Assertion-Based Verification of Look Aside Interface

Asif Iqbal Ahmed

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical Engineering) at

Concordia University

Montréal, Québec, Canada

April 2005

Abstract

Assertion-Based Verification of Look Aside Interface

Asif Iqbal Ahmed

This research addresses assertion-based verification to verify Look Aside Interface (LA-1 Standard). The Look Aside interface is intended for devices located adjacent to a network processing device (NPE) that off load certain tasks from the network processor. This verification is performed in several steps. First, we added assertions within the control logic of the RTL block and then we added assertions between interfaces of RTL blocks by following the Look Aside interface design specification. We developed the simulation environment in Verilog and relevant testcases derived from the specification, in order to perform simulation and observe the assertion messages for possible firings. Each firing pinpoints an error in the RTL. For the assertion monitors the Accellera Open Verification Library (OVL) was used. Finally, we transformed the same assertions into Sugar 2.0 properties and performed the model checking of our Verilog RTL model using Rulebase, from IBM corp.

Acknowledgments

The contributions of others to the completion of this thesis have been substantial and varied. I will here acknowledge those whose name most immediately comes to mind and apologize those names I have omitted inadvertently.

I would like to express my sincere gratitude to my supervisor Dr. Otmane Ait-Mohamed. His guidance, assistance and encouragement were an asset during the thesis work in particular and in my stay at Concordia, in general. Professor Otmane Ait-Mohamed also read and criticized many versions of this thesis. The final version owes much to him, the mistakes are my own.

I would like to thank the examination committee members, especially to Dr. Sofiene Tahar for serving in my thesis defence. His comments and feedback were of great value in improving the research pertaining to this thesis.

To all my research associates in the Hardware Verification Group (HVG) at Concordia, thank you for your encouragement, thoughtful discussions, and productive feedback. I wish to express my sincere and heartfelt thanks to Ali Habibi, Yassine Mokhtari, Sadegh Jahanpour, Abu Nasser Mohammed Abdullah, Haja Moinudeen and Sayed Hafizur Rahman who have helped me during the course of my research.

Last but not least I would like to thank my wife, my parents and the rest of my family members in Bangladesh for their constant moral support and encouragement which were invaluable in completing this thesis.

TABLE *of* CONTENTS

LIST *of* FIGURES

LIST *of* TABLES

List of Acronyms

- SoC : System-on-chip
- OVL: Open Verification Library
- LA-1 : Look-Aside Interface
- PSL : Property Specification Language
- IBM: International Business Machines
- BDD: Binary Decision Diagram
- CTL: Computational Tree Logic
- VIS: Verification Interacting with Synthesis
- SMV: Symbolic Model Verifier
- ASM: Abstract State Machines
- RTL: Register Transfer Level
- IP: Intellectual Property
- FSM: Finite State Machine
- ABV: Assertion-Based Verification
- HDL: Hardware Description Language

Chapter 1: Introduction

1.1: Motivation and Goal

As the modern designs past one-million and move towards system-on-chip (SoC) of 10 million gates, the SoC development teams require state-of-the art tools and methodology to verify these devices. Most of the case studies have shown that functional verification consumes more than 50 percent of the design-cycle time (Fig. 1.1) [1].

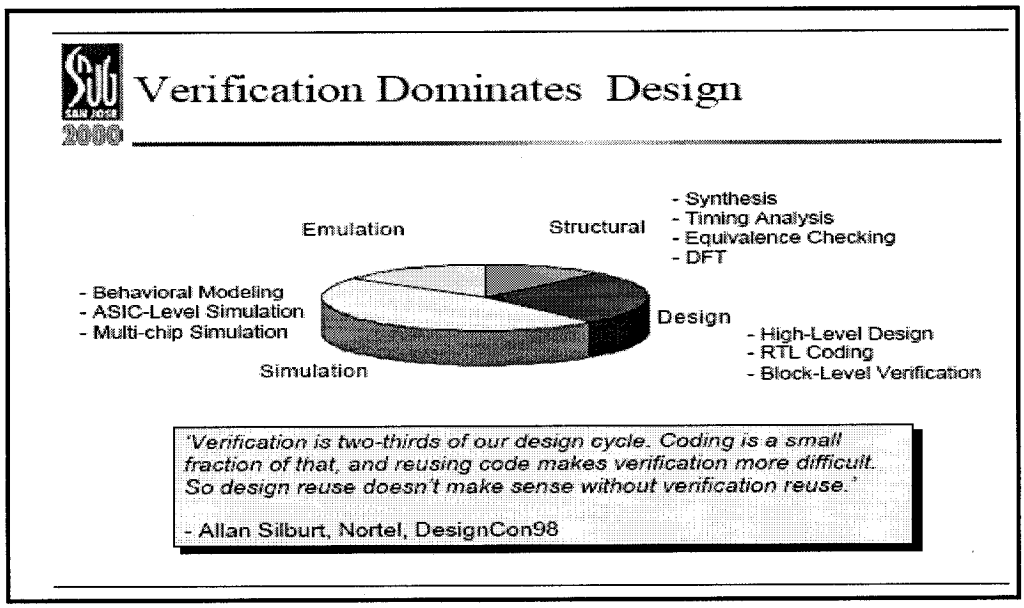

Figure 1.1: Verification Dominates Design

With functional verification involving such a dominant portion of the chip development process, SoC teams look for any opportunity for leverage. Some of the best leverage comes from the designers who wrote the RTL code that must be verified. Designers can no longer pass their code "over the wall" to the Verification team; they need to be involved in the verification process in order to ensure that the SoC works as intended. Therefore, the process of SoC verification is becoming a necessity for large, complex chip projects. This process encompasses a broad range of concepts, including

verification friendly RTL coding standards for the designers, cross-participation in design and Verification plan reviews, and clean, consistent interfaces at multiple levels of design abstraction. Perhaps the greatest leverage is obtained when the designers specify assertions i.e. properties of the design, in order to capture the specific design's intent, communicate the intent to team members, and aid the verification process. The primary usage of assertions addresses the RTL designer's assumptions (property specification of the design) and thus checks for errors in RTL code.

Therefore, property specification, in terms of assertions, is the key ingredient of this revolutionary SoC verification process, whose end result is an improved and re-usable verification process, through an **intelligent test bench,** as shown in Fig. 1.2 [2]. An assertion-based verification platform is an integral part of this intelligent testbench.

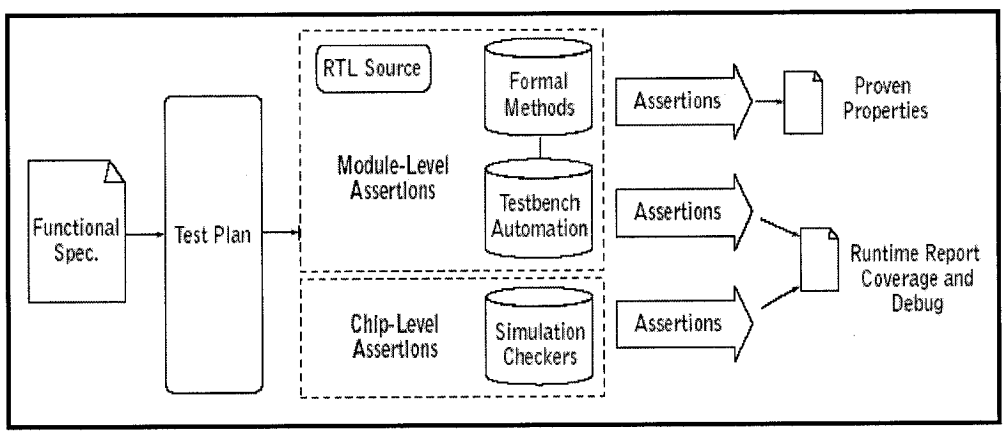

Figure 1.2: Intelligent Test Bench

The key components of an assertion-based verification platform are the following:

1. ***Verifiable testplans*** (Verification Plan) through safety property specification.

2. ***Reactive coverage driven testbenches*** based on property specification (assertions and functional coverage).

3. ***Exhaustive formal verification techniques*** (i.e. model checking, property checking).

Our research provides some essential background on assertions and their value as part of SoC verification, and discusses different methods for assertion and property specification. There are many different techniques, languages and libraries that design (and verification) engineers can use for assertions, each with its own merits. This work summarizes the key requirements for assertion specification and assesses how well the various techniques i.e. **model checking**, meet these requirements. Finally, the motivation behind our approach was to investigate the important role of assertion-based verification and model-checking techniques in the context of a SoC verification flow.

1.2: Background and Related Work

The design specification considered for our thesis is called Look-Aside Interface (LA-1 standard) [3], which is provided by the Network Processing Forum. Currently, the LA -1 interface is an open standard which has been developed and endorsed by multiple memory and network device vendors, and so is expected to achieve high adoption rates over a range of compatible network products. The main aspect of our work is to illustrate the application of Open Verification Library (OVL) [17] for assertion-based verification and the role of Sugar 2.0 for model checking using RuleBase [12].

As related work to ours, we cite the approach proposed by A.Habibi *et* al. [4]. In that work the authors proposed a technique (Fig. 1.3) to design and verify the Look-Aside Interface (LA-1). Their design flow includes several refinements starting from an informal UML specification until getting to an RTL modeled in Verilog. Afterwards, they integrated the verification of the LA Interface in the design flow by considering two intermediate levels: (1) Abstract State Machines (ASM) [20]; and (2) SystemC [21]

assertions. The first one serves the verification by model checking of a set of PSL properties, while the second includes a set of assertions to be verified by simulation.

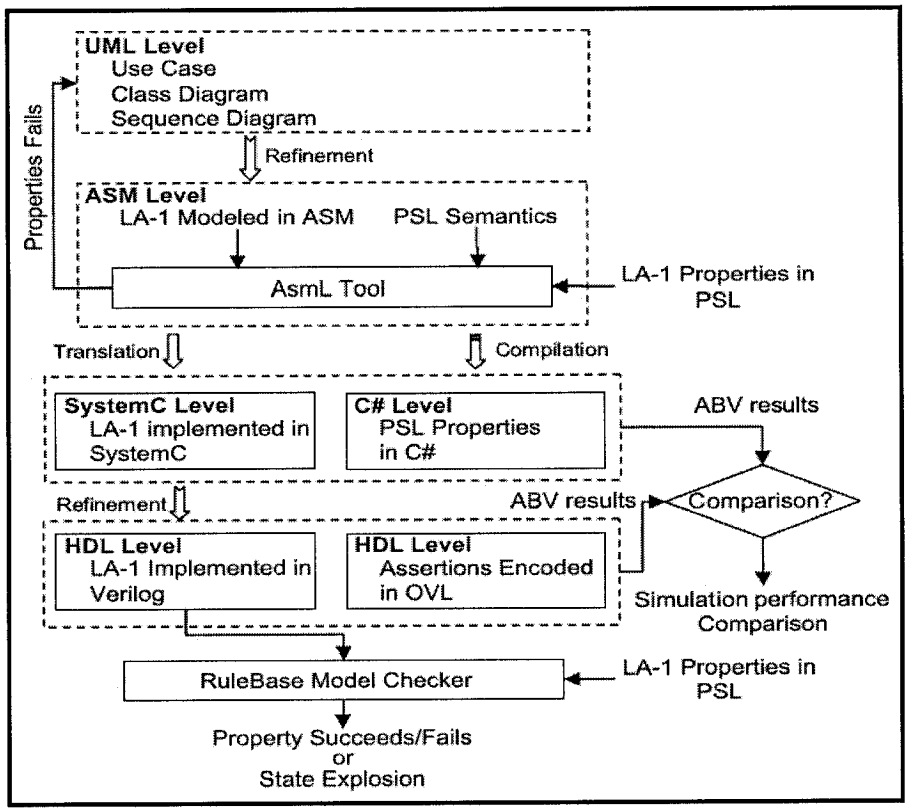

Figure 1.3: Hierarchical Design Verification Methodology for LA-1 Interface

Another related approach was proposed by Kanna *et* al. [5] to implement a PCI bus as a Verilog monitor and to verify its properties using SMV [6]. In [5], the bus was implemented in Verilog with all the properties (assertions) embedded as part of the RTL code. Our approach based on OVL enabled us to differentiate between the properties (assertions) and the RTL code, as the assertion monitors used in our RTL model were localized in a separate area of the code.

Pankaj *et* al. [7] also described a methodology for verifying system-on-chip designs. Their methodology was comprised of three major tasks. First, they verified, the standard bus interconnecting IP Cores in the system. The next task was to verify the glue

logic, which connects the IP Cores to the buses. Finally, using the verified bus protocols and the IP core designs, the complete IP-Core based system was verified (Fig 1.4).

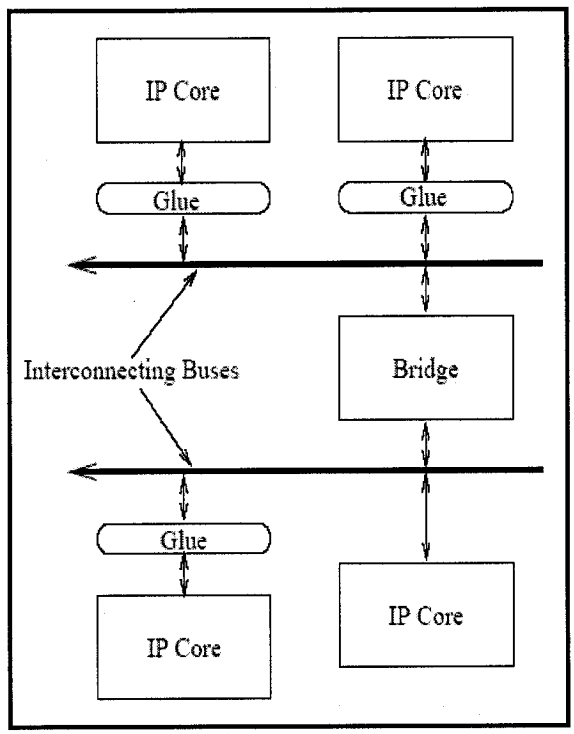

Figure 1.4: A typical IP-Core Based System

To illustrate their methodology, they verified the PCI Local Bus, a widely used bus protocol in system-on-chip designs. The methodology involved various modeling (not the actual RTL implementation) of the PCI Bus and verification technique. The strategy described by the authors lacked the practice of defining assertions which makes it impossible to re use previously well tested properties. That means the effort of writing a similar specification is duplicated from project to project, and from company to company. The consequences are huge when standard protocols, such as PCI and PCI-X, are to be verified. It is clear that a capability to develop libraries of assertions can provide time-savings for many designers. Some examples are:

- Standard protocols

- Core designs that are the basis for derivative systems
- Commonly used assertions for design elements

Intellectual property (IP) vendors that ship pre-built designs can supply built-in assertions, thereby alleviating the quality concerns of customers. An assertion language verification IP has well defined and tested ready-to-use assertions for designers. Therefore, the burning question of today's SoC Verification truly depends on the assertion writing and property defining skills of a RTL designer. With this view in mind the rest of the sections in this chapter elaborates the idea of assertion-based verification and afterwards introduces different techniques of formal verification.

1.3: Scope of the Thesis

In this thesis work, we illustrate assertion-based verification methodology for the formal verification of the Look-Aside design interface specification using OVL [17] and PSL/Sugar 2.0 [8]. The highlight of our work is to present the novel idea of Assertion-Based Verification in order to carry out a verification process of standard design interfaces currently used in the industry. The rest of this thesis is organized as follows: Chapter 2 provides a brief introduction to ***Accellera's Open Verification Library (OVL)*** and its related assertion monitors then it illustrates, ***Model Checking using PSL/Sugar*** and the general features of ***Rulebase***. Chapter 3 gives a brief overview of the Standard Bus Protocols. Chapter 4 explains the main features of Look-Aside interface (LA-1 Standard). Chapter 5 discusses the verification plan. Chapter 6 & 7 elaborates the verification results. Finally, chapter 8 concludes this report.

Chapter 2: STATE-OF-THE-ART

2.1: Simulation Based Verification

With the introduction of hardware description languages (Verilog or VHDL), it became common to describe both the Device Under Test (DUT) and the test environment in VHDL or Verilog. This process of Design Verification is known as Simulation-Based Verification. In a Simulation-Based Verification, the test environment has the following features:

- The testbench consisted of HDL procedures that provided stimulus data to the DUT or read data from it.
- The tests (testcases), which called the testbench procedures in sequence to apply selected input stimuli (random or directed) to the DUT and check the results, were directed towards specific features of the design.

All these features depict Black-Box Verification technique. Currently, the verification practice in the industry is driving more towards white-box verification technique. In this context the key role is played by Assertion-Based Verification. In the following sections enlighten the reader about Assertion-Based Verification and then introduces the usage of OVL Assertion Monitors.

2.1.1: Assertion-Based Verification

The main goal of our research is to apply assertion-based verification, in order to verify Look Aside Interface protocol. Assertion monitors are instances of modules whose purpose is to verify that certain conditions hold true in the RTL. An assertion monitor is composed of an event, a message, and a severity. Event is a property that is being verified by a monitor. An event can be classified as a safety (invariant) property. A safety

property is a property that must be valid at all times i.e. a testcase for a certain scenario. Message is the string that will be displayed in case of an assertion failure. Severity represents whether the error captured by the monitor is a major or a minor problem. Severe errors can be used to halt simulation. We briefly enumerate some of these benefits here [22].

- Halts simulation (if desired) on assertion errors to prevent wasted simulation cycles
- Simplifies debugging by localizing the problem to a specific area of code
- Increases test stimuli observability, which enhances pseudo-random or directed test vector generation strategies
- Provides a mechanism for grading test stimuli functional coverage (e.g., event monitoring coverage)
- Enables the use of formal and semi-formal verification techniques (e.g., provides verification targets and defines constraints for formal assertion checkers)
- Provides a means for capturing and validating design environment assumptions and constraints and relating it to the design specification.

In general, assertions document the RTL designer's assumptions on the design inputs and design's intended behavior on the design outputs. This means that there is still the fundamental need of a test bench, but the verification environment (test bench) no longer depends on the input stimulus, generated to propagate all effects resulting from the bug to an output port. Traditional approach to verify a design is to implement the design using HDL and simulate it using an environment, i.e. testbench. Simulation is a black-box method which treats design as a black box and mainly relies on input and expected output behavior. This approach is not suitable for designs of higher complexity. When an

expected output does not occur, the cause of error is difficult to trace. There might be some errors which are hard to detect due to incomplete specification especially regarding the corner case behavior of the system. Therefore, the verification process is improved with a better controllability and observability of the design during the procedure through white-box verification based on assertion monitors. In this verification approach we directly check the corner case behavior and detect the error at its source rather than at output. This white box approach uses the designer's knowledge of the system, i.e. assertions in order to specify the assertion which describes the legal and illegal behavior in the design as shown in Fig. 2.1 [1].

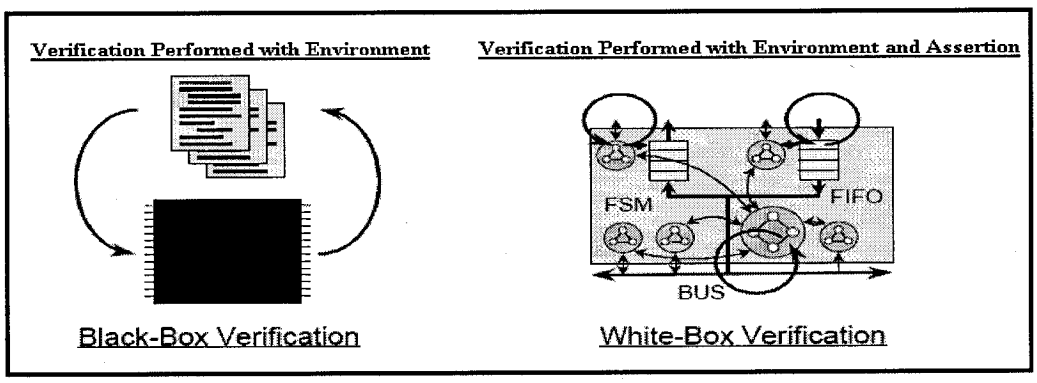

Figure 2.1: Black-Box vs. White-Box Verification

In shorts, assertion monitors document the RTL designer's assumptions on the design inputs and design's intended behavior on the design outputs. Suppose, a the Traffic Light Controller RTL block (written in verilog) needs an assertion monitor in order to perform as a watchdog so that both sides of the road crossing never go green at the same time. This monitor will use ***clk***, ***reset_n*** and the ***test_expr*** as inputs and will check that the ***test_expr*** is never equal to 1. The *clk* is the clock signal provided by the test bench and the reset_n signal is used as an enable signal which initiates the monitoring of assertions. The *test_expr* is the property or the assertion of the design

extracted by the designer. For example: The property, "**The traffic light on both sides of the crossing should never be green at the same time**" is implemented by using the *assert_never* monitor : **assert_never both_lights_are_green (clk, reset_n, (major_rd_light=='GREEN && minor_rd_light =='GREEN))**; The *test_expr* in this case is *(major_rd_light=='GREEN && minor_rd_light=='GREEN).*

Assertion monitors are written in either Verilog or VHDL or Standard Property Specific Language (PSL) [6]. In order to accommodate the need for assertion monitor there is some amount of additional glue logic added to the RTL code. Synopsys off/on comment based pragmas (synopsys translate_off / synopsys translate_on) should be used to ensure that the glue logic is not synthesized. For our verification, we selected assert_always, assert_never and assert_next which are used as most general assertion monitors in the industry. These assertion monitors are employed whenever the user wants to verify an invariant property [4]. The semantic definition for ***assert_always*** is similar to the definition for ***assert_never***. However, the assert_never monitor checks that test_expr is never equal to 1 while the assert_always monitor checks that the test_expr is always equal to 1 [2].

2.1.2: OVL Monitor Methodology for Design Verification

The ***Accellera Open Verification Library (OVL)*** [17] provides designers, integrators and verification engineers with a single, vendor-independent interface for design verification using simulation, semi-formal and formal verification techniques. By providing a single well-defined interface, the Open Verification Library (OVL) can bridge the gap between the different verification techniques, enabling a seamless flow between simulation and formal tools. For the first time, user-defined properties of a

design can be specified in a straight forward, standard way, so that they can be written once and used by multiple verification tools in the design flow. The OVL is composed of a set of assertion monitor modules that verify specific properties of an HDL design. These modules are bundled together in a library, enabling designers to instantiate these monitors during simulation. The initial version of the OVL was written using standard Verilog HDL syntax. Currently, a VHDL version of the Library is also available. Both versions of the OVL may be freely downloaded and modified or extended by the user community (http://www.accellera.org). All approved extensions will become part of the standard release.

The main advantages of using OVL monitors are:

- OVL monitors are assertion checkers, which can test internal points of the design, thus increasing observability of the design (white-box approach).
- OVL monitors simplify the detection and diagnosis of bugs by constraining the occurrence of a bug to the monitor being checked.
- OVL monitors allow designers to use the same design property specification for both simulation and formal verification (in form of properties or events).

It is difficult to setup all the specified operations based on input and output alone. The verification process is improved with a better controllability and observability of the design during the procedure through white-box verification. In this approach we directly check the corner case behavior and detect the error at its source rather than at output. The white box approach uses the designer's knowledge of the system to specify the assertion which describes the legal and illegal behavior in the design. Formal verification is one such traditional white-box approach. In formal verification we describe design using

mathematical techniques to specify and prove the specifications of the design. Detailed knowledge of the behavior of the design is required for the valid assertions that should to be proved. The formal verification methodology explores all possible behavior of the design to find the circumstances under which the assertion is violated. The difficulty with this approach is to write the set of assertions that completely represents the behavior of the design. Therefore it is used as an added necessity and not replacement to the simulation procedure.

By instantiating a module for the assertion checker, the verification process is able to isolate assertion implementation details from the function of the design. We can create assertion libraries optimized for specific verification processes within the flow. Static assertions are used for specific event or undesirable condition, i.e., they try to capture an undesirable event. The temporal assertions can be viewed as an event-triggered window, bounding the assertion. Assertion check for invariant property (condition or expression should be true between the occurrence of two events) and assertion check for liveliness property (condition or expression should be true at least once between occurrence of two events) come under temporal assertions. Assertions also check the safety properties.

2.1.3: OVL Assertion Monitors usage and definition

The following are the brief descriptions of the assertion monitors that are currently available in the assertion monitor library and are being used for the purpose of our research. They can be used by including the library in the design and instantiating them. Each assertion has the parameters *severity*, *options* and *msg*. *severity* describes the level of severity of the error based on which the tool decided whether to end the

simulation. *options* are used to describe the characteristics of the assertion monitor to the EDA tools. Currently the only *options=0* is supported which describes the assertion monitor as a constraint to the formal verification tools. *msg* is used to describe the message which should be displayed when the assertion fires.

➢ assert_always

Syntax: [17]

assert_always [#(severity_level, options, msg)] **inst_name** (clk, reset_n, test_expr);

Where

- severity_level - severity of the failure - default 0
- Options - Vendor options - default 0
- msg - Error message that should be given out when assertion fires
- inst_name - Name of the instance
- clk - Triggering or clocking event that monitors the assertion
- reset_n - reset signal, unless global *RESET* signal is set
- test_expr - Expression being verified at the specified edge of *clk*

Overview: ***assert_always*** assertion continuously monitors the ***test_expr*** at every positive edge of the triggering event or clock *clk*. This assertion will fire whenever ***test_expr*** evaluates to FALSE.

Usage: ***assert_always*** can be used in verification of a property that should always hold TRUE at clock boundaries or at specified edge of clock. Fig. 2.1 shows the Control Flow graph of ***assert_always*** monitor.

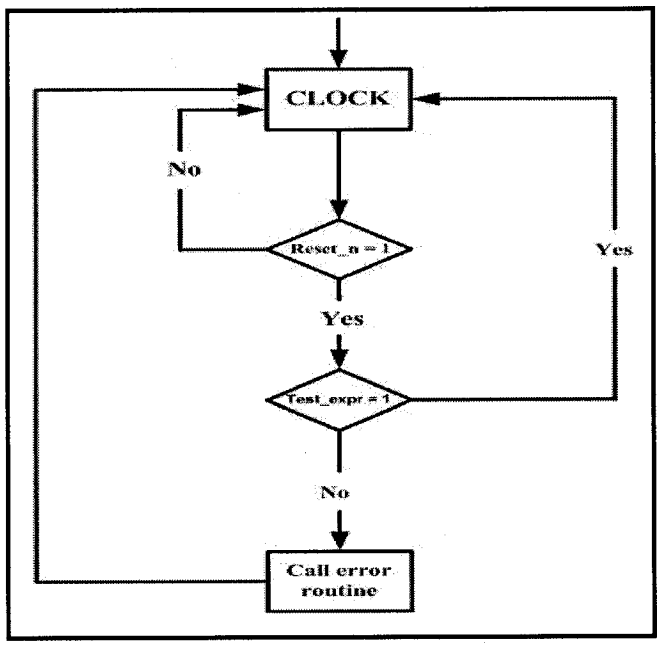

Figure 2.2: Control Flow Chart of *assert_always* monitor

The associated ***FSM chart*** for the assert_always monitor is shown in Fig. 2.2. In the FSM chart S0 is the initial state where the test_expr is being checked if the reset_n is active high. If the test_expr is not true then FSM transits to S1 and the OVL assert_always monitor flags an error which is captured by the verilog simulator.

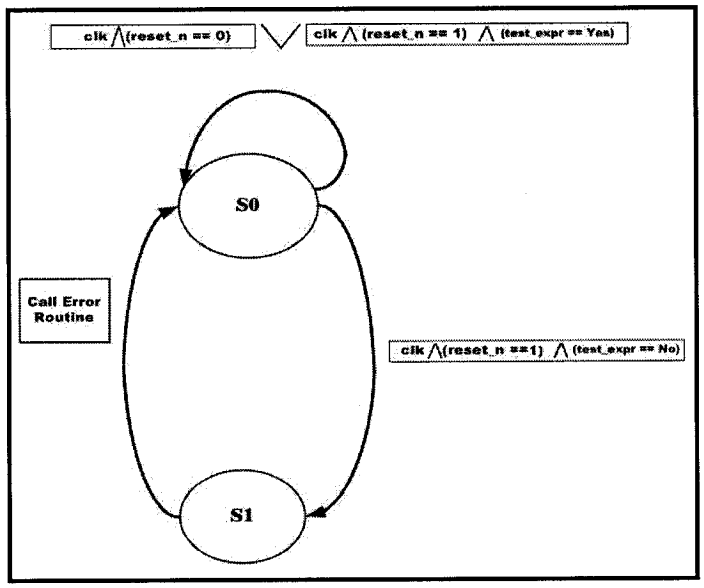

Figure 2.3: FSM Chart of assert_always monitor

The OVL monitor assert_always is written in verilog and has a synchronous process which is described below [17]. Note, that the process governs the error checking process which is described by the Control Flow Graph and the respective FSM.

```
always @(posedge clk) begin
  `ifdef ASSERT_GLOBAL_RESET   // defined if needed
       if (`ASSERT_GLOBAL_RESET != 1'b0) begin
       `else
       if (reset_n != 0) begin  // active low reset local to the monitor itself
       `endif
       if (test_expr  != 1'b1) begin  // test_expr under investigation
       ovl_error("");                 //ovl error call
     end
   end
 end
```

➢ assert_never

Syntax: [17]

assert_never [#(severity_level, options, msg)] **inst_name**(clk, reset_n, test_expr);

Where

- severity_level - Severity of the failure - default 0
- options - Vendor Options - default 0
- msg - Error message that should be given out when assertion fires
- clk - Triggering or clocking event that monitors the assertion
- reset_n - reset signal, unless global *RESET* signal is set
- test_expr - The expression that is monitored at every specified clock pulse

Overview: assert_never assertion monitor continuously checks the *test_expr* at every

positive edge of the sampling event and triggering event or clock *clk*. The assertion monitor checks that *test_expr* is always false. The assertion fires when the *test_expr* evaluates to true.

Usage: assert_never assertion monitor should be used to verify a property that should never hold TRUE at clock boundaries. Fig. 2.4 shows the Control Flow graph of ***assert_never*** monitor.

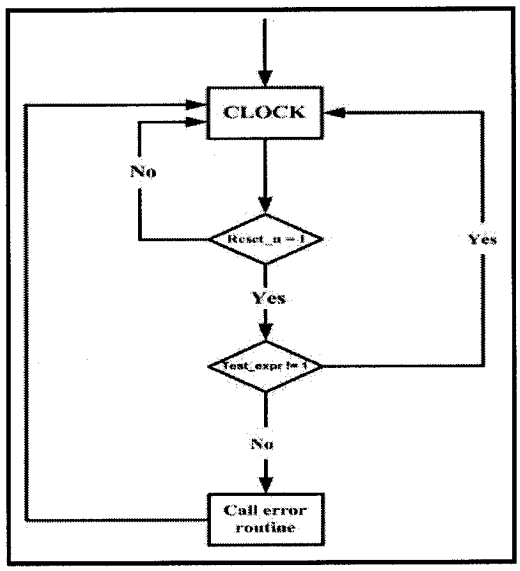

Figure 2.4: Control Flow Chart of assert_never monitor

The associated ***FSM chart*** is shown in Fig. 2.5. In the FSM chart S0 is the initial state where the test_expr is being checked if the reset_n is active high. If the test_expr is true then FSM transits to S1 and the OVL assert_always monitor flags an error which is captured by the standard verilog simulator.

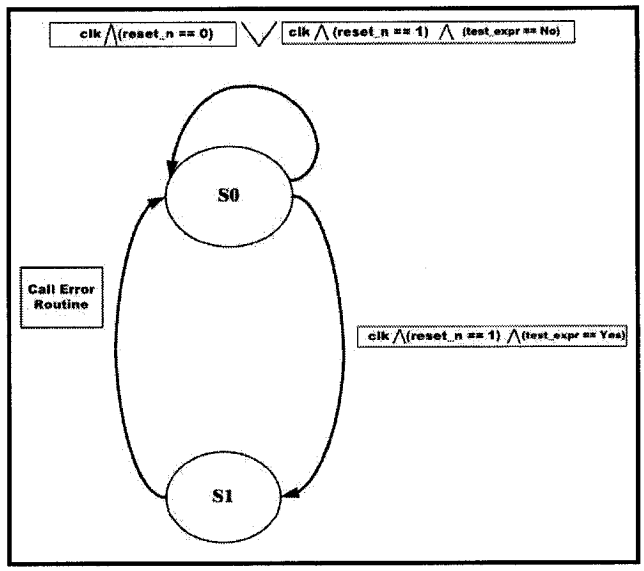

Figure 2.5: FSM Chart of assert_never monitor

The source code of ***assert_never*** monitor depicts the error checking scenario.

```
always @(posedge clk) begin
  `ifdef ASSERT_GLOBAL_RESET
        if (`ASSERT_GLOBAL_RESET != 1'b0) begin
  `else
       if (reset_n != 1'b0) begin // active low reset
  `endif
                if (test_expr  != 1'b0) begin       // checking of test_expr
                       ovl_error("");               // ovl error function call
                end    // test_expr
    end
 end // always
```

- **assert_next**

Syntax [17]:

assert_next [#(severity_level, num_cks, check_overlapping, only_if, options, msg)]

inst_name (clk, reset_n, start_event, test_expr);

Where

- severity_level - Severity of the failure - default 0
- num_cks - The number of clock cycles after *start_event* when *test_expr* should evaluate to true
- check_overlapping - When true overlapping of sequences is allowed, i.e. another *start_event* is accepted before *test_expr* occurs
- only_if - When true, *test_expr* can evaluate to true only when *start_event* is true *num_cks* before it occurs
- options - Vendor Options - default 0
- msg - Error message that should be given out when assertion fires
- clk - Triggering or clocking event that monitors the assertion
- reset_n - reset signal, unless global *RESET* signal is set
- start_event - The event that starts the monitoring of *test_expr*
- test_expr - The expression that is monitored at every specified clock pulse

Overview:

assert_next assertion continuously monitors the relationship between two events. It checks that whenever *start_event* evaluates to true, *test_expr* evaluates to true exactly after *num_cks* clock cycles.

Usage:

The **assert_next** assertion should be used to ensure proper sequence of events. Fig. 2.6 shows the Control Flow graph of ***assert_next*** monitor.

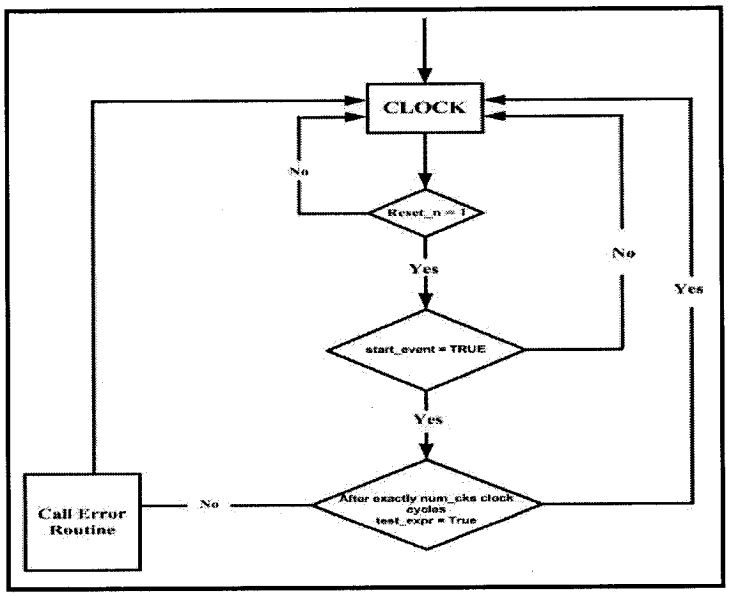

Figure 2.6: Control Flow Chart of assert_next monitor

The associated ***FSM chart*** is shown in Fig. 2.7. In the FSM chart S0 is the initial state where the start_event is being checked if the reset_n is active high. If the start_event is true then FSM transits from S0 to S1, where the test_expr is being validated, after the exact number of clock cycles. If the test_expr is false then S1 transits to S2 and an error flag is being raised.

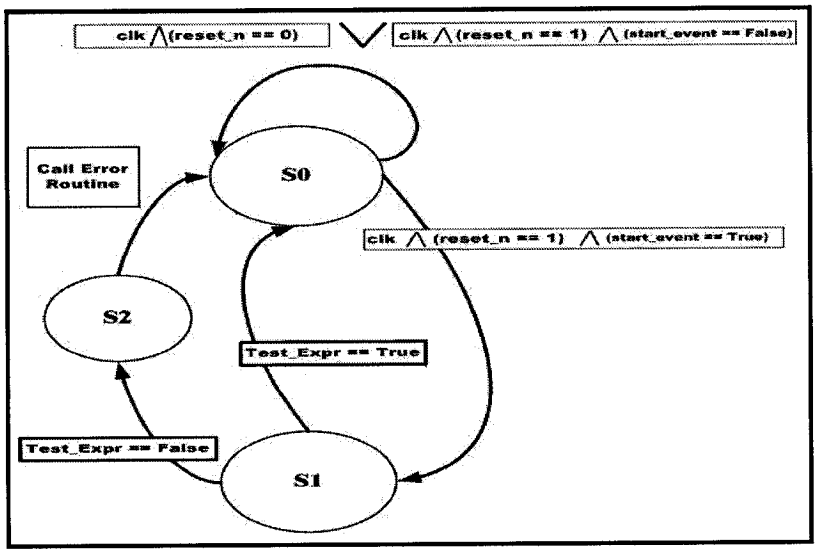

Figure 2.7: FSM Chart of assert_next monitor

The verilog source code of assert_next monitor illustrates the above validation scenario:

```
always @(posedge clk) begin
`ifdef ASSERT_GLOBAL_RESET
        if (`ASSERT_GLOBAL_RESET != 1'b0) begin
  `else
        if (reset_n != 0) begin  // active low reset
  `endif
        monitor <= (monitor_1 | start_event);
        if ((check_overlapping == 0) && (monitor_1 != 0) && start_event) begin
                ovl_error("illegal overlapping condition detected");
        end
       else if (only_if == 1) begin
        if (!monitor[num_cks-1] && test_expr) begin
                ovl_error("test_expr without start_event");
        end
       end
       else begin
                if (monitor[num_cks-1] && !test_expr) begin
                        ovl_error("start_event without test_expr");
                end
       end
       end   else begin
                monitor <= 0;
     end
 end // always
```

2.1.4: OVL Based Verification Techniques

Assertion monitors address design verification concerns. In general there are certain guidelines to follow in order to make decisions about placement for the OVL

assertion monitors in the RTL code of the Designers. The guidelines are listed as follows:

- Combine assertion monitors to increase the coverage of the design (for example, in interface circuits and corner cases).
- Include assertion monitors when a module has an external interface. In this case, assumptions on the correct input and output behavior should be guarded and verified.
- Include assertion monitors when interfacing with third party modules, since the designer may not be familiar with the module description (as in the case of IP cores), or may not completely understand the module. In these cases, guarding the module with assertion monitors may prevent incorrect use of the module.

Usually there is a specific assertion monitor suited to cover a potential problem. In other cases, even though a specific assertion monitor may not exist, a combination of two or three assertion monitors can provide the desired coverage. The number of actual assertions that must be added to a specific design may vary from a few to hundreds, depending on the complexity of the design and the complexity of the properties that must be checked. Writing assertion monitors for a given design requires careful analysis and planning for maximum efficiency. While writing too few assertions may not increase the coverage on a design, writing too many assertions may increase verification time, sometimes without increasing the coverage. In most cases, however, the runtime penalty incurred by adding assertion monitors is relatively small. The following example depicts the usage of assert_always monitor in a counter RTL code.

```
module counter_0_to_9(reset_n,clk);
input reset_n, clk;
reg [3:0] count;
```

```
always @(posedge clk)  begin
        if (reset_n == 0) count = 1'b0;   // Should be if (reset_n == 0 || count >= 9)
        else count = count + 1;
end // always
assert_always #(0, 0, "error: count not within 0 and 9") valid_count (clk, reset_n, (count
>= 4'b0000) && (count <= 4'b1001)); // Instantiation of assert_always monitor
endmodule
```

Now, in order for the assertion monitor to fire the test_expr has to be false which means the value of count can never be less than numeric 0 and greater than numeric 9. In order to stimulate the above RTL we created the following testbench:

```
`define ASSERT_ON                1'b1
`define ASSERT_OVL_VERILOG       1'b1
`define ASSERT_INIT_MSG          1'b1
`define dumpon 1
`timescale 1ns/1ns               // Setting up the timescale
module counter_tb ( );
reg         reset_n;
reg         clk;
initial
begin
        clk = 0;
        reset_n = 0;
        #50 reset_n = 1;
        `ifdef dumpon
                $dumpfile("dump.vcd");
                $dumpvars;
        `endif
        #1000000 $finish;
end
always          #10 clk = ~clk; // Clock being generated
```

counter_0_to_9 cnt(reset_n, clk); // ***Instantiate the counter module***

endmodule

After simulating the file using ***vcs 7.0.1*** we receive the following ***VCS SIMULATION REPORT*** where the ***assert_always*** monitor initiated an ***OVL_ERROR***:

Parsing design file 'counter_new_tb.v'

Parsing design file 'counter_new.v'

Parsing design file 'assert_always.vlib'

Parsing included file 'ovl_task.h'.

Back to file 'assert_always.vlib'.

Top Level Modules:

counter_tb

TimeScale is 1 ns / 1 ns

1 of 3 unique modules to generate

Chronologic VCS simulator copyright 1991-2003, Compiler version 7.0.1;

OVL_NOTE: ASSERT_ALWAYS initialized @ counter_tb.cnt.valid_count.

ovl_init_msg Severity: 0, Message: error: count not within 0 and 9

OVL_ERROR: ASSERT_ALWAYS: error: count not within 0 and 9:: severity 0 : time 250 : counter_tb.cnt.valid_count.ovl_error $finish at simulation time 1000050

VCS Simulation Report Time: 1000050 ns

Afterwards, to analyze the error report we open the waveform viewer in order to monitor the error (Fig.2.8). The error count changed its value from 0 to 1 at 250 ns, which is the snapshot of the time where the test_expr went low and the assertion failed.

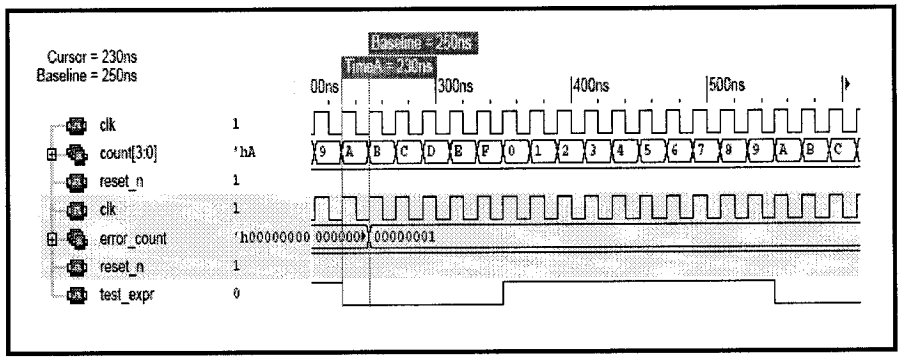

Figure 2.8: Error Investigation Based on OVL Error

Though OVL error was able to pin-point the error, the stimulus generated from the environment was directed in order to pinpoint the error in the RTL model. In case of an existence of an unknown bug, the directed stimulus generation won't be able to find the design error. In order to confront this particular issue, the Model Checking Technology using PSL/Sugar comes fully into play. Model checking scheme generates all possible stimulus patterns in order to verify all legal and illegal sequences of the inputs and therefore, increases performance of an intelligent testbench. In the following sections (2.3), of this chapter we will discuss Model Checking techniques using Property Specification Language (PSL) which is also known as Sugar 2.0.

2.2: Formal-Verification

Formal verification is the process of checking whether a design satisfies it specifications (properties). We are concerned with the formal verification of designs that may be specified hierarchically; this is also consistent with how a human designer operates. Formal Verification is a powerful technology which is gaining acceptance and is likely to serve as a fundamental verification methodology for future design-verification flow. Fig. 2.9 displays a typical hierarchical design and verification flow currently used in the industry [9]. The starting step is the concept, which is actually, the design specification of a given SoC design. Formal Verification Techniques have three main categories, namely, ***Theorem Proving, Equivalence Checking*** and ***Model Checking***. Later sections of this chapter will provide brief overviews of all the techniques

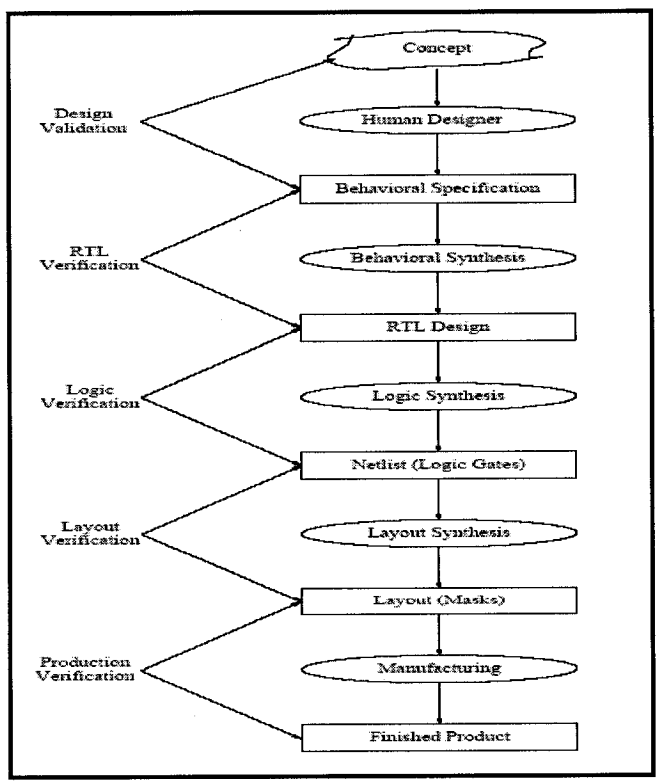

Figure 2.9: Flow of hierarchical design and verification

2.2.1: Theorem Proving

Theorem proving is the most advanced formal verification technique, in which a specification and its implementation are usually, expressed as first-order or higher-order logic formula. The mutual relationship between the specification and its implementation, equivalence or implication, is regarded as a theorem to be proven within the proof system using a set of axioms and inference rules. It is far more involved than both equivalence checking and model checking and requires changes in the basic design flow. Theorem proving requires that the design be represented using a "formal" specification language. Present-day hardware description languages such as VHDL and Verilog are unsuitable for this purpose, as they have no formal semantics. At each level of abstraction in the design flow, a theorem prover is used to ensure that the design corresponds to the original design specification. Some of the widely used theorem prover in the hardware verification community are HOL (Higher-Order Logic) [13], PVS (Prototype Verification System) [14], Nqthm (a Boyer7 Moore theorem prover) [15], ACL2 (Industrial strength version of the Boyer-Moore theorem prover) [16].

2.2.2: Equivalence Checking

Equivalence checking is the most widely used of the three formal verification techniques. This method uses a mathematical approach to verify equivalence of a reference and a revised design. In this form of verification, the tool performs an exhaustive check on the two designs to ensure that the designs behave identically under all possible conditions (Fig. 2.10) [10].

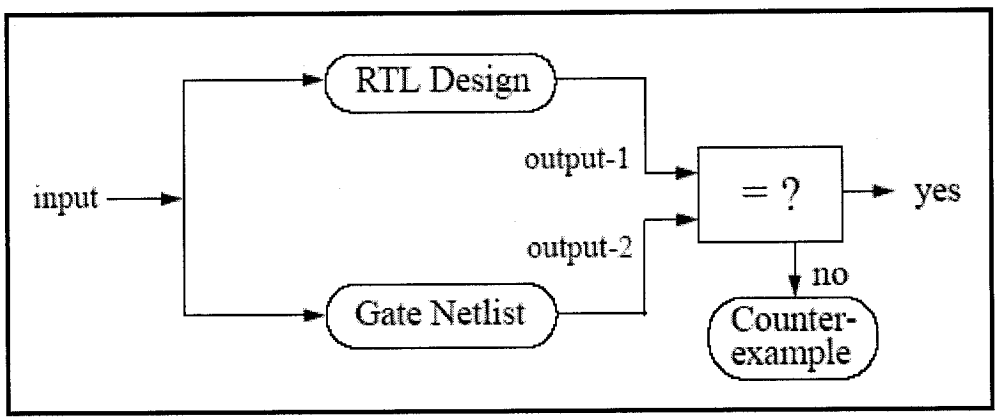

Figure 2.10: Equivalence Checking in Formal Verification

One of the limitations of Equivalence Checking is that it assumes that the reference design is correct. If there are any errors in the reference design, they will not be detected in the revised design. Therefore, it is crucial that the reference design is functionally correct. In modern days, Equivalence Checking is used in conjunction with Static Timing Analysis. Formality from Synopsys, Conformal LEC from Cadence are the two notable tools widely used in the industry for Formal Verification i.e. Equivalence Checking.

2.2.3: Model Checking

Model Checkers compare a design to an existing set of logical properties of a design's behavior as shown in Fig. 2.11[10]. These properties are a direct representation of the specifications of the design. The use of model checkers is more involved than equivalence checking. This is because the properties are needed to be generated by the user (RTL designer). Properties about the system are expressed as formulas in temporal logic of which the state transition system is to be a ``a model". The main advantage of model checking over simulation is that it frees the designer from the need to generate test vectors.

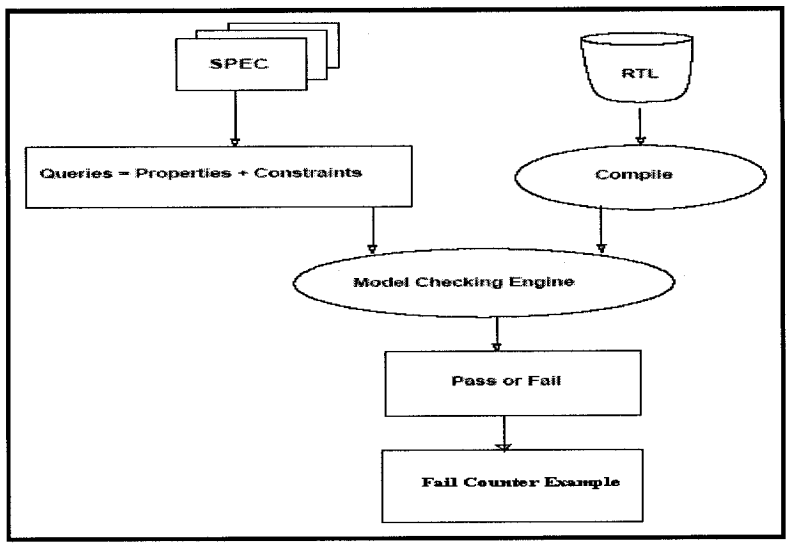

Figure 2.11: Basic Model Checking Idea

Model checking checks the properties specified for every possible input sequence. However, most chips are not designed to accept every possible input sequence, so if a given property fails for an illegal input sequence, it is of no interest. Thus, we need a way in which to specify all the legal input sequences to the formal verification tool. We can do this by specifying a model of the expected environment. This model describes the legal input sequences to the design under test.

One of the practical problems of model checking is known as "the size problem". Because of the size problem, complete model checking runs can verify designs that have a few hundred state variables (latches or flip-flops). This is not enough to be useful in real hardware designs. A number of tools have been developed, to perform model checking, the three well-known among them are SMV (Symbolic Model Verifier) [6], VIS (Verification Interacting with Synthesis) [11] and RuleBase Model Checker [12].

2.2.4: Temporal Logic Formulas

One of the practical challenges in order to adopt Model Checking in the verification process is writing properties in terms of Temporal logic. Temporal logic

formulas can be difficult to interpret, so that a designer may fail to understand what property has been actually verified. Therefore it is important to be familiar with the most common constructs of CTL used in practical hardware verification [11].

- ***AG (req ⟶ AF ack)***

  For all reachable states *(AG)*, if ***req*** is asserted in the state, then always at some later point *(AF)* we must reach a state where ***ack*** is asserted. ***AF*** is interpreted relative to the initial states of the system. ***AF*** is interpreted relative to the state where ***req*** is asserted. In other words, it is always the case that if the signal ***req*** is high, then eventually ***ack*** will also be high.

- ***AG AF enabled***

  From every reachable state, for all paths starting at that state we must reach another state where ***enabled*** is asserted. In other words, ***enabled*** must be asserted infinitely often.

- ***AG EF restart***

  From any reachable state, there must exist a path starting at that state that reaches a state where ***restart*** is asserted. In other words, it must always be possible to reach the restart state.

- ***EF (started ∧ ¬ ready)***

  It is possible to get to a state where ***started holds***, but ***ready*** does not hold.

- **AG (send ⟶ A (send U receive))**

  It is always the case that if ***send*** occurs, then eventually ***receive*** is true, and until that time, ***send*** must continue to be true.

- **AG (inp ⟶ AX AX out)**

  Whenever ***inp*** goes high, ***out*** will go high within two clock cycles.

In short, once RTL implementation begins, formal technology, especially Model Checking can be used to explore the design space around assertions within the implementation. In short, process increases confidence in the final design's ability to function correctly when built.

2.2.5: PSL/Sugar Formal Specification Language

Ensuring that a design's implementation satisfies its specification is the bottom-line of Model Checking technique. Key to the design and verification process is the vital role of specification. The pivotal act of specification consists of creating a natural language description of a set of design requirements. This form of specification is both ambiguous and, in many cases, unverifiable due to the lack of a standard machine-executable representation. Furthermore, in order to ensure that all functional aspects of the specification have been adequately verified (that is, covered) is problematic. The Accellera Property Specification Language (PSL) was developed to address these critical shortcomings. It was first introduced by IBM and had an in house name as Sugar. Later, Sugar was selected by the Formal Verification Technical Committee of the Accellera

EDA Standard organization, as the basis for an IEEE standard property specific language and adopted the name of PSL (Property Specific Language).

PSL 1.1/Sugar 2.0 gives the system/design architect standard means of specifying design properties (used for Model Checking) using a concise syntax with clearly-defined formal/regular semantics. Similarly, it enables the RTL design engineer to capture design intent in a verifiable form, while enabling the RTL verification engineer to validate that the implementation satisfies its specification through dynamic (that is, simulation/functional verification) and static (that is, formal) verification means. Furthermore, it provides a means to measure the quality of the verification process through the creation of functional coverage models built on formally specified properties. Plus, it provides a standard means for hardware designers and verification engineers to rigorously document the design specification (machine-executable). Therefore, due to the above stated advantages the specification language of Rule Base is chosen to be PSL 1.1/Suagr 2.0. It is used in order to formally describe properties to which the design under verification must adhere. Sugar is an extension of the temporal logic LTL (Linear Temporal Logic) which evolved from Computational Tree Logic (CTL). Particularly, complex LTL specifications are difficult to read and write. Sugar semantics adds, on top of CTL/LTL, a set of new operators that simplify formulation of complex properties and add significant expressive power. These operators are called "syntactic sugar". Figure 3.1 shows the evolvement abilities of Sugar including Syntactic Sugar and the Sugar Extended Regular Expression.

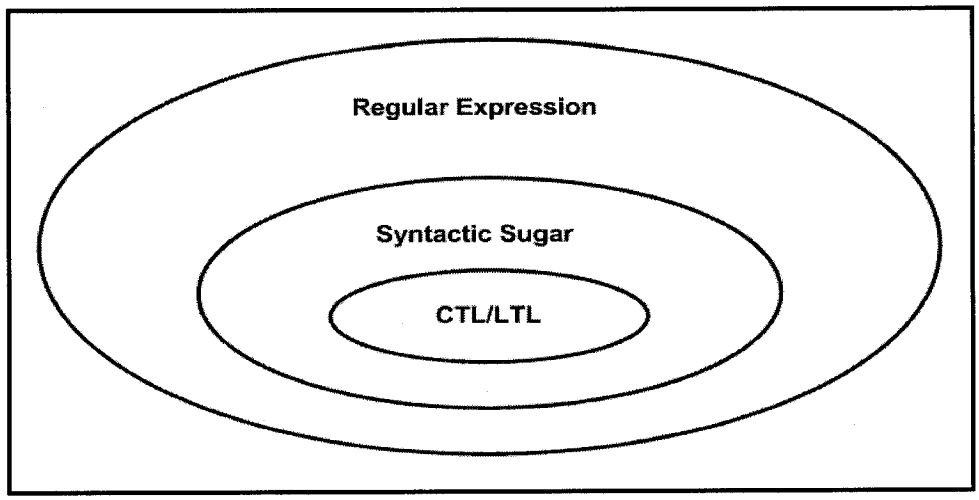

Figure 2.12: The evolving abilities of PSL/Sugar

As a result of this evolvement, properties written in Sugar can be divided into two formalisms. These formalisms are discussed in the following:

- The branching formalism, based mainly on CTL. Properties or Formulas of the branching formalism reason about computation trees. Such structures are generated by unfolding all the possible futures from a current state into an infinite tree.
- The linear formalism is based on Sugar Extended Regular Expressions (SEREs), SEREs describe sequences of events. For example, the sequence where signal ready is asserted 2 clock cycles after signal busy is described as: {[*], busy, true, ready}

Linear formulas link these sequences together using schemes and operators. For example, {Sere1} |=> {Sere2} means that the occurrence of Sere1 implicates that Sere2 should also occur. Besides, these regular expressions, other linear schemes such as "always" and "never" are natural for a user accustomed to logic design.

Sugar 2.0 can express properties that cannot be easily evaluated in simulation, although such properties can be addressed by formal verification methods [8]. In particular, it can express properties that involve branching or parallel behavior, which

tend to be more difficult to evaluate in simulation, where time advances monotonically along a single path. The simple subset of Sugar is a subset that conforms to the notion of monotonic advancement of time, left to right through the property, which in turn ensures that properties within the subset can be simulated easily. The simple subset of Sugar contains any PSL/Sugar FL property meeting all of the following conditions:

- The operand of a negation operator is a Boolean.
- The operand of a never operator is a Boolean or a Sequence.
- The operand of an eventually! operator is a Boolean or a Sequence.
- The left-hand side operand of a logical and operator is a Boolean.
- The left-hand side operand of a logical or operator is a Boolean.
- The left-hand side operand of a logical implication (->) operator is a Boolean.
- Both operands of a logical iff (<->) operator are Boolean.
- The right-hand side operand of a non-overlapping until* operator is a Boolean.
- Both operands of an overlapping until* operator are Boolean.
- Both operands of a before* operator are Boolean.

All other operators not mentioned above are supported in the simple subset without restriction. In particular, all of the next_event operators and all forms of suffix implication are supported in the simple subset. The current version of Sugar 2.0 has additional features, such as the ability to intermix it more freely with EDL (Environment Description Language) and add modeling in HDL Languages (VHDL, Verilog). The common goal for the hardware design industry is that PSL/Sugar will be used as a standard for many stages of the future VLSI design. The stages include design specification, functional verification (simulation), assertion-based verification and most

notably formal verification. In the following sections, we address the usage of Sugar 2.0 with respect to RuleBase Model Checking Tool. Note, RuleBase was the pioneer tool which used Sugar 2.0 in order to define the properties, which were used to perform model checking.

2.2.6: Rulebase Model Checker – Main Features and Counter Example Generation

One of the practical problems of model checking is known as "the size problem". Because of the size problem, complete model checking runs can verify designs that have a few hundred state variables (latches or flip-flops). This is not enough to be useful in real hardware designs. The RuleBase formal verification tool solves the size problem by renouncing the proof of truth that is possible with model checking on small designs. By renouncing the proof of truth, RuleBase can verify designs that contain up to a few thousand state variables. Although an answer of "true" to a specification is no longer a firm indication that the design is correct, an answer of "false" with a counter-example is an indication of a bug in the design (or specification or environment). This way, RuleBase can be used to obtain much better verification than is possible using simulation alone, even for designs that are too large for complete model checking.

One of the ways of dealing with the size problem is to reduce the design under verification. Reduction is accomplished by analyzing the environment description provided by the user as well as the specification to be checked, and eliminating any logic that has no bearing on the specification under the environment. Using the techniques of reduction in combination with renouncement of the proof of truth is known as over-reduction. For instance, instead of describing the complete environment of the design under test, the user may choose to describe a subset of that environment. RuleBase uses

the environment to reduce the design to a size that is suitable for model checking. Then, another subset of possible behaviors can be described. Thus, the user has complete control over the reduction process. An answer of "true" for a specification under a specific environment indicates that in this specific environment, the specification is true, but it does not indicate anything about the truth or falsity of the specification under other environments. Following is a list of RuleBase main features, which will be further discussed and exercised with the case study done in context of our thesis.

- **Sugar 2.0 specification Language**

  As described earlier, Sugar 2.0 is used to specify the properties of the given design.

- **Environment Description Language (EDL)**

  RuleBase checks the properties specified for every possible input sequence. However, most chips are not designed to accept every possible input sequence. Designers often assume a correct behavior of the target environment and simplify the design by ignoring illegal behaviors. RuleBase must be made aware of the environment's legal behavior, otherwise it might produce "false negatives", which are counter-examples that result from illegal input sequences. The way to specify environment behavior is to write environment models, which are the logic that drives the inputs of the design to be verified. Every input of the design must be assigned some behavior. Some inputs are kept constant (e.g., configuration inputs), others remain completely free (nondeterministic), while the control signals of interest are usually assigned detailed and exact behavior. Environment models are written in the RuleBase Environment Description Language (EDL), a dialect of the SMV language [6]. EDL is somewhat

similar to common hardware description languages (HDLs), but it also supports non determinism and multiple environments.

Environments are linked to the design and to other environments by signal names. Signals produced by the environment will match and drive design signals that have the same name even if they are internal to the design, which is a way to abstract by overriding. Signals (both output and internal signals) produced by the design will match and drive environment models that require these signals. In some translation paths, design signals are converted to upper-case. Writing good environment models is an art. Good environments should be **small** and **simple**, while allowing **all and only** the legal behaviors. Environments should be small to avoid overloading the model-checker, and simple in order to be easily written, read, and maintained. Good environment models should not produce illegal behavior, or else false-negative results will be produced. On the other hand, they should model all the legal behaviors because an unmodelled behavior is a good place for bugs to hide. An attempt should be made to hide as much detail as possible using abstraction techniques (as explained later in Chapter 4).

The following are the stages of environment modeling [12]:

a. We studied the block interfaces in detail and understood the behavior of every input and every relevant output. This information can be gathered from standard bus protocols, design documents, and communication with the designers.

b. We planned the hierarchical structure of the environment models, grouping related signals and reusing components where possible.

c. Decided how to model each input. Some inputs are held constant, at least during the initial stages of verification. Usually there is a set of interesting control inputs that need detailed modeling. We have to design and implement logic to drive these signals.

d. Finally, we coded the environment logic in EDL.

- **Model Checking Engines**

Mostly BDD-based and SAT-Based model checking algorithms are used in RuleBase. The main engine, which appears in the basic, classical version of RuleBase, is the BDD-based SMV engine. In a BDD, every state variable has a distinct level, from 1 to n, where n is the number of state variables. The order in which the levels are allocated to the state variables has a large impact on the size of the BDD. For example, a design whose verification with a good BDD requires 30 MB of memory may require 300 MB or more with a bad order. Therefore, it is important to find a good order. RuleBase can perform BDD reordering during model checking. This is known as dynamic BDD ordering. Because BDD ordering is extremely CPU-intensive, it is inactive by default. User should turn it on for initial runs, and feed the resulting order back into RuleBase for all consecutive runs.

- **Counter-examples & witnesses**

When a rule fails, the user needs some snapshot regarding the scenario. The waveform display window displays an execution trace that is a counter example or a witness to an assertion or a rule. The number bar at the top of the display counts the clock cycles of the fastest clock. Signals that have a textual display (e.g., enumerated constant values) only display a change in the signal value. If no value appears at time X, find the current value by looking to the left for the value at the last time the signal changed.

Figure 3.2 shows the counter example of an error found in our case study, which is described in chapter 4. The property states that "On every path, always, if (!(READ_SEL)) then On the next cycle, (edge_detect) XOR (edge_detect_retimed). The property means always when the READ_SEL signal is active low then in the next clock edge, the internal signals edge_detect and edge_detect_retimed cannot be of same logic value. The error scenario is captured with the bold dot which is shown clearly in the figure.

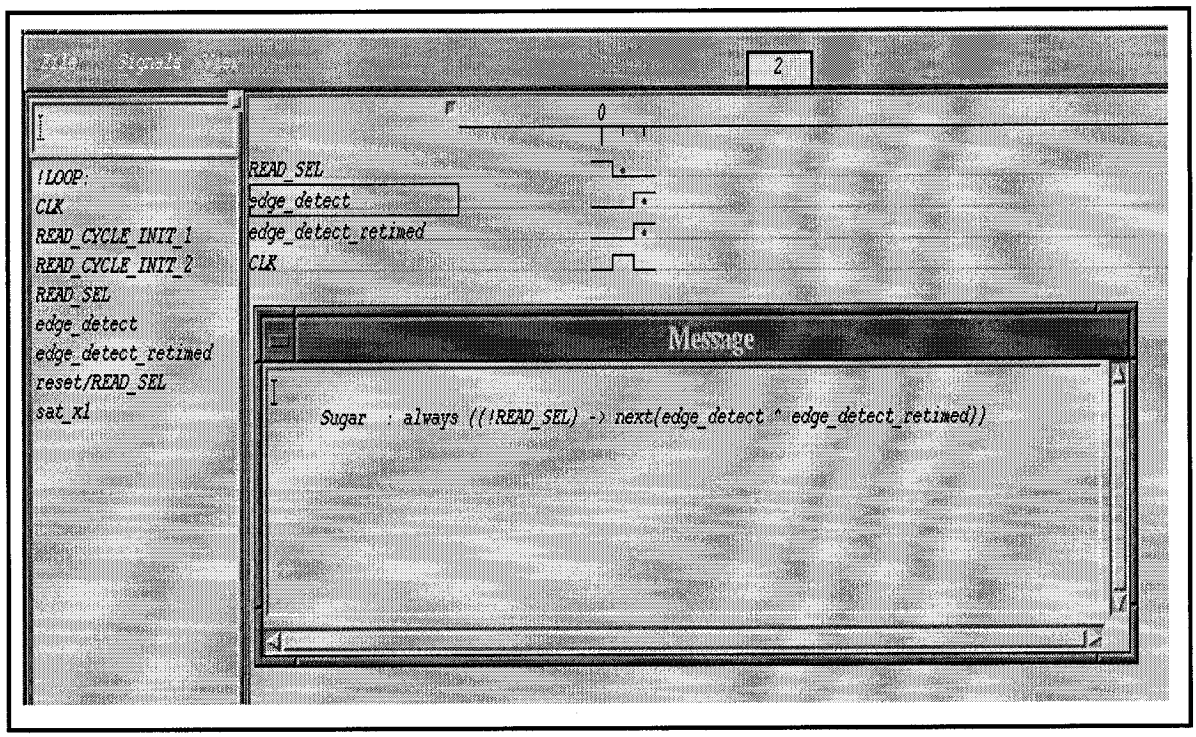

Figure 2.13: Counter Example generation of RuleBase

Chapter 3: Introduction to Standard Bus Protocols

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* IP cores on a standard bus. These IP cores 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 which in process makes the validation of such designs is becoming more and more challenging.

3.1: Role of Standard Bus Protocol Verification in SoC Design

A System-On-Chip (SoC) can be viewed as a collection of various IP cores, with interconnecting buses running among them. 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, i.e. PCI Interface, Look-Aside Interface which 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 by the IP core vendors. 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 the scenario of verifying SoC is dictated by the following issues,

- Pre-verified IP Cores with certain guarantees and confidence.
- A standard bus protocol
- IP Core specific glue to connect cores to the bus

Since the bus protocol is standard, it needs to be verified once and for all. One of the major tasks in this process is the development of bus specific protocol models i.e. RTL model for our case. These models should be general enough to incorporate all the behaviors of the interesting verification-related bus transactions. Assertion and temporal properties verified in this RTL development stage should be focused based on the Verification plan which is extracted directly from the design specification.

3.2: Look-Aside Interface as standard bus protocol

Experience in industry with SoC designs shows that most of the bugs are found in the bus protocol or the interface that connects the IP cores. To our knowledge, there is still no agreement on a standard bus protocol for system-on-chip designs. However, our thesis focuses on verifying system-on-chip designs having IP cores such as Network Processor Unit (NPU) and Network Packet Search Engine (NPSE) and the standard protocol used for this purpose is Look-Aside Interface (LA-1). A typical use for LA-1 Interface is shown in Fig. 3.1 [23]. It has the advantage of requiring comparatively few pins while still achieving very high throughput through a combination of:

- High frequency (up to 250 MHz in the first version)

- Double data rate transfers (data is sent and received on both edges of the input and output clocks)
- Concurrency (read and write transfers occur simultaneously and independently)
- Source-synchronous clocking

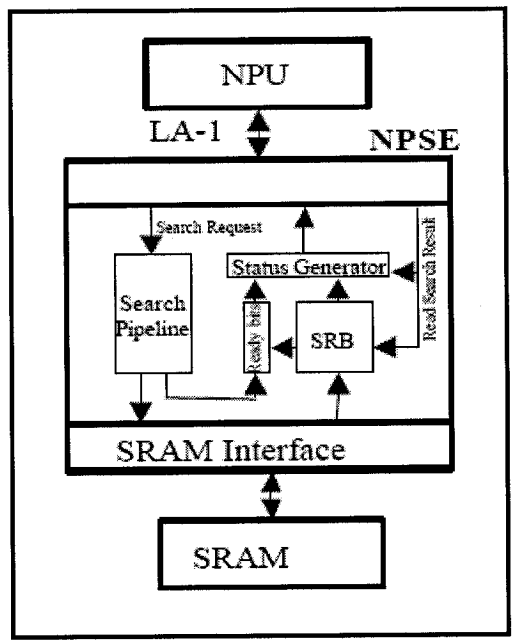

Figure 3.1: Look-Aside Interface used between NPU and NPSE

The LA -1 data interface is logically 32 bits wide (or 36-bits wide including parity), but physically only 16 bits wide, with each half of a 32-bit word being transferred on successive positive and negative edges of the clock, thus keeping the pin-count reasonable despite having separate read/write data buses. With a 24-bit address bus, the LA -1 interface requires a total of about 70 signal pins. The implementation of LA -1 is capable of maintaining an independent 32-bit transfer in each direction (read and write) in every clock-cycle at up to 250 MHz, for an aggregate throughput of up to 2 GB/s. The high frequency of operation is maintained by use of low-voltage (1.5V) High Speed Transceiver Logic (HSTL) buffers, and source-synchronous clocking. The principle of

source-synchronous clocking is to allow signal clocks to be deliberately skewed by signal sources to compensate for output delays, and then transmitted along with the signals to compensate for flight times, thus providing a reasonable data window for set-up and hold times with respect to the active edge of the clock at the signal destination, even at very high frequencies. Lastly, the LA-1 interface is an open standard which has been developed and endorsed by multiple memory and network device vendors i.e. Cypress, IDT, NEC, Samsung and so is expected to achieve high adoption rates over a range of compatible network products. In the following sections of this chapter, we give some brief overviews of the other standard bus protocols currently used in the Industry. Chapter 4 of this thesis report will discuss the main features of Look-Aside Interface (LA-1) standard in details.

3.3: PCI Local Bus Protocol

The PCI Local Bus [24, 25, and 26] 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. PCI bus signals 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 signaling, 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.

3.4: PCI-X Bus Protocol

PCI-X is the latest implementation of PCI [24, 25, and 26]. It was adopted as industrial standard ratified by the PCI-SIG. Using the same 64-bit architecture as the current standard, PCI-X has tremendously increased the clock speed to 533 MHz, allowing transfer speeds up to 4 GB/sec and it is backward compatible with standard PCI cards.

Further-more, the PCI-X bus plays a vital role in today's System-On-Chip (SoC) designs involving various components connected using high-speed standard buses. The introduction of the PCI-X technology has provided the necessary bandwidth and bus performance needed to avoid the I/O bottleneck, thus achieving optimal system performance. For instance, version 2.0 of PCI-X specifies a 64-bit connection running at speeds of 66, 133, 266, or 533 MHz, resulting in a peak bandwidth of 533, 1066, 2133 or 4266 MB/s, respectively.

PCI-X provides backward compatibility by allowing devices to operate at conventional PCI frequencies and modes. Moreover, PCI-X peripheral cards can operate in a conventional PCI slot, although only at PCI rates and may require a 3.3 V conventional PCI slot. Similarly, a PCI peripheral card with a 3.3 V or universal card

edge connector can operate in a PCI-X slot; however the bus clock will remain at a frequency acceptable to the PCI card.

Fig. 3.2[27] shows the general architecture of PCI-X with one Initiator (Master) and Target (Slave). There is an arbiter that performs the bus arbitration among multiple Initiators and Targets. Unlike the conventional PCI bus, the arbiter in PCI-X systems monitors the bus in order to ensure good functioning of the bus. The general architecture and the port operation overview for Look-Aside Interface i.e. our case study are elaborated in Chapter 4.

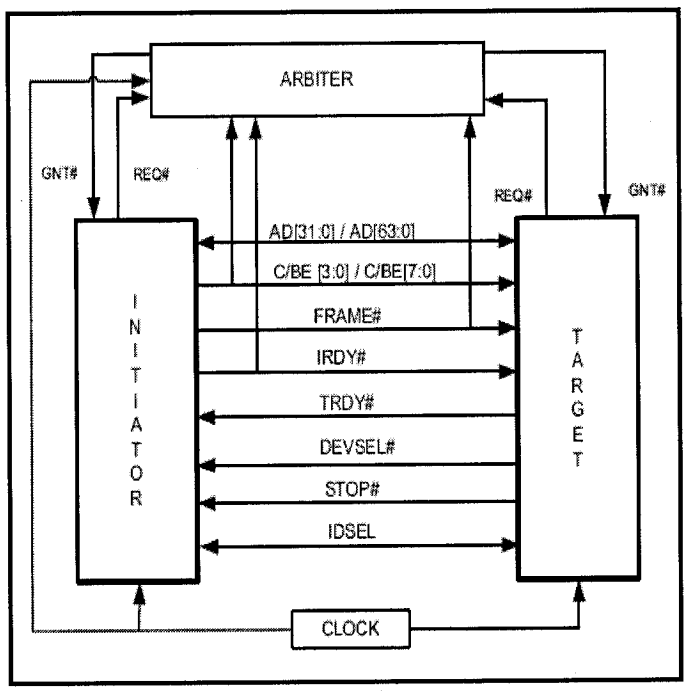

Figure 3.2: General Architecture of PCI-X

Chapter 4: Functional Specification of Look-Aside Interface (LA-1 Standard)

After evaluating several options, we adopted the Network Processor Forum's Look-Aside Interface (LA-1) standard to be the topic of our interest. This is a fast and narrow, low voltage interface based on the QDR II (tm) SRAM interface standard. Moreover, the LA -1 interface is an open standard which made it an ideal candidate for us to implement the RTL model in house and then use it for our case study. In this chapter we will first introduce the basic features of Look-Aside Interface (LA-1) standard and then discuss the verification plan chalked out directly from the verification plan. Finally, we will enlighten the readers about the verification methodology which was applied on this interface.

As line rates in core networks increase from OC-48 to OC-192, core routers will need to process hundreds of millions of packets per second. This creates the need for packet processing elements, such as network processors, to offload tasks to look aside co-processors to increase the system performance. The look-aside interface to coprocessors, such as Network Search Engines (NSE) and Classifiers, then becomes a key choke point in the packet processing architecture. To help curb this bottleneck, the Network Processing Forum (www.npforum.org) released the standard look aside interface [3], dubbed LA-1, that accelerates and standardizes the movement of packets between data plane and co-processor solutions.

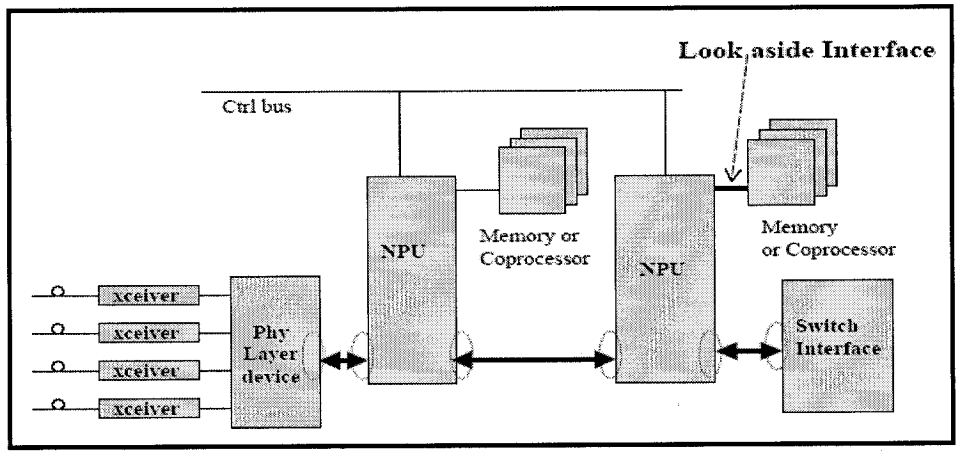

Figure 4.1: System Level Diagrams

The work on the LA-1 specification, the Look Aside Interface to network-processing elements (NPEs), started at the beginning of 2001 and became an approved Network Processing Forum (NPF) specification in April 2004. Major work was done in the Look Aside Task Group, Hardware Working Group of the NPF whose goal is to define and deliver hardware interfaces for the components located adjacent to an NPE via the Look Aside Interface.

The LA-1 interface specification is based on separate dual data rate (DDR) buses for data inputs and data outputs. Using DDR interfaces, data is clocked on the rising and falling edges of the clock signals. This effectively doubles the bandwidth of the interface without increasing the clock speed or the bus width. Although modeled on an SRAM interface, the LA-1 specification aims to accommodate other devices as well, such as classifiers and encryption co-processors. Overall, the LA-1 interface operates at clock speeds between 133 and 200MHz. The minimum performance specification for lookup co-processors is four lookup operations at OC-48 or one lookup operation at OC-192. The main goal of our research is to apply assertion-based verification, in order to verify

Look Aside Interface (LA-1 Standard). Afterwards, we verified the same assertion monitors in the form of model checking using PSL/Sugar 2.0.

4.1: LA-1 Interface Overview and Major Features

The LA-1 interface supports unidirectional 16-bit read and write interfaces. These data inputs and outputs operate simultaneously, thus eliminating the need for high-speed bus turnarounds (i.e. no dead cycles are present) as shown in Figure 4.2 [3]. Access to each port is accomplished using a common 27-bit address bus. Addresses for reads and writes are latched on rising edges of K and K# input clocks, respectively. Each address location is associated with two 16-bit data words that burst (**2-WORD**) sequentially into or out of the device.

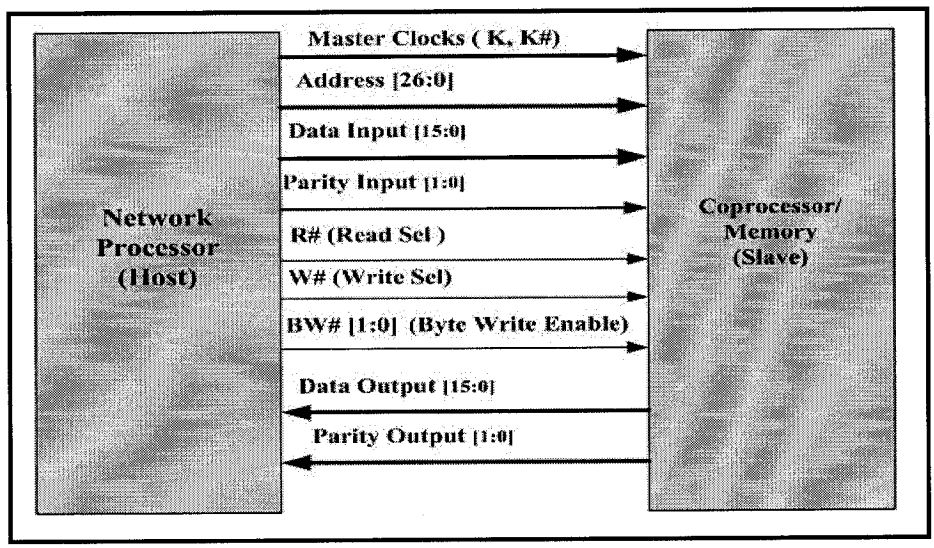

Figure 4.2: LA-1 Interface Major Bus Signals

The LA-1 interface major features include [3]:

- Concurrent read and write operation
- Unidirectional read and write interfaces
- Single address bus

- 18 pin DDR data output path transfers 32 + 4 bits of even byte parity per read.
- 18 pin DDR data input path transfers 32 + 4 bits of even byte parity per write
- Byte write control for writes

4.2: Look Aside Interface Architecture

A single Look Aside Interface includes a master clock pair, address, control pins, 16 bit Data in, 2 bit input parity pin, byte enable pins for write operations and 16 bit Data out and 2 bit data for Output parity. The master clocks are ideally 180 degrees out of phase with each other. The LA-1 interface ports follow a few simple rules:

- W#[1], R# and BW# are always captured on the rising edge of K clock.
- For address and data are captured on the rising edges of K and K# clocks.

The data path for a single bank LA-1 Interface is shown in **Figure 4.3** [18].

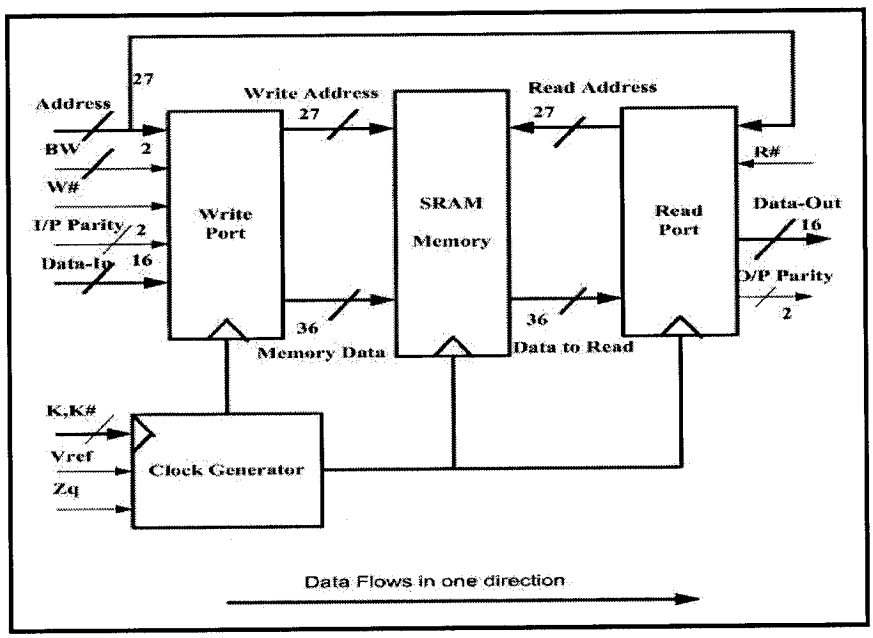

Figure 4.3: Data path for a Single Bank LA-1 Interface

The LA-1 port depth expansion is done by adding a set of port enable inputs (1 bit each),

[1] # sign means the control input is active low. The control inputs W#, R#, BW# are all active low inputs.

which are the most significant two bits of the Address bus and thus four single banks LA-1 Interface can used. Programmability of two enable inputs (E1 and E2) i.e. most significant two bits of the Address bus, would allow four banks of depth expansion to be accomplished with no additional logic. By programming the enable inputs of four LA-1 ports in binary sequence (00, 01, 10, and 11) and driving the enable inputs with two address outputs, four LA-1 ports can be made to look like one port with a larger address space to the system (**Fig. 4.4**).

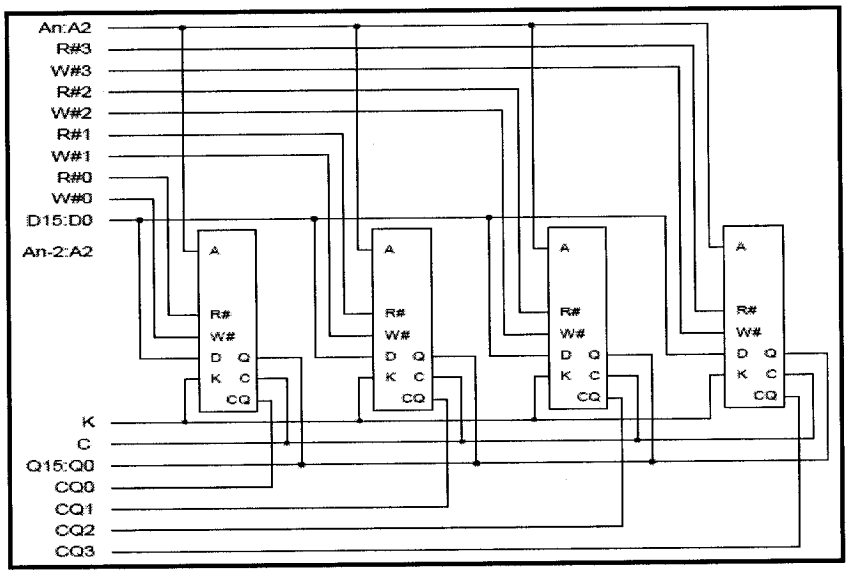

Figure 4.4: Look-Aside Interface Four Bank Data Path

4.3: Look Aside Interface Port Operation Overview

The key objective of Single Bank LA-1 architecture is to clearly distinguish read and write ports therefore, QDR architecture is designed to offer the best performance on alternate read and write cycles. The **2-word burst QDR SRAM** can indefinitely sustain both a 2-word read and a 2-word write each clock cycle. Internally, the first half-clock cycle is used to execute the read function, and the second half-clock cycle is used to execute the write function. The address bus is shared for the read and writes data ports.

The rising edge of a master clock signal "K" is used to register the read address. The falling edge of this clock signal "K" (rising-edge of K#) is used to register the write address.

4.3.1: Write Port Operation Overview

A write cycle is initiated by asserting WRITE_SEL (W#) low at rising edge of K clock. The address of the Write cycle is provided at the following edge of K# clock which 180 degrees out phase from clock K. BW# (Active-low byte-write inputs) are used to enable or block write of a specific byte a write cycle initiated with W#. BW0# controls D [0:7] and DP 0, while BW1# controls D [8:15] and DP 1 (Table 4.1) [3].

Byte Enable	BW 1#	BW 0#
Data Input pins	D [15:8]	D[7:0]
Parity Input pins	DP 1	DP 0
K ↑	Byte 0 Bits [31:24] Parity Bit 0	Byte 1 Bits [23:16] Parity Bit 1
K# ↑	Byte 2 Bits [15:8] Parity Bit 2	Byte 3 Bits [7:0] Parity Bit 3

Table 4.1 Control & Data pin names vs. 32 bit write data alignment

In the case of an SRAM, a read can immediately follow a write even if they are to the same address. In the case of a coprocessor, there will be a minimum latency between a write and a read to the same address that is dependent on the architecture of the coprocessor. Latency bounds on this operation are out of scope of this specification. Assumes a write cycle was initiated via W# low. BW0# and BW1# are sampled at data in times and can be altered for any portion of the burst write operation provided that input setup and hold requirements are satisfied.

Data In Sample Time	BW1#	BW0#	D[15:8] DP[1]	D[7:0] DP[0]
K ↑	1	0	Data In	Don't Care
K# ↑	0	1	Don't Care	Data In

Table 4.2 Write sequence using Byte Write Enable (BW#)

Resulting Write Operations from Table 4.2 are shown in Table 4.3.These 32 bits of data along with the 4 bits of parity is sent to the Memory with the respective address.

Byte 0 D [15:8], DP [1]	Byte 1 D [7:0], DP [0]	Byte 2 D [15:8], DP [1]	Byte 3 D [7:0], DP [0]
Written	Unchanged	Unchanged	Written

Table 4.3 Resulting formation of 32 bits of data

4.3.2: Read Port Operation Overview

A read cycle is initiated by asserting R# low at K rising edge and the read address is presented on A. Data is delivered after the next rising edge of K. (Table 4.4)

Data Output pins	Q [15:8]	Q [7:0]
Parity Output pins	QP 1	QP 0
K ↑	Byte 0 Bits [31:24] Parity Bit 0	Byte 1 Bits [23:16] Parity Bit 1
K# ↑	Byte 2 Bits [15:8] Parity Bit 2	Byte 3 Bits [7:0] Parity Bit 3

Table 4.4 Data output pin names vs. 32 bit read data alignment

4.3.3: SRAM Port Operation Overview

The LA-1 interface uses an SRAM style memory mapped structure. Address pins, as needed, are used to address logical registers on the device. The NPU uses register style read and write operations to initiate co processor actions, retrieve results, and optionally provide in band management. A memory mapped logical layer provides a flexible and minimum weight interface for co processor applications. Therefore, co processor architectures may provide differentiation and innovation because the logical layer does not excessively limit the designer. One of the best advantages of SRAM[2] devices over SDRAM devices are the ease of interfacing them in terms of control signals. For example, there is no need for sending a refresh command to the memory device. The usage of the address bus, in 2-word burst devices, the read address location is considered on the rising edge of K clock and the write address location on the falling edge of K clock as shown in Fig. 4.5. According to the figure, the address bus A runs in DDR mode which is similar to Data Bus D which also runs in DDR mode. Moreover, the Data Clock relationship is strictly followed for the Write and Read Port. Therefore, in Write Port, Write Data must be center aligned in regard to K clock when sending the data. On the contrary, Data are sent with a guaranteed interval delay with respect to K and K# clock from the Read Port.

[2] A 2-word burst QDR SRAM devices able to handle a 167 MHz clock in Virtex-II FPGAs (-5 speed grade) QDR SRAMs have read and write data buses, both operating in DDR mode. On the write bus, the clock needs to be center aligned with the data. This physical placement is advantageous for the memory device (**Fig. 4.5**)

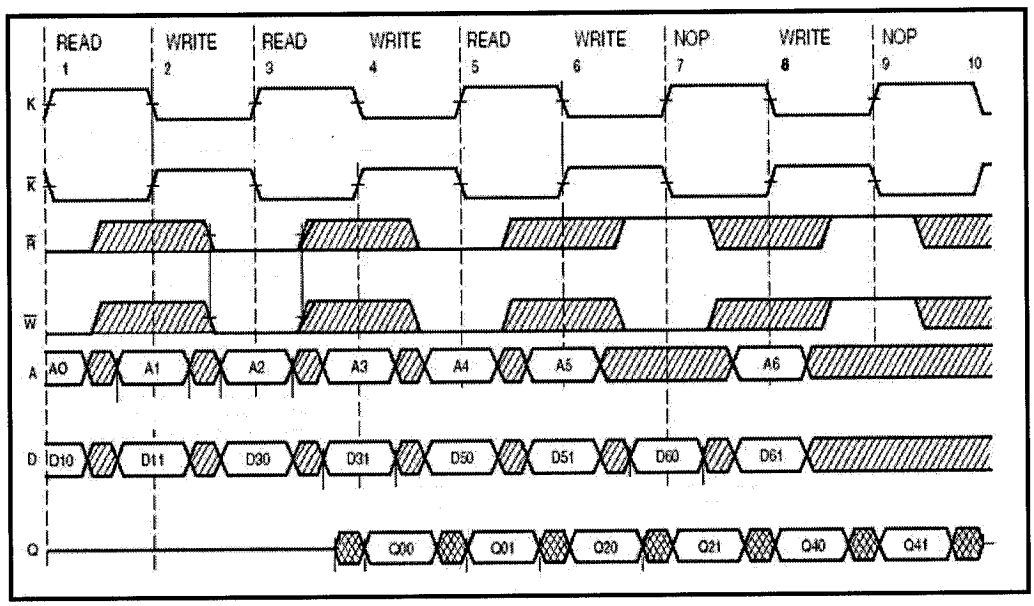

Figure 4.5: Simplified Timing Diagram for LA-1 Port Operation

In our RTL implementation of the Design, we assumed the echo C and C# clock inputs are tied high and then according to the design specification the device reverts to K and K# control of the outputs, allowing the device to function as a conventional pipelined read device. In the later chapters of this thesis, we will discuss the Functional Coverage and the related verification plan for the Look-Aside Interface. In addition, we will also explore the implementation of the verification plan with respect to OVL and Sugar 2.0.

Chapter 5: Functional Coverage and Verification Plan

Functional verification comprises a large portion of the resources required to design and validate a complex system. Often, the validation must be comprehensive without redundant effort. To minimize wasted effort, coverage is used as a guide for directing verification resources by identifying tested and untested portions of the design. Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design. Broadly speaking, there are two types of coverage metrics. Those that can be automatically extracted from the design code, such as code coverage, and those that are user specified in order to tie the verification environment to the design intent or functionality. This latter form is referred to as Functional Coverage.

Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the verification plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions—captured as features of the test plan—have been observed, validated and tested.

The key aspects of functional coverage are:

- It is user-specified (assertions), and is not automatically inferred from the design.
- It is based on the design specification (i.e., its intent and designer's valid assumption) and is thus independent of the actual design code or its structure.

Since it is fully specified by the user, functional coverage requires more up front effort i.e. assertion and property writing skills. Functional coverage also defines a more

structured approach to verification. Although functional coverage can shorten the overall verification effort and yield higher quality designs, the inadequate translation (informal translation) of design specification can impede its adoption. Nowadays, Functional coverage strategy is not only synonymous with simulation but also involves exhaustive usage of simulations, assertions, verification plan and formal coverage in order to make sure all aspects of the design under test meets the design specification document. Therefore, Functional Coverage needs a reactive verification suite as shown in Fig. 5.1.

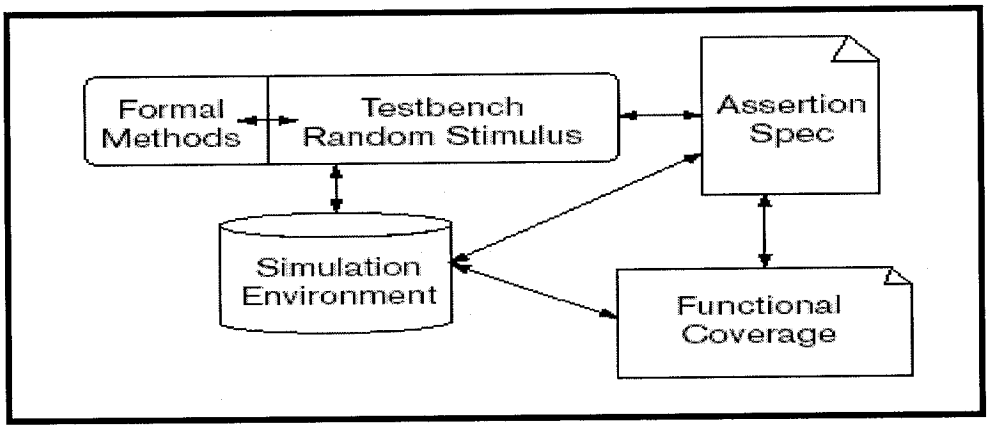

Figure 5.1: Reactive Verification Suites for Functional Coverage

With the aid of a reactive verification suite, the typical scenarios of error cases, corner cases and protocols are covered in the verification system. Functional coverage also measures if all important combinations of input stimulus have been exercised at different states of the design-under-test. This is achieved by applying Formal methods (model checking) on the design under test. In addition, Functional coverage elevates the discussion to specific transactions or bursts without overwhelming the verification engineer with bit vectors and signal names i.e. flagging an **OVL_ERROR** points a specific transaction error of the design. Therefore, the bulk of low-level details are hidden

from the report reviewer. This level of abstraction enables natural translation from verification plan items to coverage results.

Thus, functional coverage is very important because:

- It allows the verification engineer to focus on key areas of the design that need most attention based on the verification plan. The verification plan of Look-Aside Interface is described later in this chapter.
- It explains the verification engineer how much verification is enough.
- It improves the efficiency of test generation i.e. it includes assertions, model checkers.
- It improves the quality of tests.
- It avoids repetitive generation of tests for the same set of combinations of input stimulus.

One of the notable problems in conducting functional verification is that without assertion or property specifications is that there is no easy way to know what checks are being targeted or monitored. The tools cannot process the information produced from a simulation run to determine specific checks that have been detected as failures, exercised or re-exercised. In other words, without assertion specification and model checking, the measurement of quality is not practical, and without this measurement, the entire design verification process suffers from a lack of coherent information about its progress. To economize the verification effort, Functional coverage methodology take the same assertion specifications and model checking results to provide a detailed account of what tests, checks, scenarios and data have been exercised. Without access to this comprehensive functional coverage methodology, random test generation has to be

manually adjusted for each simulation run. The input constraints and seeds are modified so that more scenarios can be exercised to catch bugs. This process of manual trial and error is inherently inefficient and incomplete. But, the availability of functional coverage in terms of assertions and model checking during run-time can change this situation remarkably. The error report obtained from VCS 7.0.1 helped us to drive the test benches to make necessary adjustments as soon as an **OVL ERROR** is determined. On the other hand, the error report from RuleBase helped us to pinpoint unknown bugs which were dormant in the course of our VCS simulation with OVL. Therefore, the overall process gave us an enriched functional coverage for the design, where functional verification and formal verification both played its crucial part. Thus, for the implementation of an intelligent testbench which was discussed in Chapter 1, the development of a reactive testbench is the principal and vital step. With this view in mind, our primary aim was to create an effective verification plan of Look-Aside Interface (LA-1 Standard). In the following section of this chapter we will discuss the verification plan with respect to the main features of Look-Aside Interface as described earlier.

5.1: Verification Plan for the Write Port

- **Write Port main features**
- Asserting the write-select W# (WRITE_SEL) input low at the master-clock
- CLK_K, rising edge initiates a write cycle.
- The following master-clock-bar CLK_K1, rising edge provides the address for the write cycle. At the same cycle, you can expect data at the rising edge of the master clock and master clock bar.

- By using the byte-write control (BW#) signals, any input byte of D (Data In) may be masked in any write sequence or the full input word may be passed through.
- Fig. 5.2 gives detailed timing diagram overview of the Write Port operation.

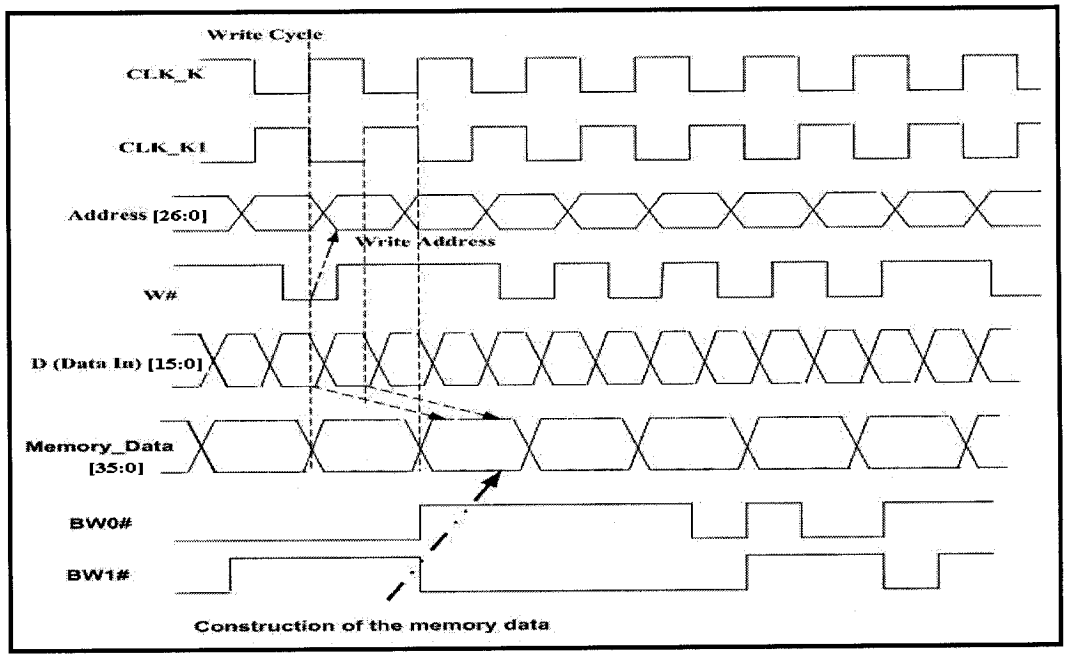

Figure 5.2: Timing Diagram of the Write Port of LA-1 Interface

- **Testcase 1 : WRITE_CYCLE_INIT**[3]

This testcase checks the proper initialization of the Write Cycle, based on asserting W# low at the rising edge of CLK_K and then verifying that at following edge of CLK_K, that the internal signal WRITE_CYCLE_INIT_1 is logic high. The sequence is described below:

- Assert W# (WRITE_SEL) Low
- At the rising edge of CLK_K verify that WRITE_CYCLE_INIT_1 is '1'
- Note that WRITE_CYCLE_INIT_1 is an internal signal.

[3] Each testcase is named with a unique name. The same names were used for defining assertion monitors in OVL and property specification using Sugar 2.0. We strictly named all our assertions. This eased our effort associated with debugging assertion condition failures, **OVL ERROR** in our case.

- **Testcase 2 : WRITE_ADDRESS_SAMPLE**

This testcase checks whether the proper Write Address was sampled in the Write Port. If a Write Cycle is initiated at the rising edge of CLK_K then the address sampled at the following rising edge of CLK_K1 is the Write Address. The sequence is given below.

  - Assert W# (WRITE_SEL) low at the rising edge CLK_K.
  - Store the address sampled at the following rising edge of CLK_K1
  - Compare this address with WRITE_ADDR in the RTL.

- **Testcase 3 : WRITE_DATA_PASSTHROUGH_MSW & LSW**

This testcase checks that BW# controls writing of Data In[15:0]. When BW# = "00" then if the Write Cycle is initiated then the sampled Data In[15:0] is written on the rising edge of CLK_K and CLK_K1. The sequence is given below.

  - Assert BW# [1:0] to "00" (active low)
  - Initialize the Write Cycle and at the rising edge of CLK_K , store the Data_In [15:0] and the following rising Edge of CLK_K1 store the Data_In [15:0]
  - Verify that the stored data of CLK_K1 rising edge is equal to the LSW of Memory Data and the stored data of CLK_K rising edge is equal to the MSW of Memory Data sampled at the following CLK_K rising edge.

- **Testcase 4 : WRITE_DATA_ALIGNMENT_1**

This testcase checks that BW# controls writing of specific input byte of Data In[15:0]. When BW# = "01" and Write Cycle is initiated then the LSB of the Data In[15:0] sampled on the rising edge CLK_K is equal to the Memory Data[23:16] and

the LSB of the Data In sampled at the rising edge of CLK_K1 is equal to the Memory Data[7:0]. The sequence is described below. Note that this scenario is checked with two different assertion monitors.

- Assert BW# [1:0] to "01"
- Initialize the Write Cycle and at the rising edge of CLK_K and store the Data In [15:0]
- At the following rising Edge of CLK_K1 store the Data_In [15:0]
- Verify that the stored data of CLK_K1 rising edge is equal to the LSB of Memory Data and the stored data of CLK_K rising edge is equal to the Memory Data sampled at the following CLK_K rising edge.

- **Testcase 5 : WRITE_DATA_ALIGNMENT_2**

This testcase checks that BW# controls writing of specific input byte of Data In[15:0]. When BW# = "10" and Write Cycle is initiated then the MSB of the Data In[15:0] sampled on the rising edge CLK_K is equal to the Memory Data[31:24] and the MSB of the Data In sampled at the rising edge of CLK_K1 is equal to the Memory Data[15:8]. The sequence is described below. Note that this scenario is checked with two different assertion monitors.

- Assert BW# [1:0] to "10".
- Perform the second and third steps as described in the above test case.
- Verify that the stored data of CLK_K1 rising edge is equal to the [15:8] of Memory Data and the stored data of CLK_K rising edge is equal to the [31:24] bits of Memory Data sampled at the following CLK_K rising edge.

- **Testcase 6 : WRITE_DATA_NOP**

This testcase checks that if BW# = "11" and a Write Cycle is initiated then Memory Data retains its old value. The sequence is given below:

  - Assert BW# to "11"
  - Verify that at posedge of CLK_K, Memory Data keeps the previous value. (no change)

5.2: Verification Plan for the READ Port

- **Read Port main features**
- Asserting the read-select input R# (READ_SEL) low at the CLK_K rising edge initiates a read cycle
- At the same rising edge of CLK_K, the address bus presents the read address (READ_ADDR).
- Data is delivered after the next rising edge of the CLK_K.
- For an SRAM, a read operation can immediately follow a write operation even if they are to the same address.

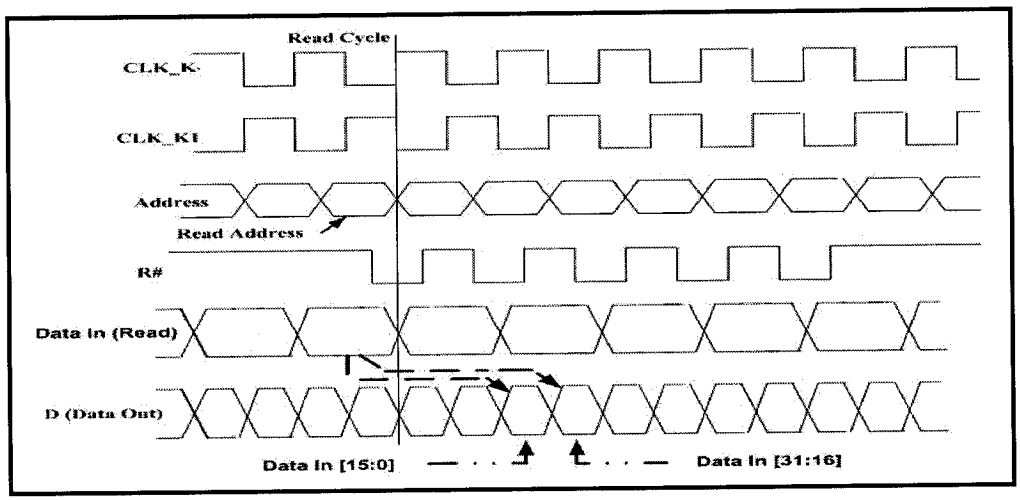

Figure 5.3: Timing Diagram of the Read Port of LA-1 Interface

- **Testcase 7 : READ_CYCLE_INIT**

This testcase checks the proper initialization of the Read Cycle, based on asserting R# low at the rising edge of CLK_K and then verifying that at following edge of CLK_K, that the internal signal READ_CYCLE_INIT_1 is logic high. The sequence is described below:

  - Assert R# Low at rising edge of CLK_K
  - Verify that the READ_CYCLE_INIT is '1'
  - Note that READ_CYCLE_INIT_1 is an internal signal.

- **Testcase 8 : READ_ADDRESS_SAMPLE**

This testcase checks whether the proper Read Address was sampled in the Read Port. If a Read Cycle is initiated at the rising edge of CLK_K then the address sampled at the same rising edge of CLK_K is the Read Address. The sequence is given below.

  - Assert R# low at the rising edge CLK_K and store the address sampled at this rising edge.
  - Compare this address with READ_ADDR in the RTL.

- **Testcase 9 : READ_DATA_ALIGNMENT_1**

This testcase checks whether the proper Read Data was sent out after the next rising edge of CLK_K after a Read cycle is initiated. The sequence is given below.

  - Assert R# low at the rising edge CLK_K store the Data In to the Read Port.
  - Compare this Data In [15:0] with the Data Out after the next rising edge of CLK_K

- **Testcase 10 : READ_DATA_ALIGNMENT_2**

This testcase checks whether the proper Read Data was sent out after the next rising edge of CLK_K1 after a Read cycle is initiated. The sequence is given below.

  - Assert R# low at the rising edge CLK_K and store the Data In to the Read Port.
  - Compare this Data In [31:16] with the Data Out at the following rising edge of CLK_K.

5.3: Verification Plan for the Memory Port

- **Memory Port main features**
- Each loaded write command and write address (WRITE_ADDR) provides the base address for a "2-beat" data transfer, so 32 data bits plus four even-byte parity bits (36 bits in Total) are transferred for each address(27 bits) loaded to the Memory.
- Memory is controlled by the master clock (CLK_K) only, as the Memory Data width is twice of input data (Data_In) width.

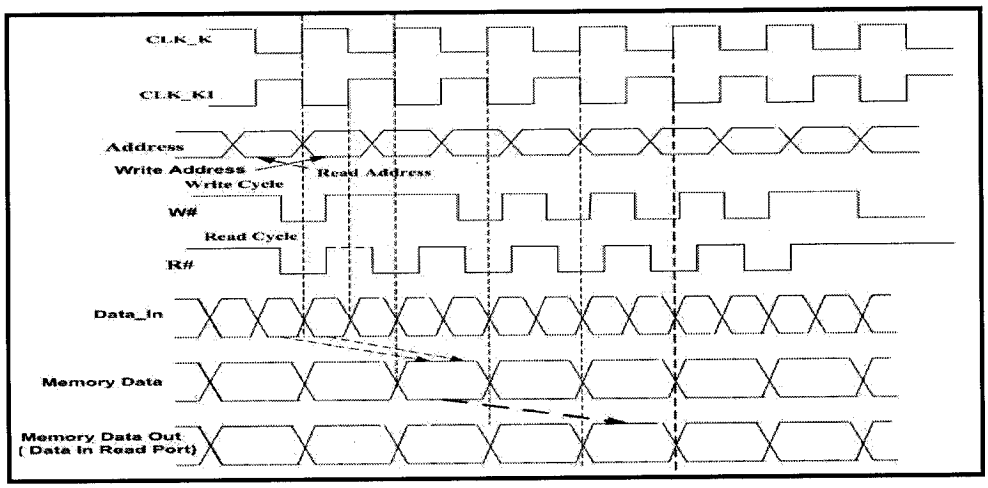

Figure 5.4: Timing Diagram of the Memory Port of LA-1 Interface

- **Testcase 11 : MEMORY_VALID_ADDRESS_DATA**

This testcase checks whether the Memory Address and Memory Data do not read X value. The sequence is given below.

  - Initialize the memory in the testbench
  - Generate random address through the test bench.
  - Verify that the memory address do not read X value.
  - Verify that the memory data do not read X value.

- **Testcase 12 : MEMORY_THROUGHPUT**

This testcase checks if the Memory is properly writing and reading the data. Please note that the Read Data from the Memory is send to the Read Port as input data. The sequence is given below.

  - Initialize the memory in the testbench
  - Generate random address and Input data through the test bench
  - Assert the Write Cycle and Store the Write address and Memory Data
  - After 2 clock cycles of CLK_K) assert READ_SEL and make sure Read Address is equal to Write Address.
  - Verify that the stored Memory Data is equal to the Data_In of the Read Port.

5.4: Verification Plan for the Look-Aside Interface (TOP Level)

- **Look-Aside Interface Top Level Main Features**
- Data path of Look-Aside Interface consists of a Write Port, Memory and Read Port (Fig 4.3). Thus it inherits all the feature of the Write Port, Read Port and the Memory.

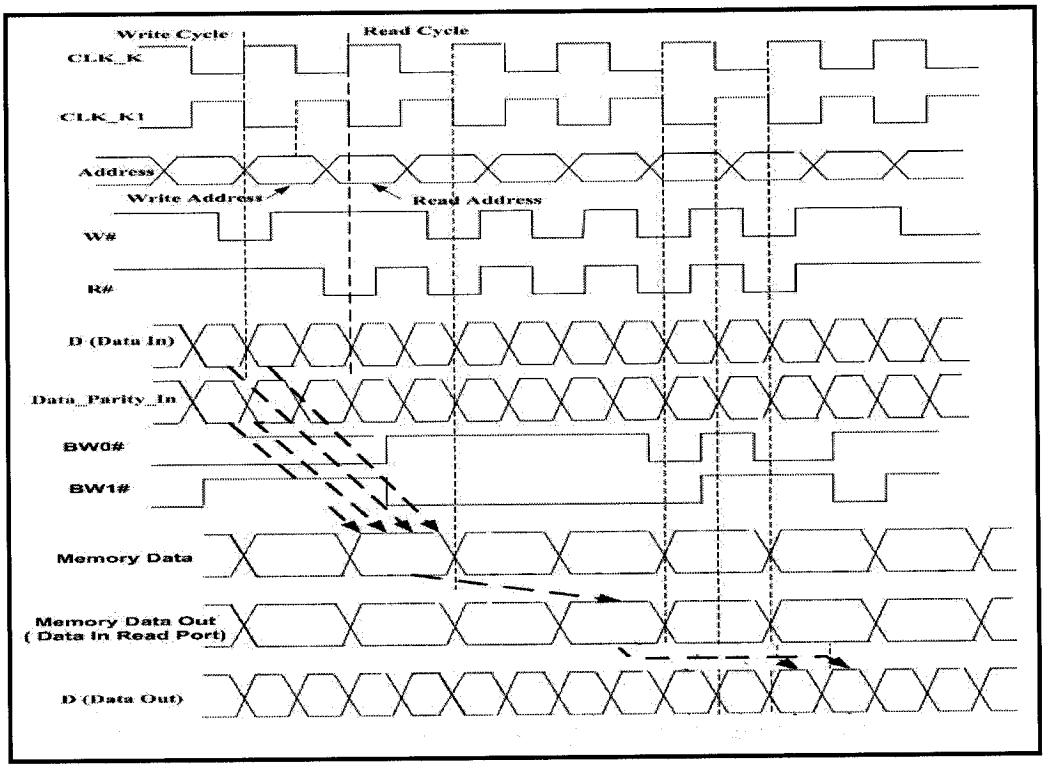

Figure 5.5: Timing Diagram of the TOP Level of LA-1 Interface

- **Testcase 13 : LA1_TOP_PASSTHROUGH**

This testcase checks whether the Data Pass through mode (When BW# = "00") works properly in the Top Level of the RTL model. The sequence is given below.

- Set BW# to "00"
- Initialize the Write Cycle at the rising edge of CLK_K and store the Data In [15:0]
- At the following rising Edge of CLK_K1 store the Data_In [15:0]
- Verify that the stored data at rising edge of CLK_K1 is equal to the Data Out sampled at the 4th rising edge of CLK_K
- Verify that the stored data at rising edge of CLK_K is equal to the Data Out sampled at the 5th rising edge of CLK_K

- **Testcase 14 : LA1_TOP_WRITE_READ**

This testcase checks whether in the Write and Read is being done correctly at the top level. The sequence is given below.

  - Set BW# to "00"
  - Initialize the Write Cycle at the rising edge of CLK_K and store the Data In [15:0]
  - At the following rising Edge of CLK_K1 store the Data_In [15:0] and the address.
  - At the 4th rising edge of CLK_K read the stored address and verify that Memory Data Out[15:0] is equal to stored data, stored at rising edge of CLK_K1.

In Chapter 6 and 7, we will enlighten the reader about the usage of this Verification plan which served as the basis of the intelligent testbench. We will also elaborate a clear relationship between the Verification Plan and derivation of the OVL Assertions and Model Checking properties using Property Specification Language (PSL/Sugar2.0)

Chapter 6: Applying OVL Assertion Monitors to LA-1 Interface Verification Plan

The usage of assertion monitors was illustrated clearly, with its application on the verification plan. Hence, in the following sections, the OVL assertion monitors are used in order to verify the testcases which were described in the verification plan earlier. For our verification purpose we used assert_always,assert_never and assert_next monitor. Please note that the **assert_always** and **assert_never** assertion continuously monitors the *test_expr* at every positive edge of the triggering event or clock *clk*. On the other hand, **assert_next** verifies the assertions based on a clock cycle relationship. In the following section, we enlighten the use of some assertion monitors in context of the Verification plan. The rest of the assertions are described in Appendix A. Appendix C gives examples of the Write Port RTL model of the design along with the embedded assertion monitors.

6.1: OVL Assertions for WRITE Port

- **Assertion 1:**

  assert_always #(0,0,"Write cycle should be initialized")

  valid_write_cycle_init // *inst_name* of assert_always monitor

  (CLK_K, // Clock CLK_K signal of the design

  write_enable, // signal indicating complete initialization

  (WRITE_CYCLE_INIT_1 == 1'b1)); // *test_expr*

This assertion is continuously exercising Testcase 1 for the Write Port (described earlier) and validating the Write cycle initialization scenario.

- **Assertion 3:**

  assert_always #(0,0,"WRITE_DATA PASSTHROUGH LSW")

  valid_data_out_passthrough_lsw

  (CLK_K1, //Clock CLK_K1 signal of the design

  WRITE_CYCLE_INIT_3, // signal indicating initialization

  (DIN_NEGEDGE == DATA_WRITE_MEM_OUT[15:0])); *//test_expr*

This assertion is continuously exercising Testcase 3 (described earlier) and validating the Pass-through Mode. *WRITE_CYCLE_INIT_3* is the internal flag which is initializing the Write Cycle and setting BW# to "00". DIN_NEGEDGE is an internal register that is storing the value of Data_In.

6.2: OVL Assertions for READ Port

- **Assertion 8:**

  assert_always #(0,0 " Read Address is sampled at the rising edge of CLK_K"

  valid_read_address

  (CLK_K, //Clock CLK_K1 signal of the design

  read_address_enable, // signal indicating initialization

  (stored_address == READ_ADDR)); *//test_expr*

This assertion is continuously exercising Testcase 8 (described earlier) and validating the correct sampling of the Read Address. **read_address_enable** is the internal flag which is initializing the Read Cycle. Stored_address is an internal register that is storing the value of address at the rising edge of CLK_K and when R# (READ_SEL), is asserted low.

- **Assertion 10:**

  assert_always #(0,0," At the rising edge of CLK_K Byte 0 and Byte 1 of the data

coming from the memory and entering the read port will be equal to the Data out from the read port")

valid_data_out_k (CLK_K, // Clock CLK_K signal of the design

READ_CYCLE_INIT_3, // signal indicating initialization

(stored_data_K[31:16] == DATA_OUT); *//test_expr*

This assertion continuously exercisedTestcase 10 (described earlier) and validated the correct sampling of the Data Out. READ_CYCLE_INIT_3 is the initialization flag. Stored_data is an internal register that is storing the value of Data In to the Read Port at the rising edge of CLK_K and DATA_OUT is value of the output data from the Read Port sampled after the next rising edge.

6.3: OVL Assertions for Memory Port

- **Assertion 11:**

assert_never #(1,0,"Data In to the memory cannot have X ")

invalid_data_memory_in (CLK_K, //Clock CLK_K signal of the design

invalid_check_enable, // signal indicating initialization

^DATA_WRITE_IN [35:0] === 1'bX); *//test_expr*

This assertion continuously exercises Testcase 11 and validates the input Data to the memory. The assertion flags an error when it reads an *'X'* in the input data of the memory. This assertion makes sure that the memory is properly initialized with a memory initialization file (.mif) in the simulation environment. Note that in the VCS simulation environment we have used a behavioral model of the SRAM. As the top level of the RTL model uses all the low level modules, therefore, we just needed to instantiate the top level in the our simulation environment, written in verilog and run our

simulations using VCS 7.0.1 (Results are discussed in the following section). All the assertion monitors were initialized by default when the Top Level RTL model was simulated.

6.4: Verification Result and Error Analysis using OVL

The report file was generated using the **vcs7.01** with a timescale of **1ns/1ns**.

OVL ASSERTION	Datapath Module	OVL ERROR	OVL Time stamp
Assertion 1	Write Port	No	-
Assertion 2	Write Port	No	-
Assertion 3	Write Port	Yes	200 ns
Assertion 4	Write Port	Yes	220 ns
Assertion 5	Write Port	No	-
Assertion 6	Write Port	No	-
Assertion 7	Write Port	No	-
Assertion 8	Write Port	No	-
Assertion 9	Write Port	No	-
Assertion 10	Read Port	No	-
Assertion 11	Read Port	Yes	260 ns
Assertion 12	Memory	Yes	120 ns
Assertion 13	Memory	No	-
Assertion 14	TOP Level	No	-
Assertion 15	TOP Level	Yes	260ns

Table 6.1 OVL Assertion Verification Report

Based on the error report as shown in Table 6.1, we pinpointed the error, by viewing the waveform dump of the RTL design. Viewing the error report, we can clearly observe that although there was an environment (testbench) which generated directed and random stimulus for each testcase scenario, all the assertions were initialized at the same instant. Therefore, all the assertions are acting like a constant watchdog for the functional verification process. By default, this gives us a better functional coverage for verification. Fig. 11 shows the error at time 200 ns as reported by VCS. The reason for the OVL error was pretty much clear. We can clearly see that in the pass-through mode which means when BW# (Byte_Write_Enable) is "00", DATA_WRITE_MEM_OUT [31:16] is not equal to the value of the Data_In [15:0] sample at the rising edge of CLK_K. The flagging of the error right prompted us to investigate the RTL and fix the bug.

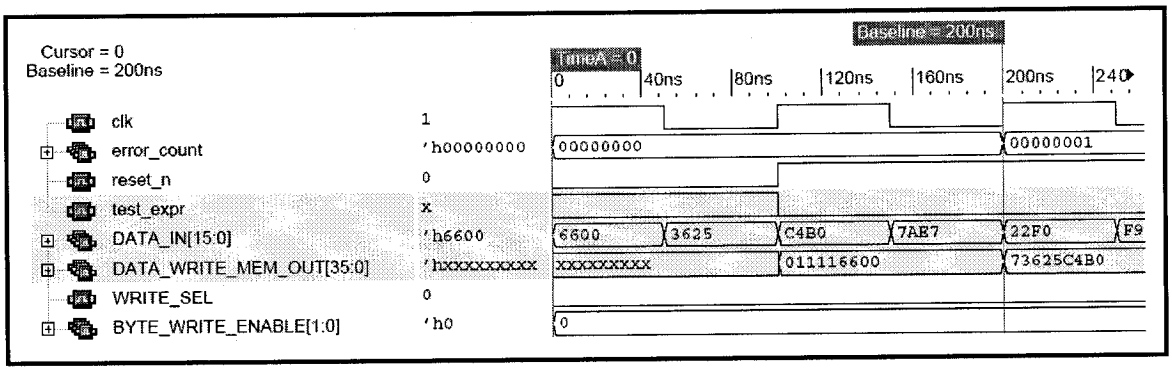

Figure 6.1: Write Data Alignment Error for Pass through Mode

Another critical error, which was captured by OVL assertions, was the *X* detection pattern. During the RTL development the engineer often leaves an unconnected input port to a module, defines a new variable without an assignment, or neglects to drive a signal within a testbench. The *X* detection pattern is useful for identifying and isolating this class of problem. Also, an *X* detection pattern can flag the error of the problems

typically encountered at the chip startup (that is, during the reset process). This may also include some potential, hidden bugs which were not tested before. Fig. 4.12 shows the *X* pattern detection in the memory during the startup phase of our RTL model. This problem occurred as we did not create a memory initialization file in order to initialize the memory used in our datapath.

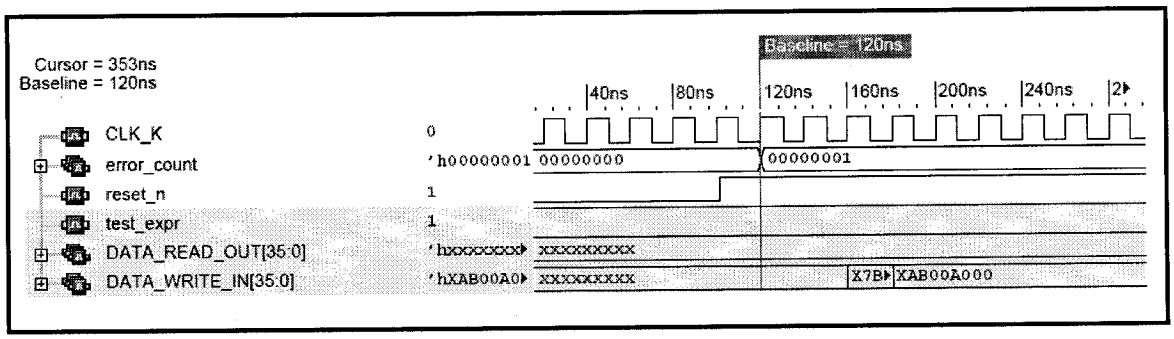

Figure 6.2: Memory 'X' pattern detection

IP (Intellectual Property) consumers often have little or no control over the coding of the RTL they choose to reuse. Detecting *X* or *Z* values on block boundaries can significantly reduce debug during system-level integration of multiple blocks or IP [19]. The remainder of this chapter discusses the counter part of OVL assertions, which is model checking using RuleBase. We reused the same Verification plan from section 4.4 in order for our methodology to be applicable in the same level of abstraction as used by OVL.

Chapter 7: Model Checking of Look-Aside Interface using Rulebase

RuleBase supports a wide variety of design styles and methodologies. While in many cases the user are not required to make special adjustments to existing design methodology, the following design guidelines will further describe the verification process.

7.1: Modeling the Environment and reducing the size of data Model

- **Reducing the Size of the Data Model**

Sometimes testing the data consistency of all the vector's 32 bits at once may not work very well, especially in large models. One technique is to test a single bit instead of the whole vector. So, instead of comparing DI(0..31) with DO(0..31), you can compare DI(0) and DO(0), while setting all other DI input vector elements to a constant, for example, of 0. The above vunit (property keyword used in RuleBase) can be simplified as follows: **vunit** keeping_1bit { **assert forall** xx in **boolean** : **G** (!RST & rose(BTOS_ACK) & DI(0)=xx -> **next_event** (rose(RTOB_ACK)) (DO(0)=xx)) ; In this property BTOS_ACK and RTOB_ACK are internal signals and RST is the reset signal. In addition to, bitwise verification we have to constrain our data model according to the control signals used in the datapath. This constrain definition should be modeled in the environment. For example, the requirement of keeping the input data DI(0..31) stable while the StoB_REQ is active is given in the following :

var DI(0..31) : **boolean**;
assign next(DI(0..31)) :=
    case

!StoB_REQ : nondets(32);

else : DI(0..31);

esac;

The design specification for the Write operation also states that the address for the Write cycle is provided at the following CLK falling edge provided that the WRITE_SEL was asserted low at CLK rising edge. The above scenario was captured using following EDL statements.

#ifdef CORRECT_ADDRESS -- *Generating the correct address*

VAR ADDR_IN(0..26) : boolean;

ASSIGN next (ADDR_IN (0..26)) :=

case -- ***WRITE_CYCLE_INIT_1 flags the next falling edge of CLK***

WRITE_CYCLE_INIT_1: ADDR_IN (0..26);

else: nondets(27);

esac;

#endif

- **Modeling Clocks**

To use formal verification properly, it is essential to understand the way RuleBase deals with clocks, and to choose the proper clock scheme. The environment assumes that the clock signal is generated externally and drives the verified design through input clock pins. The simplest case is a design that only has one clock, in which only one level or edge of the clock is used in the design. In this case, the clock input should be held constant at the value '1':

define CLK := 1; -- CLK is the clock input pin.

RuleBase understands it as the clock being active in every cycle. This works even when some of the flip-flops are gated. The gated flip-flops will work only when the gate is active. The next scheme has one clock, but both levels (and edges) are used in the design. This form of clock definition served our purpose as the master clock(CLK_K) and master clock bar(CLK_K1) used in Look-Aside Interface were 180 degree out of phase. Therefore, we could define only one clock for RuleBase and use both edges for our verification purpose as in this case, the clock can have alternate values 0 and 1, which is shown below:

```
#ifdef CORRECT_CLK
-- Using the clock using both edge
    VAR CLK: boolean ; -- Here CLK is a variable
    ASSIGN   init(CLK) := 0;
             next(CLK) := !CLK;
#endif
```

The LA-1 interface specification is based on separate dual data rate (DDR) buses for data inputs and data outputs. Using DDR interfaces, data is clocked on the rising and falling edges of the clock signals. This effectively doubles the bandwidth of the interface without increasing the clock speed or the bus width. Overall, the LA-1 interface operates at clock speeds between 133 and 200MHz. In order to model dual data rate, the above scheme of using the clock at both edge was applied successfully and the clock was modeled in the Rulebase environment accordingly.

7.2: Coverage Model Development for RuleBase

Though Model Checking using RuleBase addresses the stimulus coverage problem of verification by simulation but it does not solve it completely. Firstly, there is the burning question "Have we coded all the vunits?" Secondly, due to the size problem (state explosion), behavioral partitioning (sub-modules) which adds the following question to the coverage problem "Have we coded enough environments?"

We discuss these two questions in this section. Before proceeding, we would like to emphasize that despite the fact that the second coverage question sounds very similar to the regression suite problem of simulation. While a complete solution in order to address the coverage problems do not yet exist, our methodology regarding vunit and environment writing will enlighten the readers about the implementation of the coverage model of our design.

- **Coverage Model**

The methodology is based on an attempt to obtain block Input-Output relationship coverage. This methodology is described in the following in two steps:

1) The First step is that the RTL Blocks (Top level and the internal blocks) will be fed with every possible legal input sequence. Inputs are defined in the environments.

For example let us analyze the RuleBase Environment of the Write Port. The Interface block diagram of the Write Port is shown in Figure 7.1.

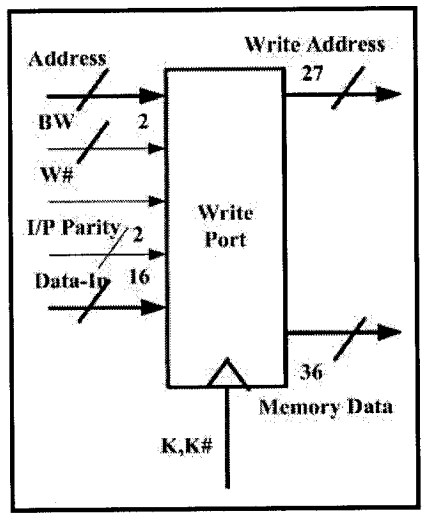

Figure 7.1: Write Port Block Diagram

Now, let us look at the switches defined in the environment in the form of #define are invoked to generate correct Data (Data-In), Address and the Byte Write Enable (BW) signal. Finally, as the Look-Aside Interface runs in Double Data Rate mode, therefore, we have to define the correct form of clock. Note, that all the EDL implementation details of these switches were discussed in the previous section.

```
#define CORRECT_ADDRESS
#define CORRECT_DATA
#define CORRECT_BYTE_WRITE
#define CORRECT_CLK              -- Switch on the default clock both edges are active
```

The above switches invoked the fixed data model for the design.

```
#ifdef CORRECT_DATA    -- switch used to invoke the constrain data model
-- The input data can change only when WRITE_SEL is active low at rising edge CLK
-- and in the falling edge of CLK when WRITE_CYCLE_INIT_1 is active high.
VAR DATA_IN (0..15) : boolean;
ASSIGN      next (DATA_IN (0..15)):=
    case        -- Note WRITE_CYCLE_INIT_1 is an internal signal
```

```
    ! WRITE_SEL | WRITE_CYCLE_INIT_1:DATA_IN (0..15);
    else: nondets(16); -- WRITE_CYCLE_INIT_1 flags the next falling edge of CLK
  esac;
#endif
```

The design specification for the Write operation also states that the address for the Write cycle is provided at the following CLK falling edge provided that the WRITE_SEL was asserted low at CLK rising edge. The above scenario was captured using following EDL statements.

```
#ifdef CORRECT_ADDRESS -- Generating the correct address
    VAR ADDR_IN(0..26) : boolean;
    ASSIGN next (ADDR_IN (0..26)) :=
    case  -- WRITE_CYCLE_INIT_1 flags the next falling edge of CLK
          WRITE_CYCLE_INIT_1: ADDR_IN (0..26);
          else: nondets(27);
esac;
#endif
```

The default verification environment exercised all the possible sequence of CORRECT_ADDRESS, CORRECT_DATA, CORRECT_BYTE_WRITE and CORRECT_CLK (described earlier) along with the WRITE_SEL signal using EDL. The design specification of Look-Aside Interface (LA-1 standard) was the main source of this implementation and this process made sure that the WRITE PORT is being fed with every possible legal input sequence.

2) The second step states that for **every** output signal and selected internal signals, and for **every** clock cycle:

- We have to determine the relationships of the signal to all other signals (inputs, internal signals and outputs) and then write the vunits that check the preservation of these relationships.

For example, the WRITE_SEL_0 which is an internal wire and it gets the value of the WRITE_SEL, when only the ENABLE pins choose the first bank in the Four Bank Implementation of LA-1 Interface. This is implemented using tri-state buffers to block boundaries of each bank. The following vunit checks this scenario:

vunit valid_write_cycle_init_0 { assert "Write cycle should be initialized"

always (ENABLE_PORT (0..1)= 0 & !WRITE_SEL & !WRITE_SEL_0);}

RuleBase does not necessarily indicate that vunits that contain complex signals find design errors more often, and it doesn't mean that they cover all errors either. It is the careful and methodical coverage of all signals that makes RuleBase effective. In the following section, we elaborated some vunits written to serve our model checking purpose. All the properties are enumerated in Appendix C.

7.3: Vunits used for RuleBase Model Checking

We underwent the same procedure as we did for OVL in order to write the vunits of the RTL models. We coded the vunits for the sub-modules (internal blocks) and the top level of a single bank Look-Aside Interface (LA-1 standard). Finally, all the rules were applied on the top level of four-bank Look-Aside Interface. The properties in RuleBase start with keyword vunit and are written in the file named as rules.

7.3.1: Properties for the Write Port

- **vunit valid_write_cycle_init** { assert "Write cycle should be initialized"

  always ((!WRITE_SEL) & rose(CLK) → next (WRITE_CYCLE_INIT_1)); }

This vunit is the same as Assertion 1. This vunit validates the write cycle initialization scenario for the Write Port.

- **vunit valid_data_out_passthrough_msw** { assert forall xx(0..15) in boolean :

  always (BYTE_WRITE_ENABLE(0..1) = 0 & WRITE_CYCLE_INIT_1 &

  DATA_IN(0..15)=xx(0..15) → next next next

  (DATA_WRITE_MEM_OUT(16..31)=xx(0..15))); }

This vunit is exercising Assertion 3 and validating the Passthrough Mode. Note that for model checking using Rulebase we are using one single clock named as CLK (CLK_K in the RTL model, but RuleBase uses CLK as a default CLK name). As we are using both edges of the CLK (CLK_K and CLK_K1), one "next" operator is half a clock cycle.

7.3.2: Properties for the Read Port

- **vunit valid_read_address** { assert "READ ADDRESS CHECK" forall xx(0..26) in boolean : always (!READ_SEL & ADDR_IN(0..26) = xx(0..26) & rose(CLK) → next(READ_ADDR(0..26) = xx(0..26))); }

This vunit is exercising Assertion 8 and validating the correct sampling of the Read Address, when R# (READ_SEL), is asserted low at the rising edge of CLK (CLK_K)

- **vunit valid_data_out_read_port_passthrough_msw_other** {assert forall xx(0..15) in boolean : always (edge_detect & (DATA_IN(16..31) = xx(0..15)) → next next (DATA_OUT(0..15) = xx(0..15))); }

This vunit is exercising Assertion 10 and validating the correct sampling of the Data Out.

Note that though it is supposed to be linear mapping of the OVL assertions to the Sugar 2.0 vunits, but it is not the case. Due to the fact, that Sugar 2.0 is superior in terms of addressing CTL properties efficiently than OVL, it is more expressive in terms of defining invariant properties (static or temporal).

7.4: Model Checking Results and Counter Example

After performing, a rigorous exercise of coding vunits and environments we achieve the following experimental results for all the properties of Single Bank and Four Banks are listed in Table 4.5 and 4.6 respectively. The experiments were performed on 2X UltraSPARC-III+ machine with 2 900 Mhz processors and 4096M of RAM.

Property	Single Bank	CPU Time (in sec.)	Memory (in MB)	No. of BDD's
Property 1	Write Port	40	18	347
Property 2	Write Port	680	52	707692
Property 3	Write Port	737	58	812785
Property 4	Write Port	671	48	595673
Property 5	Write Port	659	45	593443
Property 6	Write Port	761	57	802184
Property 7	Read Port	35	20	326
Property 8	Read Port	38	22	326
Property 9	Read Port	349	42	500131
Property 10	Read Port	878	69	852879

Table 7.1 Model Checking Results of the single Bank LA-1 Interface

By observing Table 7.1, we can see that although RuleBase can check both control logic and datapath, it is more effective for verifying control logic. Datapaths usually have many memory elements, which may increase the size of the internal model representation beyond the capacity of RuleBase. When verifying a design that includes both control and datapath, the datapath is often replaced by an abstract model with fewer memory elements. This abstraction is easier when there is a clear separation between control and data in the design. This observation will be more justified when the four banks of LA-1 was verified. The verification result is shown in Table 7.2.

Property	**Four Banks**	**CPU Time (in sec.)**	**Memory (in MB)**	**No. of BDD's**
Property 1	Write Port	42	27	128
Property 2	Write Port	40	25	125
Property 3	Read Port	39	25	129
Property 4	Read Port	35	28	123
Property 5	Write Port	134728	405	971608
Property 6	Read Port	131870	448	972330
Property 7	Write Port	136880	530	837708
Property 8	Read Port	**State Explosion**		

Table 7.2 Model Checking Results of the Four Banks LA-1 Interface

One notable issue regarding the verification using RuleBase, was manual over riding of the internal signals in the environment. Therefore, one major bug in the Read Port was camouflaged. The reason was one of the inter signal named "edge detect" was manually defined in our environment in order to over ride its actual RTL definition. This

caused an error in the Read Port, as the Data Out from the Read Port was being latched out one clock cycle before it was supposed to be sent. Once this issue was solved, RuleBase was able to pinpoint bugs efficiently than OVL.

Fig. 7.2 gives the counter example of the error found on the Read Port. The property is checks the relationship between the Data In to the Read Port to the Data Out from the Read Port, when the internal signal edge_detect is high. All the properties are listed in Appendix B.

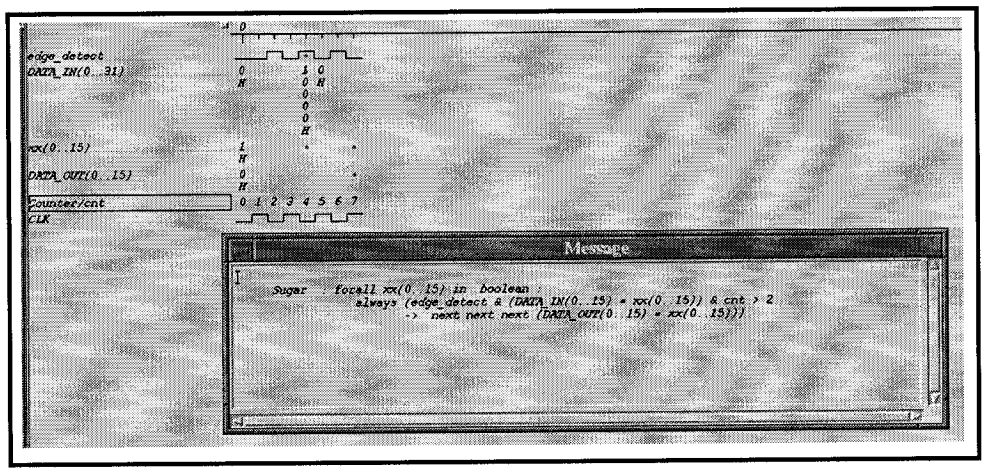

Figure 7.2: Read Port Error Captured By RuleBase

Unfortunately, the version of RuleBase available to us did not solve the state explosion problem of our verification. Therefore, the following suggestions were made:

- The environment should be reduced until rules can reasonably run. We abstracted the size of Data model in order to serve our purpose.
- The number of flip-flop and state variables should not be over several hundreds.
- The environment should be stabilized as a reduced version. Only after the reduced environment version is assured to be correct and stable, we should expand it to further modes and degrees of freedom.

Chapter 8: Conclusion and Future Work

BDD-based symbolic model checking and equivalence checking have proven to be successful formal verification techniques that can be applied to real industrial design. However, since it requires the design to be described at the boolean level, they often fail to verify a large-scale design because of the *state space explosion* problem caused by the large datapath. On the other hand, Assertion-Based Verification can realize a seamless relationship between Model Checking Techniques and Functional Verification. In this thesis we clearly depicted the need for both Assertion Specification and Model-Checking Techniques in the context of a System-On-Chip Verification Flow. The major contributions of this work are listed below:

1. This thesis investigates the Assertion-Based verification of the Look-Aside Interface (LA-1 standard) provided by the Network Processing Forum. We studied the effectiveness of Open Verification Language (OVL) and Property Specification Language (PSL) in verifying a large-scale industrial design interface i.e. Standard Bus Protocol. The Verilog RTL model of the interface, which included Four Banks, was developed by us. The design of Four Banks had 4820 Flip-Flops and 34247 equivalent gates.
2. We suggested an intelligent testcase coverage approach in order to verify large-scale industrial designs. The method is totally based on Verification Plan derived from the Functional Specification i.e. Design Specification. Our hierarchical approach simplifies the maintenance of the testcase regression suite and the resulting verification coverage problem into assertions/properties which can be handled on a

modular basis in the RTL. This approach is applicable on a partially defined design or an internal block (sub-module) instead of waiting for the entire design model.

3. Based on the design specification provided by Network Processing Forum, we also modeled the specification which was in English text into a formal specification i.e. PSL properties written in Sugar 2.0. From a RTL designer point of view this formal specification enables the RTL model to be less pruned to functional error.
4. As the complexity of data operations increases, the default setting used by most formal verification tools may not be sufficient to avoid state space explosion. We applied data abstraction (to verify the Read Port) and environment modeling in RuleBase to handle the complexity of state space in datapath orientated module. The Rulebase engine used our RTL model composed with an EDL (Environment Description Language) which is the target environment in which the design is expected to run. The verification process had been carried out by assertion-based verification as well as model checking.

The future work of this thesis suggests that a combined assertion-based and model checking tool can improve the efficiency of SoC verification in an industrial setting. This combined approach can be widely applicable in verifying a class of designs where the control portion is composed of FSM-based and datapath orientated modules. Because our experimental results showed that RuleBase is more efficient in verifying FSM-based module (Control path) while OVL is more efficient in verifying designs with concrete datapath. This combined tool will open the way to the development of a wide range of new formal verification techniques which will leverage the usage of functional coverage in the context of SoC Verification process.

Assertion Based Verification along with Model Checking open the way to the development of a wide range of SoC verification techniques. The goal of our methodology is to develop an improved re-usable design verification process. To achieve this objective, we need to implement and develop the intelligent testbench strategy and thus enhance the performance and role of the formal verification techniques in a SoC verification scenario.

Bibliography :

1. Thomas L.Anderson, "Using VCS with White-Box Verification Techniques" Synopsys Inc., SNUG San Jose 2000.
2. Synopsys Inc., Assertion-Based Verification, White Paper, Synopsys Inc, March 2003.
3. Network Processing Forum, *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. April 15, 2004.
4. A. Habibi, A. I. Ahmed, O. Ait Mohamed, and S.Tahar. "*On the Design and Verification of the Look-Aside Interface*". In Proc. IEEE/ACM Design Automation and Test in Europe (DATE'05), pages 649–654,Munich, Germany, March 2005.
5. K. Shimizu, D. L. Dill, and A. J. Hu. "*Monitor-based formal specification of PCI*". In *Formal Methods in Computer-Aided Design*, pages 335–353, Austin, Texas, November 2000. Springer-Verlag.
6. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
7. Pankaj Chauhan, Edmund M.Clarke, Yuan Lu and Dong Wang. "*Verifying IP-Core based System-On-Chip Designs*" Carnegie Mellon University, Pittsburgh, PA 15213, April 15, 1999.
8. Accellera Organization. *Accellera property specification language reference manual, version 1.01*. http://www.accellera.org, 2004.
9. A. Ghosh, S. Devadas and A. R. Newton, *"Sequential Logic Testing and Verification"*, Kluwer Academic Publishers, 1992.
10. R.P. Kurshan. *"Formal Verification In a Commercial Setting"*, Bell laboratories, Murry Hill, NJ.In Design Automation Conference. June, 1997.

11. R. K. Brayton et. al, "*VIS: A System for Verification and Synthesis*", Technical Report UCB/ERL M95, Electronics Research Laboratory, University of California, Berkeley, December 1995.

12. IBM Haifa Research Laboratories. *RuleBase Formal Verification Tool (Version 1.5) Users Guide*. May 2003.

13. M. Gordon and T. Melham, *"Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic"*, Cambridge, UK, Cambridge Univ. Press, 1993

14. S. Owre, J.M. Rushby, and N. Shankar, "*PVS: a Prototype Verification System*", In *Proc. International Conference on Automated Deduction,* Saratoga Springs, NY, USA, 1992, pp. 748-752.

15. R. S. Boyer and J. S. Moore, "*A Computational Logic Handbook*", Academic Press, Boston, 1988.

16. M. Kaufmann and J. S. Moore, ACL2, "*An industrial strength version of Nqthm*", *In Proc. 11th Annual Conf. on Computer Assurance,* Gaithersburg, MD, June,1996, S.Faulk and C. Heithayer, Eds. pp. 23-34.

17. Accellera Organization. *Open verification library, assertion monitor reference manual, v 03.06.06*. http://www.accellera.org, June 2003.

18. O.Despaux, *Synthesizable QDR SRAM Interface: Application Note Virtex-II Series XAPP262 (v2.6)*, http://www.xilinx.com, August 29, 2003

19. H.Foster, A.Kronik, D.Lacey "*Assertion-Based Design*" Kluwer Academic Publisher, 2003, pg.165, ISBN 1-4020-7498-0.

20. Open SystemC Initiative. *SystemC 2.0.1 language reference manual*, 2004.

21. Microsoft Corporation. *AsmL for microsoft .net framework (version 2.1.5.7)*, Microsoft. www.research.microsoft.com/foundations/asml, 2004.

22. L.Bening, H.Foster, "*Principles of Verifiable RTL Design*", Second Edition, Kluwer Academic Publisher,2002 ISBN 0-7923-7368-5

23. N.Soni, N.Richardson, L.Huang, S.Rajgopal, G.Vlantis, "*NPSE: A High Performance Network Packet Search Engine*", STMicroelectronics, San Diego, USA.

24. PCI Special Interest Group. *PCI Local Bus Specification Rev 2.2*, Dec. 1998.

25. PCI Special Interest Group. *PCI to PCI Bridge Architecture Specification Rev 1.1*, Dec. 1998.

26. E. Solari, G. Willse. "*PCI Hardware and Software - Architecture and Design*", Annabooks, 1998.

27. H. Moinudeen, A.Habibi and S. Tahar, "*An Executable Specification of the PCI-X Bus Standard in AsmL*", Proc. IEEE Canadian Conference on Electrical & Computer Engineering (CCECE'05), Saskatoon, Saskatchewan, Canada, May 2005.

Appendix A: OVL Monitors for Look-Aside Interface

- **OVL Assertions for WRITE Port**
  - **Assertion 1: assert_always** #(0,0,"Write cycle should be initialized") **valid_write_cycle_init**(CLK_K,write_enable,(WRITE_CYCLE_INIT_1 == 1'b1));
  - **Assertion 2: assert_always** #(0,0,"WRITE_CYCLE_INIT_2 should be high") **valid_write_cycle** (CLK_K1,WRITE_CYCLE_INIT_1,(WRITE_CYCLE_INIT_2 == 1'b1));
  - **Assertion 3: assert_always** #(0,0,"WRITE_DATA PASSTHROUGH LSW") **valid_data_out_passthrough_lsw**(CLK_K,WRITE_CYCLE_INIT_3,(DIN_ NEG-EDGE == DATA_WRITE_MEM_OUT[15:0]));
  - **Assertion 4: assert_always** #(0,0,"WRITE_DATA PASSTHROUGH MSW") **valid_data_out_passthrough_msw**(CLK_K,WRITE_CYCLE_INIT_3,(DIN _POS-EDGE == DATA_WRITE_MEM_OUT[31:16]));
  - **Assertion 5: assert_always** #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 01") **valid_data_out_alignment_lsb_1** (CLK_K,WRITE_CYCLE_INIT_3, (DIN_NEGE-DGE [7:0] == DATA_WRITE_MEM_OUT[7:0]));
  - **Assertion 6: assert_always** #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 01") **valid_data_out_alignment_lsb_2** (CLK_K,WRITE_CYCLE_INIT_3, (DIN_POSE-DGE[7:0] == DATA_WRITE_MEM_OUT[23:16]));

- **Assertion 7: assert_always** #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW =10") **valid_data_memory_out_msb_1** (CLK_K,WRITE_CYCLE_INIT_3, (DIN_NEGE-DGE[15:8] == DATA_WRITE_MEM_OUT[15:8]));
- **Assertion 8: assert_always** #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 10") **valid_data_memory_out_msb_2** (CLK_K,WRITE_CYCLE_INIT_3, (DIN_POSE-DGE[15:8] == DATA_WRITE_MEM_OUT[31:24]));
- **Assertion 9: assert_next** #(0,1,1,0,"Write cycle initialization is checked") **valid_write_cycle_next** (CLK_K, rst_n, ~WRITE_SEL, (WRITE_CYCLE_INIT_1 == 1'b1));

- **OVL Assertions for READ Port**
  - **Assertion 10: assert_always** #(0,0 " Read Address is sampled at the rising edge of CLK_K" **valid_read_address** (CLK_K, read_address_enable, (stored_address == READ_ADDR));
  - **Assertion 11: assert_always** #(0,0," At the rising edge of CLK_K Byte 0 and Byte 1 of the data coming from the memory and entering the read port will be equal to the Data out from the read port") **valid_data_out_k** (CLK_K, READ_CYCLE_INIT_3, (stored_data_K[31:16] == DATA_OUT);

- **OVL Assertions for Memory Port**
  - **Assertion 12: assert_never** #(1,0,"Data In to the memory cannot have X ") **invalid_data_memory_in** (CLK_K, invalid_check_enable, ^DATA_WRITE_IN [35:0] === 1'bX) ;

- **Assertion 13: assert_never** #(1,0,"Data Out from the memory cannot have X") **invalid_data_memory_out** (CLK_K, invalid_check_output_enable, ^DATA_READ_OUT[35:0] === 1'bX);

Appendix B: Sugar Properties for RuleBase

- **Properties for the Write Port**
  - **vunit valid_write_cycle_init** { assert "Write cycle should be initialized"
    always ((!WRITE_SEL) & rose(CLK) → next (WRITE_CYCLE_INIT_1)); }
  - **vunit valid_write_address** { assert forall xx(0..26) in boolean :
    always (WRITE_CYCLE_INIT_2 & ADDR_IN(0..26) = xx(0..26) → next
    (WRITE_ADDR(0..26) = xx(0..26))); }
  - **vunit valid_data_out_passthrough_msw** { assert forall xx(0..15) in boolean :
    always (BYTE_WRITE_ENABLE(0..1) = 0 & WRITE_CYCLE_INIT_1 &
    DATA_IN(0..15)=xx(0..15) → next next next
    (DATA_WRITE_MEM_OUT(16..31)=xx(0..15))); }
  - **vunit valid_data_out_ms_byte** { assert forall xx(0..7) in boolean :
    always (BYTE_WRITE_ENABLE(0..1) = 1 & WRITE_CYCLE_INIT_1 &
    DATA_IN(8..15)=xx(0..7) → next next next
    (DATA_WRITE_MEM_OUT(24..31)=xx(0..7))); }
  - **vunit valid_data_out_ls_byte** { assert forall xx(0..7) in boolean :
    always (BYTE_WRITE_ENABLE(0..1) = 2 & WRITE_CYCLE_INIT_2 &
    DATA_IN(0..7)=xx(0..7) →next(DATA_WRITE_MEM_OUT(0..7)=xx(0..7)));}
  - **vunit valid_data_out_passthrough_lsw** { assert forall xx(0..15) in boolean :
    always (BYTE_WRITE_ENABLE(0..1) = 0 & WRITE_CYCLE_INIT_2 &
    DATA_IN(0..15)=xx(0..15) →
    next (DATA_WRITE_MEM_OUT(0..15)=xx(0..15))); }

- **Properties for the Read Port**
  - **vunit valid_read_sel_edge_detect** {assert "Edge detect select of Read Port" always (READ_SEL →!(edge_detect & edge_detect_retimed)); }
  - **vunit valid_read_sel_edge_detect_xor** {assert "edge_detect != edge_detect_retimed" always ((!READ_SEL) → next(edge_detect ^ edge_detect_retimed)); }
  - **vunit valid_read_address** { assert "READ ADDRESS CHECK" forall xx(0..26) in boolean : always (!READ_SEL & ADDR_IN(0..26) = xx(0..26) & rose(CLK) → next(READ_ADDR(0..26) = xx(0..26))); }
  - **vunit valid_data_out_read_port_passthrough_msw_other** {assert forall xx(0..15) in boolean : always (edge_detect & (DATA_IN(16..31) = xx(0..15)) → next (DATA_OUT(0..15) = xx(0..15))); }
- **Properties for the Four Banks Look-Aside Interface (LA-1 standard)**
  - **vunit valid_write_cycle_init_0** { assert "Write cycle should be initialized" always (ENABLE_PORT (0..1)= 0 & !WRITE_SEL & !WRITE_SEL_0);}
  - **vunit valid_write_cycle_init_1** { assert "Write cycle should be initialized" always (ENABLE_PORT (0..1)= 1 & !WRITE_SEL & !WRITE_SEL_1);}
  - **vunit valid_read_cycle_init_0** { assert "Read cycle should be initialized" always (ENABLE_PORT (0..1) = 0 & !READ_SEL & !READ_SEL_0); }
  - **vunit valid_read_cycle_init_1** { assert "Read cycle should be initialized" always (ENABLE_PORT (0..1)= 1 & !READ_SEL & !READ_SEL_1));}
  - **vunit valid_write_address** { assert forall xx(0..26) in boolean :

always (WRITE_CYCLE_INIT_1 & ADDR_IN(0..26) = xx(0..26) → next (WRITE_ADDR(0..26) = xx(0..26)));}

- **vunit valid_read_address** { assert forall xx(0..26) in boolean : always (ENABLE_PORT (0..1) = 0 & !READ_SEL_0 & ADDR_IN(0..26) xx(0..26) → next (READ_ADDR(0..26) = xx(0..26)));}
- **vunit valid_data_out_passthrough_msw** {assert " Data Pass through" always (prev(prev(BYTE_WRITE_ENABLE(0..1))) = 0 & prev (prev(!WRITE_SEL_0)) → DATA_WRITE_MEM_OUT(20..35) = prev (prev(DATA_IN(0..15)))); }
- **vunit valid_data_out_read_port_passthrough_msw** { assert forall xx(0..15) in boolean : always (ENABLE_PORT (0..1)= 0 & edge_detect & (DATA_READ_OUT(16..31) = xx(0..15)) → next (DATA_OUT(0..15) = xx(0..15))); }

Appendix C: RTL Example with OVL

```
//=============================================================================
// Author : Asif Ahmed
// This module implements the Write Port Operation of LA1 Interface
// Note that the data flow is in one direction From Write Port to Read Port
//=============================================================================

`define                 ADDR_WIDTH    29        // Address Width

//=============================================================================

module   LA1_INTERFACE_WRITE_PORT (
                                        CLK_K,                  // input
                                        CLK_K1,
                                        WRITE_SEL,
                                        ADDR_IN,
                                        ENABLE_PORT,
                                        BYTE_WRITE_ENABLE,
                                        DATA_IN,
                                        DATA_PARITY_IN,
                                        WRITE_CYCLE_INIT_2,
                                        WRITE_ADDR,
                                        DATA_WRITE_MEM_OUT      //output

                                );

//=========================================
//          Interface Declaration
//=========================================

//=============================================================
// Clock inputs for LA-1 interface. Rising Edge Active
// The rising of K is used to latch address and control i/p
// The rising edge of CLK_K and CLK_K1 are used to latch data
// CLK_K_1 is ideall 180 degree out of phase with K
//=============================================================

input           CLK_K;
input           CLK_K1;

//=====================================================================
// Active Low read select input. Note that when Low this input
// causes the address input to be registered and a read cycle to
// be initiated.
// Active Low write select input. Note that when Low this input
// causes the address input to be registered and a write cycle to
// be initiated.
//=====================================================================

input           WRITE_SEL;      // W# in the spec

//=================================================
// Byte Address Pins. For host Devices
// 23 =< ADDR_WIDTH =< 29
// A[0] and A[1] are used as port enable E[1:0]
// Will be used in the top level of the whole interface
```

```
//=================================================

input [`ADDR_WIDTH - 3:0]                    ADDR_IN;


//=================================================
// Active-Low Byte-write inputs. Used to enable or block
// write of a specific byte a write cycle initiated with
// W#. BW0# controls D[7:0] and DP0, while BW1# controls
// D[8:15] and DP1
//=================================================

input [1:0]          BYTE_WRITE_ENABLE;  // BW# in the spec

//=================================================
// Synchronous data inputs. This bus operates in
// response to WRITE_SELECT input
// Synchronous Even Parity Inputs. Correct when XOR
// of DATA_IN[7:0} = DATA_PARITY_IN [0] and XOR of
// DATA_IN[15:8] = DATA_PARITY_IN [1]
//=================================================

input [15:0]                 DATA_IN;
input [1:0]                  DATA_PARITY_IN;

//=================================================
// Synchronous data outputs. Output data is synchronized
// to respective CLK_C and CLK_C_1. This bus operates in
// response to READ_SELECT commands
// Synchronous even parity outputs. Shall be stored and
// checked.Correct when XOR of DATA_OUT[7:0]= DATA_PARITY_OUT[0]
// and XOR of DATA_OUT[15:8]= DATA_PARITY_OUT[1]
//=================================================

output                                       WRITE_CYCLE_INIT_2;          // Control Signal Needed for
the Memory Block
output [`ADDR_WIDTH - 3:0]                   WRITE_ADDR;                  // To the memory
output [35:0]                                DATA_WRITE_MEM_OUT;                  // included parity
bits with the o/p

//=================================================
// Required registers for output
//=================================================

reg [35:0]                                   DATA_WRITE_MEM_OUT, DATA_WRITE_MEM_OUT_1;
reg [`ADDR_WIDTH - 3:0]                      WRITE_ADDR,WRITE_ADDR_K1;

//=================================================
reg                                          WRITE_CYCLE_INIT_1;
reg                                          WRITE_CYCLE_INIT_2;
//=================================================

reg[15:0]                                    DIN_POSEDGE,DIN_NEGEDGE;
reg[1:0]                                     DIN_POSEDGE_PARITY,DIN_NEGEDGE_PARITY;

wire[35:0]                                   D_WRITE_TEMP;

assign               D_WRITE_TEMP = {DIN_POSEDGE_PARITY,DIN_NEGEDGE_ PARITY,DIN_P OSEDGE ,
DIN_NEGEDGE};

//=================================================
```

```
// Data Path Interpretation
//=========================================================
// Definition of read and write cycle and generating the control
// logic for the internal circuitry
//=========================================================

always @(posedge CLK_K)
begin
        if(~WRITE_SEL)
    begin
        WRITE_CYCLE_INIT_1 <= 1'b1;
            case(BYTE_WRITE_ENABLE)
            2'b00       : begin
                                DIN_POSEDGE <= DATA_IN;
                                DIN_POSEDGE_PARITY[0] <= ^DATA_IN[7:0];
                                DIN_POSEDGE_PARITY[1] <= ^DATA_IN[15:8];
                                // Write D[15:0], and Parity
                            end
        2'b01 : begin
                                DIN_POSEDGE <= {8'b0000_0000,DATA_IN[7:0]};
                                DIN_POSEDGE_PARITY[0] <= ^DATA_IN[7:0];
                                DIN_POSEDGE_PARITY[1] <= DIN_POSEDGE_PARITY[1];
                                // Write D[7:0],DP[0]
                            end
        2'b10    : begin
                                DIN_POSEDGE <= {DATA_IN[15:8],8'b0000_0000};
                                DIN_POSEDGE_PARITY[0] <=  DIN_POSEDGE_PARITY[0];
                                DIN_POSEDGE_PARITY[1] <= ^DATA_IN[15:8];
                                // Write D[15:8],DP[1]
                            end
      2'b11       : begin
                                DIN_POSEDGE <= 16'h0000;  // NOP operation
                                DIN_POSEDGE_PARITY[0] <= 1'b0;
                                DIN_POSEDGE_PARITY[1] <= 1'b0;
                            end
        endcase
        end
    else if(WRITE_CYCLE_INIT_2 && WRITE_SEL)            // Logic generated to set
                                                        //WRITE_CYCLE_INIT_1 low
                begin
                        WRITE_CYCLE_INIT_1 <= 1'b0;
                end
end
//=========================================================================
// Negedge operation of a Write Cycle. Note that the write address will be sampled
// at the negative edge of the CLK K or the positive edge of CLK_K1
//
//=========================================================================
always @(posedge CLK_K1)
begin
        if (WRITE_CYCLE_INIT_1 == 1'b1)
    begin
            case(BYTE_WRITE_ENABLE)
            2'b00: begin
                                DIN_NEGEDGE <= DATA_IN;
                                DIN_NEGEDGE_PARITY[0] <= ^DATA_IN[7:0];
                                 DIN_NEGEDGE_PARITY[1] <= ^DATA_IN[15:8];
                                // Write D[15:0], DP[1:0]
                            end
      2'b01       : begin
                                DIN_NEGEDGE <= {8'b0000_0000,DATA_IN[7:0]};
                                DIN_NEGEDGE_PARITY[0] <= ^DATA_IN[7:0];
```

```
                                    DIN_NEGEDGE_PARITY[1] <= DIN_NEGEDGE_PARITY[1];
                                   // Write D[7:0],DP[0]
                                end
        2'b10       : begin
                                   DIN_NEGEDGE <= {DATA_IN[15:8],8'b0000_0000};
                                   DIN_NEGEDGE_PARITY[0] <= DIN_NEGEDGE_PARITY[0];
                                     DIN_NEGEDGE_PARITY[1] <= ^DATA_IN[15:8];
                                       // Write D[15:8],DP[1]
                                end
        2'b11       : begin
                                   DIN_NEGEDGE <= 16'h0000;
                                   DIN_NEGEDGE_PARITY[0] <= 8'b0000_0000;
                                   DIN_NEGEDGE_PARITY[1] <= 8'b0000_0000;
                                  // NOP,
                                end
        endcase

                                WRITE_ADDR_K1 <= ADDR_IN;          // Noting the address into account
                                 WRITE_CYCLE_INIT_2 <= 1'b1;

    end
else
    begin
                    // DATA_WRITE_MEM_OUT_1 <= 32'h0000_0000;

                        WRITE_CYCLE_INIT_2 <= 1'b0;

    end
end                 // always @(posedge CLK_K1)

//=============================================================================================
//
//=============================================================================================

always @(posedge CLK_K)
begin
        DATA_WRITE_MEM_OUT <= D_WRITE_TEMP;  // Latching the 32 bit data at the negative edge
        WRITE_ADDR <= WRITE_ADDR_K1;

end //     always @(posedge CLK_K)

//=====================================================
// Assertion Insertion for Control Signals      Additional glue logic for Assertion
//=====================================================

// synopsys translate_off

wire                    write_select;
reg                     WRITE_CYCLE_INIT_3;
reg                     write_enable;
reg [15:0]              stored_data,stored_data_1;
integer                 count;

initial
begin
                        stored_data = 16'h0;

end

//=====================================================================
assign                  data_count = (count == 4) ? DATA_IN : 16'h0;
assign                  write_select = ~WRITE_SEL;  // Will retime this signal once and use it as enable signal
```

```
assign                  wrte_cycle_init = WRITE_CYCLE_INIT_1;
assign                  enable = (count == 8) ? 1'b1: 1'b0;

//=================================================================

always @(posedge CLK_K) write_enable <= write_select;

//=================================================================
//
//=================================================================

always @(CLK_K)
begin
        count = count + 1;
end

//=================================================================

always @(posedge CLK_K)
begin
        WRITE_CYCLE_INIT_3 <= WRITE_CYCLE_INIT_1;

        if (write_select)
        begin
                stored_data_1 <= DATA_IN;
        end
end



//=================================================================================
// Assertion for the initiation of the Write Port
//=================================================================================

assert_always #(0,0,"Write cycle should be initialized")
valid_write_cycle_init(CLK_K,write_enable,(WRITE_CYCLE_INIT_1 == 1'b1));

assert_always #(0,0,"WRITE_CYCLE_INIT_2 should be high")
valid_write_cycle(CLK_K1,WRITE_CYCLE_INIT_1,(WRITE_CYCLE_INIT_2 == 1'b1));

//=================================================================================
// Assertion Insertion for the relationship between Data In to the Write Port
// and Data output which is going to the memory
//=================================================================================

assert_always #(0,0,"WRITE_DATA PASSTHROUGH LSW")

valid_data_out_passthrough_lsw(CLK_K,WRITE_CYCLE_INIT_3,(DIN_NEGEDGE ==
DATA_WRITE_MEM_OUT[15:0]));

assert_always #(0,0,"WRITE_DATA PASSTHROUGH MSW")

valid_data_out_passthrough_msw(CLK_K,WRITE_CYCLE_INIT_3,(DIN_POSEDGE ==
DATA_WRITE_MEM_OUT[31:16]));

assert_always #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 01")

valid_data_out_alignment_lsb_1(CLK_K,WRITE_CYCLE_INIT_3,(DIN_NEGEDGE[7:0] ==
DATA_WRITE_MEM_OUT[7:0]));

assert_always #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 01")
```

```
valid_data_out_alignment_lsb_2(CLK_K,WRITE_CYCLE_INIT_3,(DIN_POSEDGE[7:0] ==
DATA_WRITE_MEM_OUT[23:16]));


assert_always #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW = 10")

valid_data_memory_out_msb_1(CLK_K,WRITE_CYCLE_INIT_3,(DIN_NEGEDGE[15:8] ==
DATA_WRITE_MEM_OUT[15:8]));

assert_always #(0,0,"WRITE_DATA BYTE ALIGNMENT LSB -> BW =
10")valid_data_memory_out_msb_2(CLK_K,WRITE_CYCLE_INIT_3,(DIN_POSEDGE[15:8] ==
DATA_WRITE_MEM_OUT[31:24]));


assert_next #(0,1,1,0,"Write cycle initialization is checked")
valid_write_cycle_next(CLK_K,rst_n,write_select,(WRITE_CYCLE_INIT_1 == 1'b1));


// synopsys translate_on

endmodule
```