Formal Verification of a Multistage Arbiter

Shahid Ikram[1], Craig Barner[1], John Sweeney[1],
Jim Ellis[1] and Bill Dufresne[2]

[1]Cavium Inc.
Marlboro, MA, USA
www.cavium.com

[2]Synopsys,
Mountain View, CA, USA
www.Synopsys.com

ABSTRACT

We are presenting our work on formal verification of a multi-stage arbiter using the new Synopsys formal tool (VC-Formal). The arbiter implements a priority-based arbitration scheme at the first-level and a round-robin scheme at the second-level. The main concern was to show that the arbiter is starvation free. A testbench was available but could not guarantee absence of starvation. However the testbench was helpful to create a formal verification (FV) environment around the arbiter's RTL. The FV environment tracks the requests, grants as well as outstanding credits. Initial debugging was done using auto-generated properties (AEP). Later on, a rich set of properties for verification and coverage were written and proven. We found a starvation case in the design and proved that the fixed design is bug-free. Complete Functional coverage of the design was validated using formal assertion coverage. VC-Formal uses Verdi as debugging tool that was extremely useful.

Table of Contents

Table of Figures

1. Introduction

Many digital systems use arbitration to share and allocate resources among different subcomponents. A large number of arbitration schemes exists [1] to meet various design requirements each with their own unique verification challenges. The complexity grows quickly and a multistage arbiter is certainly a challenge for exhaustive verification. Generally, exhaustive verification is not possible, for challenging properties like starvation [1] [5] in simulation based dynamic verification (DV). The user must understand the ALL the conditions under which starvation can happen and then find some way to generate the stimulus to test these scenarios. This is nearly an impossible task. Formal verification (FV) tools do provide an opportunity to do exhaustive verification without stimulus. A formal tool attempts to prove a specification, in our case all requests must receive a grant (No Starvation) using assumptions to restrict legal conditions written as SVA properties. That is precisely the case discussed in this paper. Our multistage arbiter was verified both by a UVM based DV environment and secondly, select requirements like "absence of starvation" were validated formally.

Formal verification is a challenging technology to deploy and there are many factors that affect results. This paper discusses our solution to formally validate certain requirements of our multistage arbiter. We will cover many topics which aided us in achieving our goals while mitigating our risk for inaccurate results. The following terminology is utilized in the paper.

Assumption: A FV property that is applied to the design to restrict analysis to legalized states. This can be applied on the top level design pins or internal to the design and is usually write with an SVA assume property. This is also referred to as a constraint.

Constraint: The DV equivalent term used to restrict the testbench generator to legal stimulus. In FV constraint and assumption are interchangeable whereas in DV constraints apply to random variables.

Falsify: A term used in Formal Verification that states your design does not meet the design specification or property.

Proven: A term used in FV that signifies that your design meets your property specification.

Property: A design specification written as a SVA property.

2. The Design

A block diagram of our design is depicted in Figure 1. The arbiter provides access to a common bus for multiple groups of sources. Each source serially shifts a 10-bit request to access the bus. The format of the 10-bit request is as follows, where the most significant bit is shifted first:

request [9]   = Valid
request[8]   = Priority (1=high,0=low)
request[7:0] = Cycles needed

The arbiter grants high priority requests over low priority requests.   The source will receive a grant for each cycle requested and each source may only have a single request outstanding. Within a group of sources, the arbiter will not select a new source until all cycles requested by current source have been granted.   The arbiter will interleave grants to sources of different groups.   The arbiter uses a credit/debit scheme to prevent the sources of a group from overrunning the per-group storage at the bus destination.

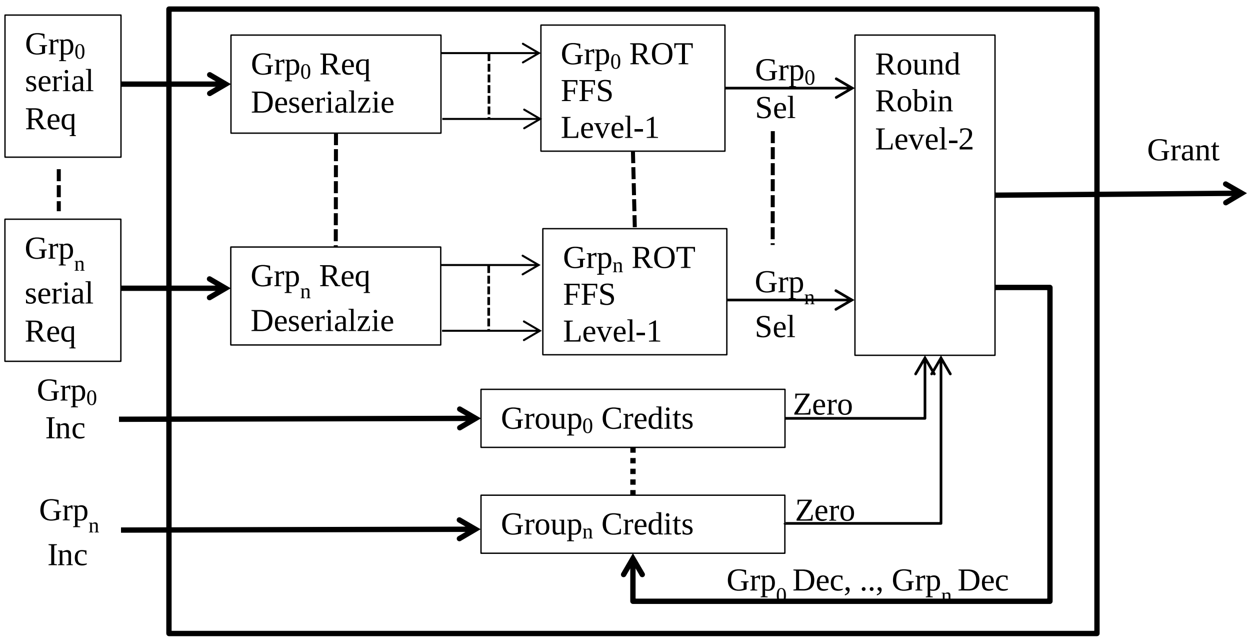


Figure 1: Arbiter

The arbitration scheme is two levels.   The first level arbitrates within a group of sources while the second level arbitrates among the groups. The second level generates the grants to the sources.

First level of arbitration

The first level arbitrates each group independently by selecting a source using rotating find-first-set.   The selection considers low priority request only in the absence of high priority requests.   Note that the selection does not require consideration of per-group credits. A new source from with a group may only be selected once the current source has received grants for all requested cycles. Therefore, the first level selection sticks until completed. The selected source from each group participates in the second level of arbitration.

Second level of arbitration

The second level arbitrates the sources from the first level in a round-robin manner. When a group runs out of credits, the corresponding source selected by the first level of arbitration is ignored.

3. The Verification Challenge

Arbiters are mostly part of the control paths of digital systems [1]. Any shared resource e.g. a bus, a buffer or a channel, needs an arbiter to assign access to different requesters. Naturally, we want an arbiter to be fair i.e. to provide equal service to the different requesters but the definition of "equal" is loaded in this case. There are arbitration schemes that are intentionally unfair with higher priority given to certain requesters but even in that case we don't want to starve low-priority requesters. William [1] etc. identifies three definitions of fairness:

- *Weak fairness*: Every request is eventually served.
- *Strong fairness*: Every requester will be served equally often.
- *FIFO fairness*: Requesters are served in the order they made their request.

A multistage arbitration scheme provides a further challenge. An arbiter may be fair locally but the system employing multiple arbiters may not be fair as a whole. Please see William [1] chapter 18 for an example case. Our multi-stage arbiter includes two arbitration schemes. The first one is priority based and second one is round-robin. There are only two-priority levels and for each level, the requests are round-robin. The low-level priority round-robin only considered when there is no request at the high-level round-robin. The round-robin scheme follows strong fairness but may not follow FIFO fairness.

Here is what we are trying to prove:

"As long as the system is eventually returning credits and making requests infinitely, our multi-stage arbiter issues grants for same priority requests with strong fairness."

4. Formal Verification

Formal verification is a type of verification. It uses mathematical proofs to validate a design meets its specification. To prove any property of a design under test (DUT), there are several key features that affect the result accuracy. First, the design must be synthesizable (normally not a problem). Second the formal tool requires initial state values (reset), legal forms of stimulus (also called assumptions) and properties that capture the intended design behaviour in a succinct format. Input assumptions and property assertions were written in the System Verilog Assertion language [8].

There are several limitations a formal tool faces, namely memory capacity and the depth of the property proof. The first relates to design size and the second to runtime. Capacity was not an issue for our design, and we will discuss runtime later when we discuss the liveness property.

4.1 VC-Formal Tool Preparation

There many steps in preparing the tool for execution. Two inportant ones is defining the design clocks and generating the initial or reset state. This intial state respresents the value held in each state element after the design is reset. There are two ways in which VC-Formal can load the initial state. One, by simulating the design, given the clock and reset information, to a user defined point or by loading the initial state value from a simulation dump file such as Verdi's verilog.fsdb. In our example, we chose to use the first method.

The design and setup entry in VC-Formal is TCL driven. Using these TCL commands, the user must provide this following design information

Note: VC-Formal provides a strutural analysis tool to infer the clocks and reset automatically if this is desired.

1. *Design Clocks*: Clock specification is essential. The tool has a concept of a reference clock for script properties, reset sequence simulation and other tasks. If the design contains a single clock then this becomes the reference clock. While designs that contain mutliple clocks, the fastest user-defined clock is chosen as the reference clock. I encourage the reader to review [6] Section 2.2.6 for detailed information on clocks. Our design is synchornous so the clock definitions were fairly straight forward. Below is the muliple clock definition for our design:

```
create_clock clk0 –period 200 -initial 0
create_clock clk1 –period 300 –initial 1
```

2. *Reset Signals*: Formal anaylsis starts from the initial or reset state. Without this your proofs are not valid because they are proven on the unknown state of your device. Below are the commands we used to define the design reset pins.

```
create_reset rst1 -high
create_reset rst2 -low
```

3. *Constants during reset simulation*: These are assumptions applied to the inputs that work in concert with reset. Example cases are credit returns or new request generation, that will be deasserted during reset analysis but will have constrained random values during formal analysis.

4. *Constants during formal analysis*: These are sets of signals that are held at a constant value during reset simulation phase as well as during formal analysis phase. They are asserted using *set_constants* command. Some examples in our design were scan, bist, fuse chain, etc, signals.

5. *Set of abstractions*: Sometimes, the design contains design constructs that describe complex behavior which are not needed for formal analysis. The user using the set_abstraction command can replace them with the desired abstration. There are some automatic abstraction performed by VC-Formal like on memory structures.

6. *Generating and Saving the initial State:* The following commands invokes the simulation and saved off the inital state values.

```
sim_run –stable
sim_save_reset
```

4.2 Constructing the Environment

Constructing a proper environment around the DUT is an equally important part of formal analysis. A user can very easily define an environment that returns incorrect results. Two environmental conditions I will cover in this paper are over-constraining and under-constraining.

4.2.1 Avoid over-constraining

There are multiple ways to over-constrain a design environment. We shall point out the two most common:

1. Creating a set of assumptions that are contradictory to each other.
2. Constraining the inputs in a way that makes parts of the design unreachable.

In FV, we build a mathematical model that is logically validated. If we have contradictory assumptions, properties will falsely be proven. We avoided this issue by inserting faulty design properties in the environment that should always falsify. These faulty properties raise an alarm if they are proven true.

The second issue is by over-constraining inputs, which could cause parts of design to become unreachable. Again, it has the potential to falsely prove a property of the design. We avoided this issue by:

- Developing an extensive list of cover properties for different features of the design.
- Used auto generated structural cover properties.

The first set of properties was manually created and maintained by the user while in the second case, VC-Formal provides built-in tools to auto-generate properties for structural (code) coverage like line, fsm, condition and toggle. After each new addition of assumptions or constraints, we confirmed that we did not over-constrain the DUT by generating line coverage through reachability analysis.

Using this conservative approach, we iteratively added assumptions to the model, while all the time using the above metrics to show that we are not over constraining the design.

4.2.2 Generating legal transactions (Assumptions and Constraints)

Formal tools do not use dynamic stimulus as a stimuli, but in order for the proofs to be valid the user must define the legal input state.

The Design environment needed to provide:

1. Randomly ordered requests for all groups with random number of credits.
2. Only one pending request per agent at any time.
3. Consume the grants and granted credits, and returning then granted credits after a random delay.
4. Properly format requests, that is, serialized requests with 10 bits information as described in Section 2.

We created a generic synthesizable module to capture all these requirements and an instance of it for each of the agents. Figure 2 partially shows this module.

```
module req_tran (input logic clk, rst_n, req_gen, prior,  gnt, [7:0] random_req, output logic
req,cred_outst,eor);
  reg eor; //end of request
  reg [3:0] cnt; //To serialize random_requst.
  reg [7:0] req_issued; //Random request captured and issued.
  reg        req_pending;// Indicates an outstanding request, means new request cannot be
issued at this time.
  reg [13:0] credits;//2**req_issued.
  always @(posedge clk)
    begin
          if ((cnt == 0) && req_gen && ~req_pending  && (random_req != 0))
            begin
         cnt <= 1;
         req <= 1;//the valid bit for the request.
         req_pending <= 1;
         req_issued <= random_req; //capture the random request, then transmit it serially.
         credits <= 0;
            end
```

Figure 2: Transaction Transformer

The module req_tran is very similar to a constrained-random environment in DV. It transforms a randomly captured transaction into a serial format acceptable to the arbiter, feeds it to the arbiter and manages credits issued. It also makes sure that there is only one pending request and waits for the grant to appear before enabling another transaction issue. Once a grant is issued, the module assumes requested credits are issued. After a random delay, the module acts as if it has got the credits back and is ready to issue another request.

A new transaction is initiated when the signal req_gen, a free input, is asserted and there is no pending request. The module captures the priority bit and the random_req values at that time. The priority bit is a free-variable that can be picked 0 or 1 by the formal analysis tool and therefore act as a random choice for the priority. The 8-bit random_req signal is also a free variable and picked randomly by the formal analysis tool to model the credits for the request being initiated. The module transforms this 10-bits packet into a serial format and feeds the arbiter.

The Figure 3 shows a screen capture of different randomly generated transactions for a witness run. A witness run provides a trace satisfying a given property. Agent ae_0 generated a low priority request (emu__arb_ae_request [0]) and got the first grant (ctl_ae_gnt[0]) shown in the last line of the waveform. The waveform also shows other requests, the design clock (sclk) and the reset(srst_n) as well.

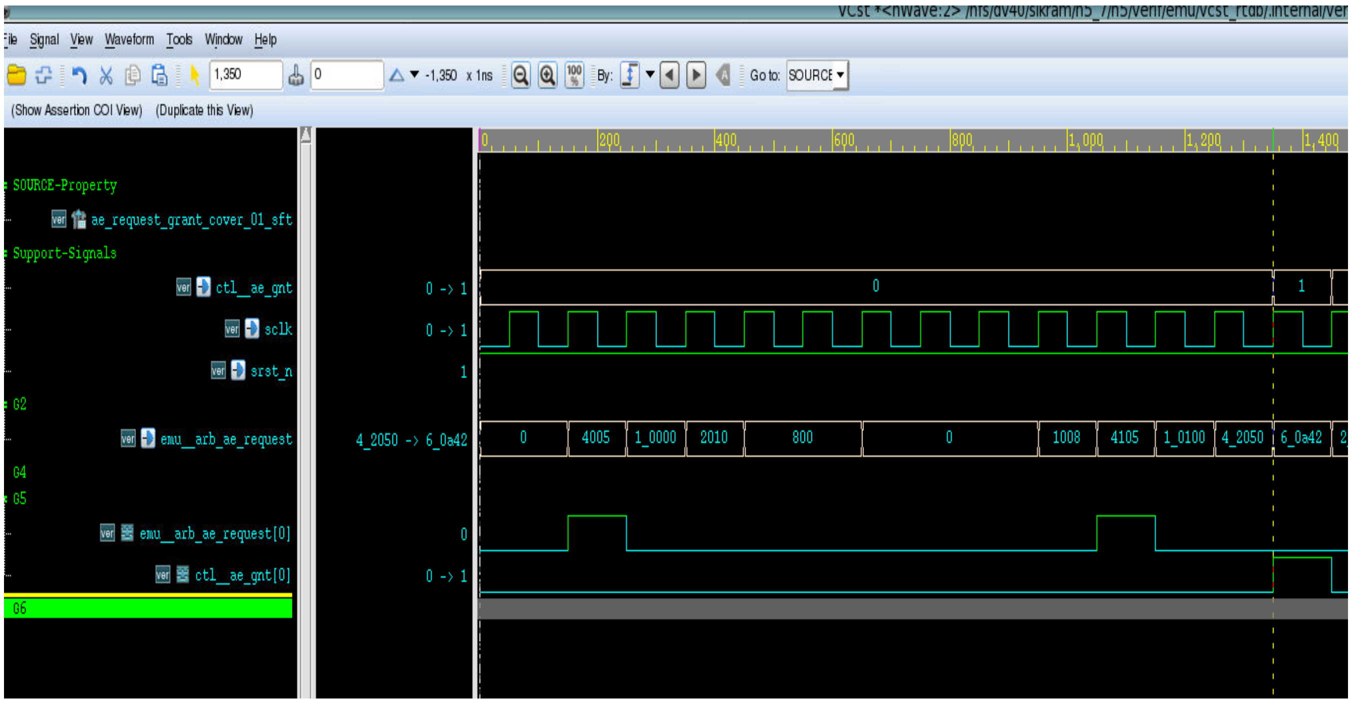
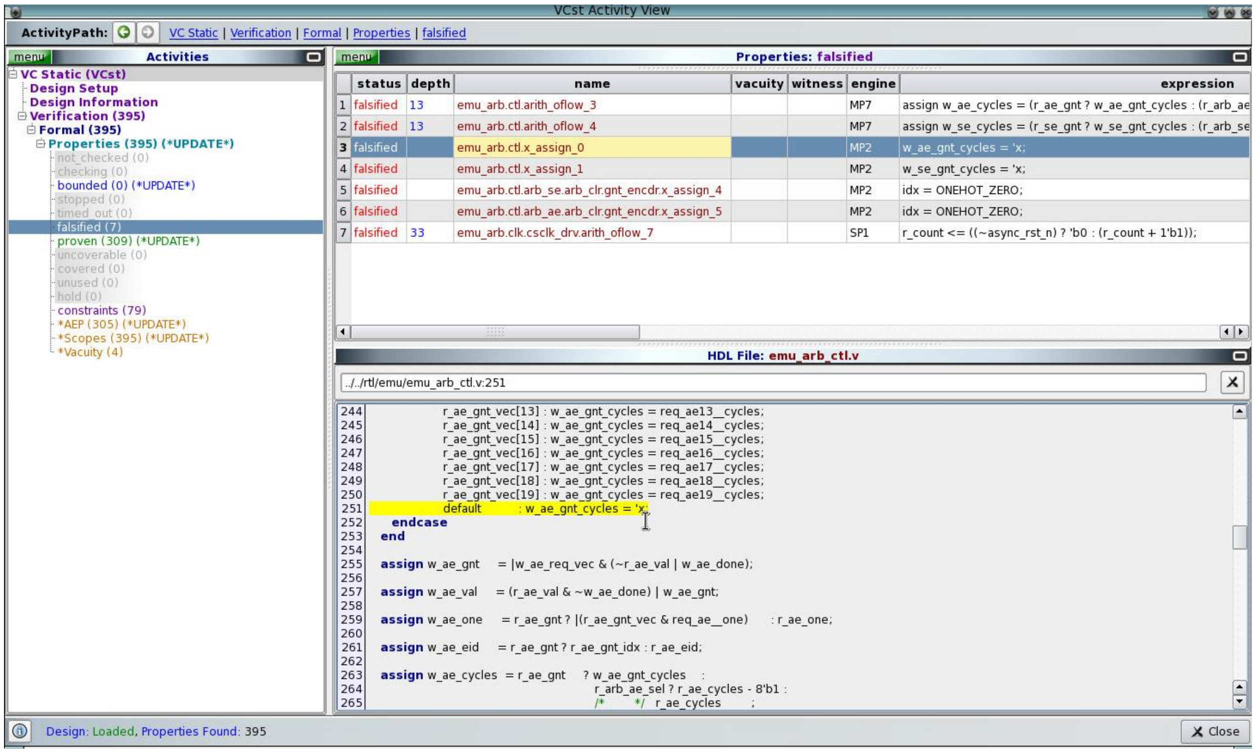

File Signal View Waveform Tools Window Help
1,350
0
-1,350 x 1ns
By:
Go to: SOURCE
(Show Assertion COI View) (Duplicate this View)
SOURCE-Property
ae_request_grant_cover_01_sft
Support-Signals
ctl_ae_gnt 0 -> 1
sclk 0 -> 1
srst_n 1
G2
emu_arb_ae_request 4_2050 -> 6_0a42
G4
G5
emu_arb_ae_request[0] 0
ctl_ae_gnt[0] 0 -> 1
G6
0 200 400 600 800 1,000 1,200 1,400
0 4005 1_0000 2010 800 0 1008 4105 1_0100 4_2050 6_0a42

Figure 3: Transactions

4.3 Auto Extracted Properties (AEP)

These are many properties which can be auto generated through structural analysis of the design. The properties list includes array boundaries checks, arithmetic overflow, full/priority case checks etc. We used these properties regularly to confirm the sanity of our verification environment as well as basic validation of the RTL. There were a total 305 generated properties, out of which 298 were proven. Figure 4 shows a screen capture for this.

Seven of properties were falsified. After reviewing the traces and discussions with the RTL designer we found that they were known issues and can be ignored for our work's objective. An example failure case is highlighted in Figure 4. AEP generated property "x_assign_0" failed for line 251 in emu_arb_ctl.v. AEP generates a check for every assignment in the RTL where an explicit logic X assignment is made. The property will fail if the assignment is ever activated. The failure of "x_assign_0" identifies an assignment of "x" to the signal w_ae_gnt_cycles right out of the reset. This was not an issue as no grant is possible right out of reset and hence it was ignored. Another point to notice is that there are two properties for each x_assign for the same line, i.e., x_assign_0 and x_assign_1. Formal tool cannot handle x-assignment. So instead, it validates the assignment of both 0 and 1.


VCst Activity View
ActivityPath: VC Static | Verification | Formal | Properties | falsified
menu
Activities
menu
Properties: falsified
HDL File: emu_arb_ctl.v
../../rtl/emu/emu_arb_ctl.v:251
Close

VC Static (VCst)
- Design Setup
- Design Information
- Verification (395)
  - Formal (395)
    - Properties (395) (*UPDATE*)
      - not_checked (0)
      - checking (0)
      - bounded (0) (*UPDATE*)
      - stopped (0)
      - timed_out (0)
      - falsified (7)
      - proven (309) (*UPDATE*)
      - uncoverable (0)
      - covered (0)
      - unused (0)
      - hold (0)
    - constraints (79)
    - *AEP (305) (*UPDATE*)
    - *Scopes (395) (*UPDATE*)
    - *Vacuity (4)

	status	depth	name	vacuity	witness	engine	expression
1	falsified	13	emu_arb.ctl.arith_oflow_3			MP7	assign w_ae_cycles = (r_ae_gnt ? w_ae_gnt_cycles : (r_arb_ae
2	falsified	13	emu_arb.ctl.arith_oflow_4			MP7	assign w_se_cycles = (r_se_gnt ? w_se_gnt_cycles : (r_arb_se
3	falsified		emu_arb.ctl.x_assign_0			MP2	w_ae_gnt_cycles = 'x;
4	falsified		emu_arb.ctl.x_assign_1			MP2	w_se_gnt_cycles = 'x;
5	falsified		emu_arb.ctl.arb_se.arb_clr.gnt_encdr.x_assign_4			MP2	idx = ONEHOT_ZERO;
6	falsified		emu_arb.ctl.arb_ae.arb_clr.gnt_encdr.x_assign_5			MP2	idx = ONEHOT_ZERO;
7	falsified	33	emu_arb.clk.csclk_drv.arith_oflow_7			SP1	r_count <= ((~async_rst_n) ? 'b0 : (r_count + 1'b1));

```
                r_ae_gnt_vec[13] : w_ae_gnt_cycles = req_ae13__cycles;
                r_ae_gnt_vec[14] : w_ae_gnt_cycles = req_ae14__cycles;
                r_ae_gnt_vec[15] : w_ae_gnt_cycles = req_ae15__cycles;
                r_ae_gnt_vec[16] : w_ae_gnt_cycles = req_ae16__cycles;
                r_ae_gnt_vec[17] : w_ae_gnt_cycles = req_ae17__cycles;
                r_ae_gnt_vec[18] : w_ae_gnt_cycles = req_ae18__cycles;
                r_ae_gnt_vec[19] : w_ae_gnt_cycles = req_ae19__cycles;
                default          : w_ae_gnt_cycles = 'x;
      endcase
   end

   assign w_ae_gnt     = |w_ae_req_vec & (~r_ae_val | w_ae_done);

   assign w_ae_val     = (r_ae_val & ~w_ae_done) | w_ae_gnt;

   assign w_ae_one     = r_ae_gnt ? |(r_ae_gnt_vec & req_ae__one)     : r_ae_one;

   assign w_ae_eid     = r_ae_gnt ? r_ae_gnt_idx : r_ae_eid;

   assign w_ae_cycles  = r_ae_gnt    ? w_ae_gnt_cycles    :
                                       r_arb_ae_sel ? r_ae_cycles - 8'b1 :
                                       /*      */  r_ae_cycles       ;
```

Design: Loaded, Properties Found: 395

Figure 4: AEP run

4.4 Arbiter Properties

A wide range of arbitration schemes exist [1]. Some of these are fair and others are priority based or a combination of these. Harry Foster etc. [2] [3] provides an extensive list of properties for Arbiters and other digital components. We provide here a list of the properties we used to verify our arbiter. All the properties are generic in the sense that they were proved for all the groups. There are few properties that are not listed here e.g. credit-check etc.

4.4.1 Exclusive Grants

Arbiters are generally used to provide mutual exclusion and this is one of their basic properties, no two grants can happen at the same time. This one was easy to capture and prove as the arbiter maintains a grant-vector for each group and the following property captured the intended behaviour:

```
property mutex_grant_p(gnt);
  @(posedge clk) disable iff (reset)
          $onehot0(gnt);
endproperty
```

4.4.2 Minimum Latency

Generally, there is a design requirement, that there will be a minimum delay between request and grant. In our case, that delay is 1 clock cycle. In other words, this means that the grant and request cannot occur in same cycle. The corresponding property captured the intended behaviour:

```
property request_grant_latency_property(req,gnt,i);
       @(posedge clk)  disable iff (reset )
               !( req[i]&& gnt[i]) ;
endproperty
```

4.4.3 Fairness

The arbitration scheme described in Section 2 is required to be fair assuming same priority. For example, let us assume agent0 and agent1 have pending requests both with same priority and agent1 gets a grant first. It will be unfair for another request from agent1 with same priority to get a grant before agent0's pending request receives a grant. The corresponding property captures the intended behaviour

```
  property fairness_property(req0,pri0,gnt0,req1,pri1,gnt1);
   @(posedge clk) disable iff (reset)
        req0 && req1 && ( pri0 == pri1) |=>
                (gnt1 |=> ~gnt1 until_with  gnt0);
endproperty
```

The operator "until_with" is defined in [8].

4.4.4 Grant without Request

Generating a grant without a request will be disastrous for any system. An arbiter should never issue a grant to a requester unless a request was made in the past. A detailed explanation of this property can be found in Harry [3].

```
  property no_grant_without_request_property(req,gnt);
   @(posedge clk) disable iff (reset)
      Gnt |=>  (~gnt throughout (request) [->1]);
endproperty
```

4.4.5 Priority Service

We have two-levels of priority and the second bit in our request transaction identifies that. We need an assurance that if a high priority request arrives at the same time as a low priority request, then high-priority request will be serviced first.

```
    property high_priority_property(req0,pri0,gnt0,req1,pri1,gnt1);
     @(posedge clk) disable iff (reset)
          req0 && req1&& !pri0 && pri1 |=>
                           ~gnt0 until gnt1;
  endproperty
```

4.4.6 Starvation and Liveness

Starvation in a concurrent environment is perpetual denial of resources to an agent and opposite of liveness. Liveness property guarantees that a good thing (like grant in our case) will eventually happen. Starvation is similar to deadlock property in the sense that an agent is stuck probably waiting forever. However proving a system is starvation-free is a stronger requirement than proving a system is dead-lock free as a system may be deadlock-free but not starvation-free[1][5]. A liveness property (i.e. anti-starvation property) is generally hard to prove without fairness i.e. we can only prove fairness of the arbiter if the system encompassing it is fair i.e.

1. All the issued credits are returned.
2. Every requester generates non-zero credits requests infinitely often.

We guaranteed the first through our environment construction and wrote the following assumption for the second one.

```
property fairness_assumption;
(@(posedge clk) disable iff (reset)
s_eventually
((start_of_req && (credits!=0 )) != 0 ));
endproperty
```

“s_eventually” is a SVA keyword. An implicit “always” before s_eventually asserts that following statement will be true for all times and “s_eventually” implies a truth sometime in future [8]. The liveness property i.e. anti-starvation property comes out like shown below:

```
  property request_grant_liveness_property(req,gnt);
    @(posedge clk) disable iff (reset)
     req  |=> s_eventually grant;
 endproperty
```

When we formally verify a property, there are three possible outcomes. Either the property is falsified, is proven or no result. This property neither failed nor passed after running for a couple of weeks. We forced the number of credits to 1 through addition of another assumption but the property still did not converge. Finally we decided to go with bounded liveness that is described below.

4.4.7 Bounded liveness

A bounded liveness [7] establishes that once a request is made and enough transitions occur (where "enough" is determined by some bound), a grant will happen. We are dealing with finite-state systems and if a finite-state system satisfies a liveness property then it satisfies a bounded liveness property for a suitable bound as well. The advantage of using bounded liveness is that it is a safety property and can be proven in relatively shorter time. It is also a good second option when unbounded liveness cannot be proven in a reasonable amount of time. The bound required for the arbiter was nearly 1000 that again was too big to finish in a reasonable time. We restricted number of credits that can be requested to 1 and tried the following property.

```
property  request_grant_bounded_liveness_property(req,gnt);
    @(posedge clk) disable iff (reset)
     req  |-> ##eventually[1:100] grant;
 endproperty
```

The property failed in a couple of hours and after debugging we were able to find a starvation case. The case will be described later in the results section. After a fix was put in the design, we reran the property and after a couple of days it was proven correct, hence showing that the fix is working.

5. When are we done?

There is more than one way answer this question. We are listing here two methods.

5.1 Assertion Coverage

VC-Formal provides a tool to perform static analysis of the design to evaluate the coverage of assertion and cover properties. It provides metrics to indicate if the design is or not sufficiently verified by assertions. The coverage analysis looks at the registers. A report generated for our design, assertions and cover properties is shown below:

```
analyze_fv_coverage report summary:
-----------------------------------
  Number of instances analyzed: 213
  Number of registers analyzed: 542
  Overall coverage score:      89/100
```

Figure 5: FV Assertion Coverage

5.2 Assertions Completeness

How to verify a verifier? A well-known method to that end is by injecting errors in the design and checking if the verifier can catch it. This method is different from the faulty assertions used to ensure our design environment is not over-constrained. In that method, we have used faulty assertions that should always fail. Certitude provides an automated method to do this kind of analysis. We were keen to try Certitude with VC-Formal but at the time of this writing it was not available. However, we manually tested our verification environment by inserting bugs at different points in design. The example cases were reversing the priority, stuck-at faults etc. The process was not comprehensive as it could be in case of Certitude but gave us some measure of quality of our verification setup.

6. Results

Formal verification found a starvation case in the arbiter. As mentiond in above, we have a priority-scheme in the arbiter compounded with a round-robin scheme. If the arbiter gets a low-priority request in the mix of high-priority requests, its‘ round-robin turn has to wait until all the high-priority requests are serviced. This potentially leads to a starvation of this low priority request. The design did not have a timer to elevate this starving request.

The reason we never found this during DV because of the way this arbiter being used. Generally microcode only uses high priority for certain small exception transactions and they are very rare. Therefore, there was never enough high priority traffic to generate this case. However this isn't a hard rule and microcode could be written differently. Formal verification anti-starvation property identified this case by generating non-stop high priority requests and hence starving a particular agent. The designer updated the design with a timer and arbiration is not causing a starvation anymore. The timer is a great enhancement to guard against a change in the use model of high-priority, avoiding silicon debug.

7. Conclusions

We presented our work on formal verification of a multistage arbiter. We consider FV as a complementary technique to DV, i.e. it is used to increase our confidence in the verification of our design. For most practical cases, we cannot prove complete correctness due to the limitations discussed in this paper. However, each additional piece of verification helps. We have presented a complete method to verify any design, starting with initial setup for reset state, constructing the verification environment to the actual properties used to verify the design. Finally property coverage and errors insertion gave us a measure of completeness. The exercise increased our confidence in our design and showed us the usefulness of Formal Verification as well.

8. References

[1] Dally, William J. etc., “Principles and Practices of interconnection Networks,” Morgan Kaufmann 2004, ISBN: 0-12-200751-4

[2] Foster, Harry D. etc., “Assertion-Based Design,” Second Edition, Kluwer Academic Publishers 2004, ISBN: 1-4020-8027-1

[3] Foster, Harry D. etc., “Creating Assertion-Based IP,” Springer 2007, ISBN: 0387366415

[4] Cerny, Eduard etc., “ SVA: The Power of Assertions in System Verilog,” Second Edition, Springer, 2014, ISBN: 3319071386

[5] Lamport, Leslie, “ Specifying Systems,” Addison-Wesley, 2003, ISBN:0-321-14306-X

[6] Synopsis, Inc., “*VC Formal Verification User Guide*,” Synopsys, Inc., Mountain View, CA, March 2015

[7] Y. Fang, K. L. McMillan, A. Pnueli, and L. D. Zuck. Liveness by Invisible Invariants. In FORTE, volume 4229 of LNCS, pages 356–371, 2006.

[8] IEEE Standard 1800-2012.

[9] C. Yan, M. R. Greenstreet, and J. Eisinger, “Formal verification of arbiters,” The 16th IEEE International Symposium on Asynchronous Circuits and Systems, May 2010.

[10] C. Yan and M. R. Greenstreet, “Verifying an arbiter circuit,” in FMCAD. Nov. 2008, pp. 1–9.