle protocol errors. Say data sits still with correct value but the wrong state. For example, consider the case of data that should have been invalidated but was not invalidated, yet its value happened to be the same as the new value.

c) Disadvantage: that the data in memory isn't actually checked, which could be a possible bug that isn't covered. Data arrays are rarely buggy, though.

d) Disadvantage: FSM checkers have their own logic inside themselves, which means that this part of the test environment must be amended each time a new design under test arrives with a new coherence protocol

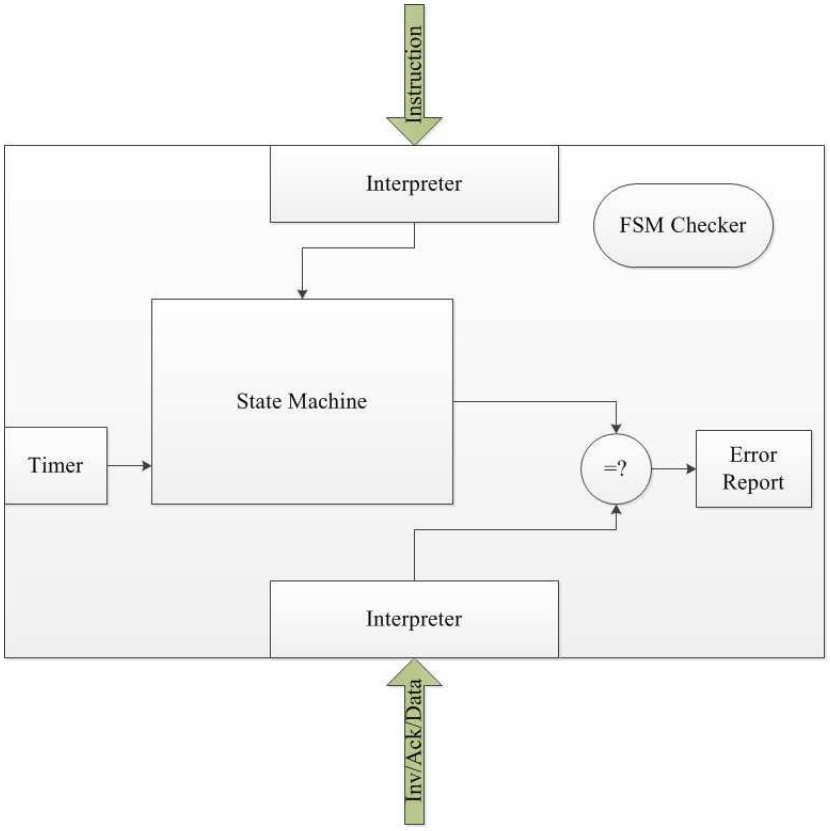

Figure 5-1 Diagram of FSM checker

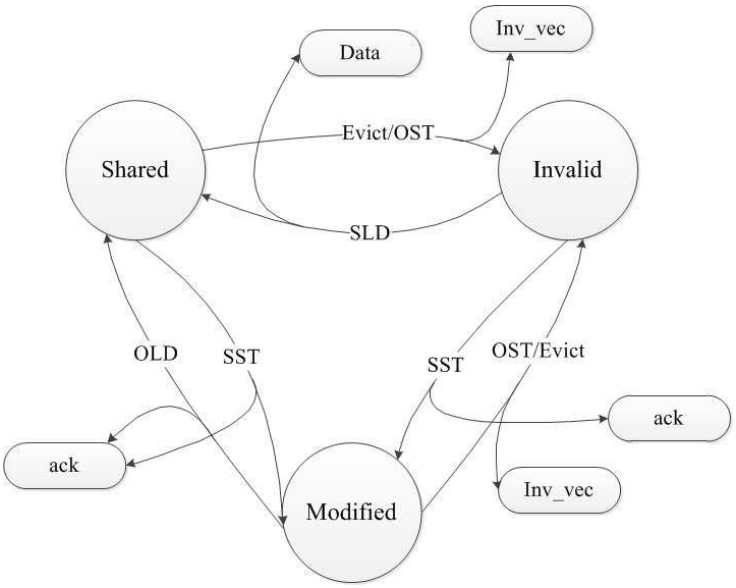

Figure 5-2 FSM machine structure inside the checker

How do FSM checkers work? First, an FSM model is constructed using the results of the protocol study. We use only the stable states (no transient states) to simplify the FSM design. Then we connect 8 checkers, each associated with a core to check its operations. That is, we capture all transactions that go into or out of these cores. Then we update states and check transactions using the current state of the that the checker FSM. Figure 5-1 illustrates the structure of an FSM checker module, while Figure 5-2 illustrates the FSM within the checker module.

Each FSM checker is connected to the logical bus using an analysis port and has a tag register, which stores the tag of the current cache line and prevents unnecessary state update. Whenever a transaction sets in, tags are checked first. If the tag in the DUT matches the tag inside the FSM checker, then we use a register to store the current cache-line state. Then we use the FSM algorithm to decide what the next state should be and what action this core should take. For example, if a read is missed and data arrives from the L2 cache, the next state of this FSM should be Shared and the action should be to do nothing. If the design under test repeats a load request, then the test environment is going to report a bug.

If transient states are added to the verification coverage goal, then we don't need to add these transient states to checkers, because we only check committed transactions. Moreover, adding transient states will make our already protocol-dependent checkers even less reusable.

A watchdog timer is a necessary helper block with FSM checkers. This is because we check only committed state transitions after 10 cycles (following the “fake atomic operation” stimulus described in the previous chapter). It may take some time for operations to commit, and as long as these transitions happen within the allotted time limit, we regard the system operations as legal.

5.3 Simple checker

The Simple checker approach is based on a more intuitive way of checking correctness: when memory is atomic and centralized, a coherence protocol is not going to be needed. With this approach, we construct a simple memory model that captures all transactions of the logic bus and update its data immediately. If the data in the DUT doesn’t match the data that lies inside the reference memory, checkers report a bug. There are some pros and cons associated with simple checker structure.

a) Advantage: Greatly simplified checker structure allowing developers devote their energy to other and more important aspects.

b) Disadvantage: Increased traffic through interfaces, due to the fact that data must be tracked, meaning that less stimulus can be applied.

c) Advantage: Protocol invariant, meaningless development time and more reusability

If a simple checker is used, then the next question would be “how should the data look?” A randomly generated data field should be considered first, for its simplicity. But random number generation costs system resources, and there is slight chance that a bug would not be caught by the test environment. Instead, we generate data like a time stamp. We generate first instruction with a data field equal to 0x0001, then increment this data until we reach 0xffff. This way, in the case that a bug is detected, we get some detailed information about the time mismatch between operations and/or which operations were omitted by the system.

In order to verify cache coherency in the OpenSPARC T2 system as quickly as possible, we would adopt the simple checker. Given more development time, we may choose to use the FSM checker approach, due to its ability to achieve more bug coverage with less simulation time.

What does the simple checker look like? Figure 5-3 illustrates the simple checker structure. The Simple Checker, by definition, does its job by comparing what actually lies inside the private L1 caches against what's in each checker's reference memory. In order to do this, these reference memories should be alongside the checkers. But they get their information from different resources. Checkers update their cache lines by signals sent back from our DUT, the L2 cache controller. On the other hand, the Reference memories in each L1 caches' checker gets their information directly from the generator, along with the information sent to other cores. Then each Reference memory uses its own little algorithm to determine the next cache line state.

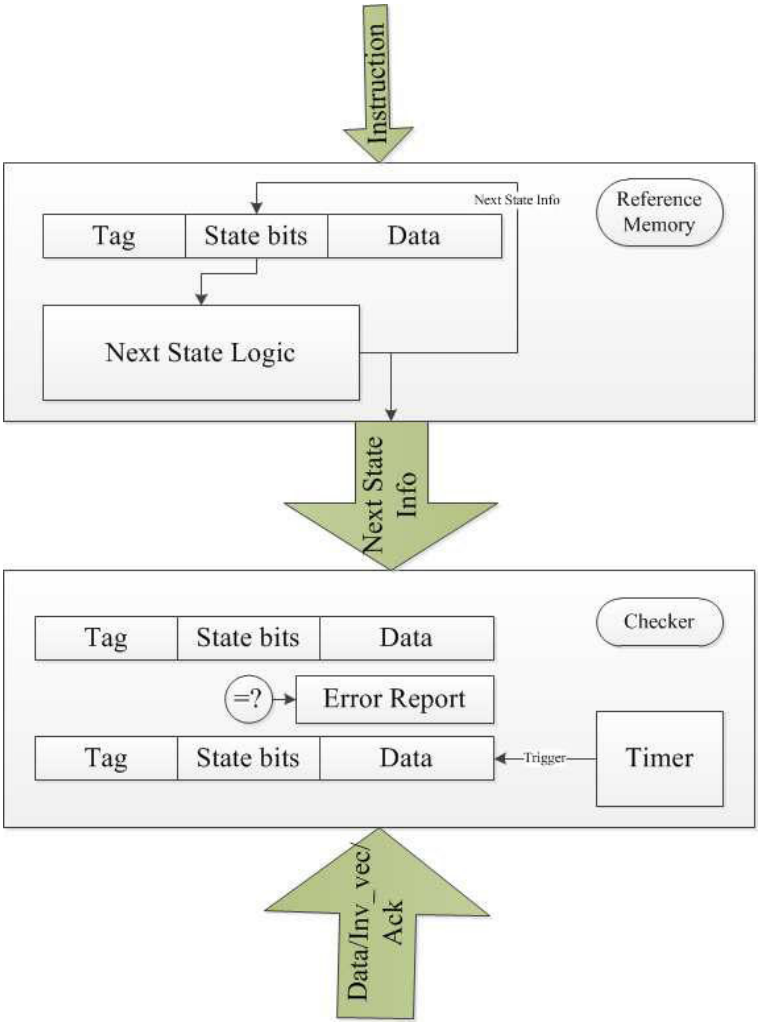

Figure 5-3 Simple checker structure

In addition to checking data for a match, simple checkers are also in charge of checking correctness of invalidation vectors. In order to do that, each checker has a valid bit response to evict requests and invalidation vectors. Tables 5-1 and 5-2 below list the expected updates to valid bits in the reference memory, depending on the operations initiated by L1 controllers or responses from the L2 memory arrays, respectively. Similarly, we need a watchdog timer to allow delayed checking.

Table 5-1 Simple Checker's Valid-bit algorithm for operations initiated by L1 controllers.

Initial State	SLD	SST	OLD	OST
I	V	V	I	I
V	V	V	V	I

Table 5-2 Simple Checker's Valid-bit algorithm for responses from L2 memory arrays.

Initial	Evict	Inv_vec
I	Error	I
V	I	I

Chapter 6

Conclusion

We established the key issues that should be considered during the course of verification of cache coherence for the OpenSPARC T2 system. The first issue we need to consider is how to present the whole system and all of its operational conditions in machine understandable terms. We discussed this in Chapter 3, coverage definition.

We came up with a procedure on how the verification environment should be built. We discussed this in Chapter 1 subsection 2, proposed general verification procedure.

We use a SPARC cache controller as an example to illustrate how the procedure is executed in detail. We did a detailed DUT study and gathered enough information to execute the verification plan.

A transition coverage model is proposed, implementation details are introduced in detail in Chapter 3, Coverage Definition.

One important supporting structure for the test environment is introduced in detail. We introduced the Test Generation for Transition Coverage (TGTC) algorithm to accelerate simulation in Chapter 4. Also, we introduced an FSM checker structure and simple checker structure. Pros and cons of them were discussed in Chapter 5.

References

[1] M. Yadav, ASIC Verification Course (ECE745). Lecture Notes available: http://asicverification.wordpress.com/verification/lectures/

[2] R. Rodrigues, I. Koren and S. Kundu. "A Mechanism to Verify Cache Coherence Transactions in Mulicore Systems," Proc. of the IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology, 2012, pp. 211-216.

[3] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," Proc. of the Design, Automation, and Test in Europe Conference and Exhibition (DATE), 2012, pp. 3-8.

[4] Sun Microsystems, Inc. "OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification Vol. 2, " [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html

[5] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Dynamic Verification of Cache Coherence Protocols," High Performance Memory Systems, H. Hadimioglu ed., Springer, 2004, pp. 25-42.

[6] Mentor Graphics Inc. "Verification Planning and Management," [Online]. Available: http://verificationacademy.com/courses/verification-planning-and-management