2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Efficient Verification of Arbitration Design with a Generic Model

Kevin Kotadiya, Ishita Agrawal
eInfochips An Arrow Company
Ahmedabad, Gujarat, India

***Abstract*- This paper proposes a generic System Verilog based arbitration verification model that can be integrated N number of times into any verification environment and highlights the challenges associated with verifying multi-level arbitration. The model is organized, reusable, easy to maintain, reduces code repetition, and human coding mistakes, which saves time and effort. It facilitates verification of intermediate arbiter(s) without white box testing and enables easy bug detection. The approach ensures all input combinations exercised for comprehensive testing via inbuild functional coverage. Overall, the arbitration model proposed in this article offers a comprehensive solution to the challenges associated with multi-level arbitration in complex systems.**

***Keywords*- WRR Arbitration Model, Re-usable, Protocol Independent, Coverage, Channel Information**

I. INTRODUCTION

In modern digital systems, the efficient and fair resolution of conflicts among multiple sources attempting to access shared resources is crucial. The arbitration mechanism plays a vital role in ensuring that conflicts are resolved in a timely and equitable manner. Verifying the correctness and effectiveness of arbitration designs is a complex task due to the presence of multi-level arbitration, where multiple layers of arbitration modules interact to make decisions.

This paper proposes a generic System Verilog based arbitration verification model that addresses the challenges associated with verifying multi-level arbitration designs. The model offers a comprehensive solution that can be seamlessly integrated into any verification environment, reducing code repetition, improving maintenance, enabling easy expansion, and facilitating efficient bug detection.

The motivation behind this research stems from the need to address the difficulties faced by verification teams in validating complex multi-level arbitration designs. Traditional verification methodologies often involve laborious and error-prone manual coding, resulting in time-consuming processes and the potential for human errors. Additionally, the lack of comprehensive coverage analysis hinders the identification of untested scenarios and the detection of arbitration misbehaviors.

The primary objective of this paper is to present a generic arbitration verification model that streamlines the verification process, enhances test coverage, and enables efficient bug detection. By providing a reusable and adaptable model, verification teams can overcome the limitations of traditional verification approaches and significantly improve the effectiveness and efficiency of arbitration design validation.

II. ARIBITRATION MODEL ARCHITECTURE

The chapter on the architecture of the arbitration model provides an overview of its design and functionality. This section describes how the model handles the N number of input channels and maps them according to the design's default order of channel selection for arbitration, as depicted in "Fig. 1," Additionally, it explains the configuration of weightage for each channel using the "set_weight" API.
To feed input to the arbitration model, the "push_packet" API allows users to provide custom user information in the form of a System Verilog data type, which is then pushed into the dedicated queue associated with the respective channel.

The arbitration process is facilitated through two approaches.

a. The first approach involves using the "get_expected_packet" API to fetch the arbiter result immediately. When this API is called, the core logic of the arbiter performs the arbitration, producing the expected result.
b. The second approach incorporates built-in result checking. Users can push the actual channel packet ID into the arbiter model using the "check_actual_packet" API. The arbitration process generates the expected result, which is then compared with the actual result obtained from the design.

The arbitration model also addresses multiple failures. i.e., in cases where one arbitration result differs from the actual result of the design, subsequent arbitration results are likely to be affected. To prevent such cascading failures, the model updates the weightage counter based on the actual result instead of the expected result. There is a configuration provided to disable it. On top of it, the user can disable the entire built-in checking of the actual result.

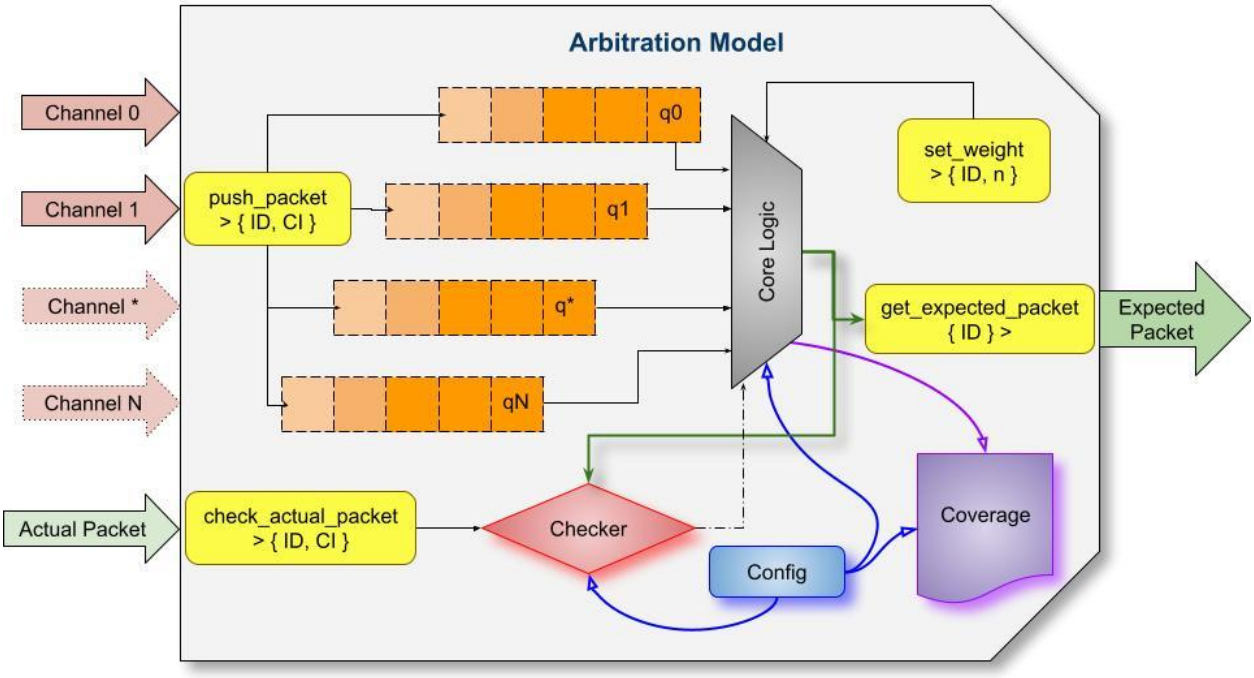

Figure 1. Arbitration Model Architecture

Furthermore, the arbitration model incorporates an inbuilt coverage feature. Coverage analysis helps to identify missing stimulus driving and ensures comprehensive testing. Users can disable the coverage facility, if desired.

The core logic of the arbiter operates based on the encoded ID for each input channel. There is a feature in arbitration model that allows users to pass custom information along with the input channel ID, which can be in form of string, enumeration, class, or structure, depending on the user's requirements.

“Fig. 2,” illustrates the channel information. Each channel input consists of the channel ID and any accompanying channel information provided by the user. This flexibility enables users to associate relevant data or context with each channel during the arbitration process.

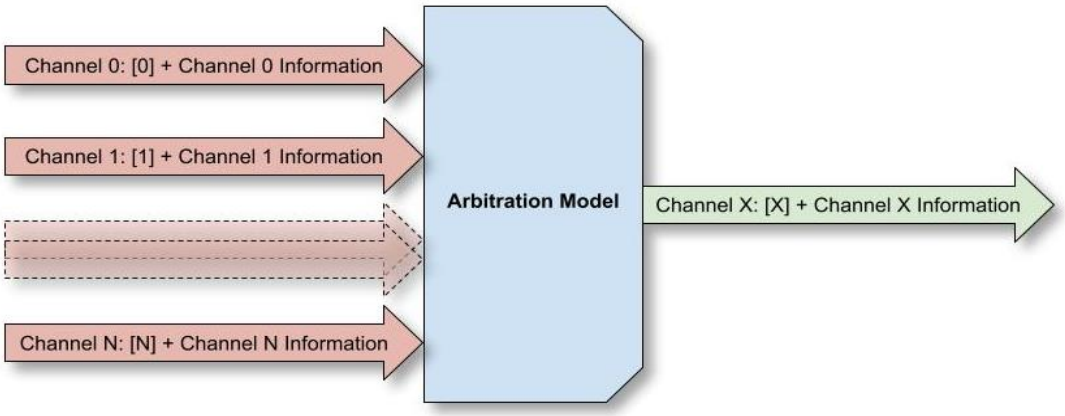

Figure 2. Arbitration Model Channel Information

III. FEATURES

This section describes the features of the arbitration model.

1. **N Number of Input Channels:** Model can be instantiated with N number of arbiter channels with a custom individual channel nomenclature. Each input channel has a dedicated channel number which is used during the arbitration process. It is a versatile model which can be used for arbitration of any protocol message, packet, or information.

2. **Integrated APIs:** Model includes following APIs that can be used for,
   a. "set_weight": Setting weightage for each input channels,
   b. "push_input": Feeding input to arbitration channel, where input is encoded channel ID itself,
   c. "get_expected_packet": Get expected arbitration result where output is encoded channel ID and,
   d. "check_actual_packet": Fetch actual result and compare it with expected arbitration result.

3. **Channel Information:** Model allows user to pass channel related information in any form of System Verilog data type through arbitration model and same channel information passed at expected results. For example, information can be a string, a structure, an entire transaction, etc. Model also facilitates verification of intermediate arbiter(s) without white box testing using channel information. Users can pass channel related information through all arbitration levels and at the end, the expected result can be compared with the actual result. For example, as shown in "Fig. 3,"

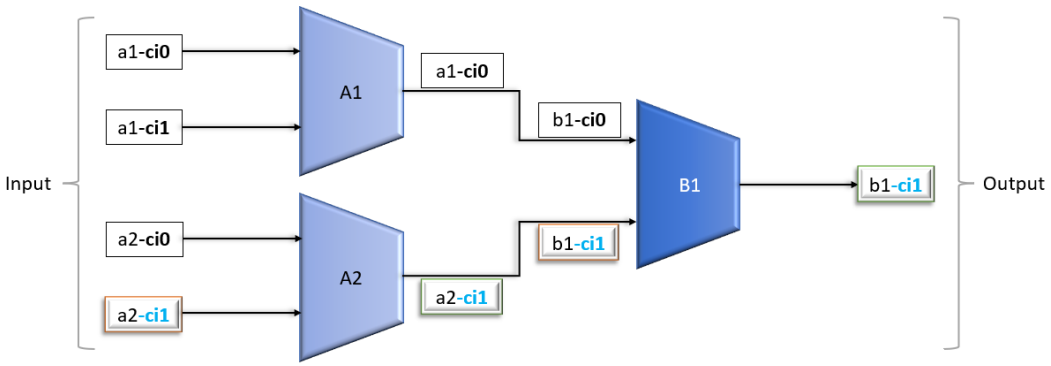

Figure 3. Multi-level Arbitration Model

a2-ci1 is the arbitration result of A2, which is mapped to **b1-ci1** input of arbitration B1. At B1 output, model checks the result of A2 and B1 both. Here, **-ci0** and **-ci1** represent the channel information. This feature simplifies the verification process, saves time, and enhances accuracy.

4. **Inbuilt Coverage Model:** The arbitration model also incorporates an inbuilt coverage model, which helps to ensure that all input combinations are exercised to produce different results. This coverage model also facilitates easy identification of untested scenarios, and quick design bug detection. Following Table I summarizes the overall arbitration coverage model.

TABLE I
Overview of Arbitration Coverage Model

Sr. No.	Name	Bin Type	Bin	Weight	Description
1	<name>_arb_cg	-	-	1	Arbitration Covergroup
1.01	arb_input_cp	Auto	auto[1] auto[2] ... auto[2^N -1]	1	Coverpoint for sampling all possible input combinations. N = number of input channels Ignore Bin = auto[0]
1.02	arb_cycle_start_cp	Auto	auto[CH 0] auto[CH 1] … auto[CH (N -1)]	1	Coverpoint for sampling channel from where one arbitration cycle started
1.03	arb_output_transition_cp	Custom	CH 0 => CH 1 CH 0 => CH 2 … CH (N -2) => CH (N -1)	0	Coverpoint for sampling all possible output transitions.
1.04	arb_output_transition_reason_cp	Custom	NO_INPUT NO_WEIGHT NO_INPUT_NO_WEIGHT	0	Coverpoint for sampling all output transitions reason.
1.05	arb_output_transition_cp_X_arb_output_transition_reason_cp	Auto	-	1	Cross of output transition with its transition reason.

IV. ARBITRATION MODEL INTEGRATION

The arbitration model presented in this paper is highly versatile and can be integrated into any environment that requires the verification of an arbiter implemented in a design. “Fig. 4,” illustrates one of the ways in which the arbitration model can be instantiated in a verification environment.

Consider that arbitration model is instantiated within a block reference model. The reference model receives packets driven to the design and packets driven by the design as output. After performing the necessary processes on these packets, they can be passed to the model. Incorporation of the arbiter model into the reference model allows comprehensive testing and validation of the design’s arbitration.

It's important to note that the instantiation and integration of the arbiter model may vary based on the specific verification environment and design requirements. The provided example of integrating the model within a block

reference model serves as a starting point, but further details and customization may be necessary based on the specific implementation context.

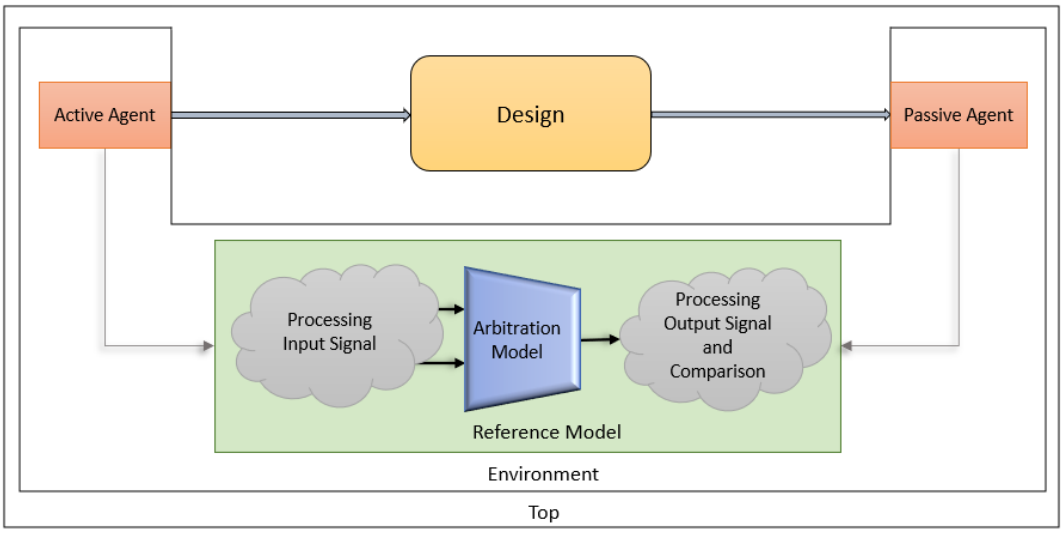

Figure 4. Arbitration Model Integration

In the context of arbitration model integration within a reference model, let's consider two examples: a single level of arbitration and multiple levels of arbitration.

1. **Single-Level of Arbitration**

   In this example, we described a simple arbitration model integrated into a reference model, which consists of a single arbiter in the input-to-output direction. The arbiter has three input channels. “Fig. 5,” illustrates this scenario, where Channel 0, Channel 1, and Channel 2 can be the sources of input packets for both the design and the reference model.

   After the necessary logical operations (Logic 0,1,2) are performed on the input packets, they can be pushed into the model using the respective channel's encoded ID. The design provides the actual packet, and the reference model performs logical checking (Logic 3) on the packet. The channel's encoded ID from the packet can be passed to the arbiter model using the "check_actual_packet" API. If there is a mismatch in the results from the design and the reference model, an error is flagged, indicating discrepancy in the expected and actual arbitration.

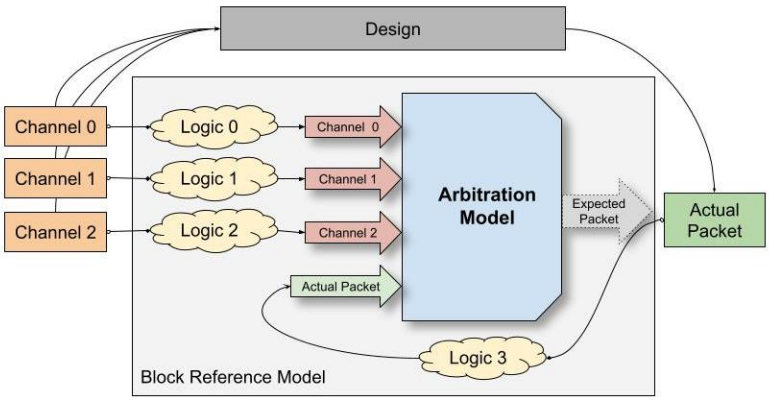

Figure 5. Signal level of Arbitration Integration

2. Multi-Level of Arbitration

In this example, we described the integration of multiple levels of arbitration within a reference model. The design incorporates three arbiter models: two models at level 1 and one model at level 2. The level 2 arbiter receives input packets from the result of the level 1 arbiter. Each arbiter has three input channels. “Fig. 6,” represents this scenario, where Channel 0 to Channel 5 are the sources of input packets for both the design and the reference model. Like the previous example, input packets are pushed into their respective arbiters (Arbitration Model 1 and Arbitration Model 2) along with channel information.

Once the inputs are pushed to the arbiters, the reference model retrieves the results using the "get_expected_packet" API and obtains the channel information. The results from both arbiters are then fed into the input of the level 2 arbiter (Arbitration Model 3) along with the channel information fetched from the level 1 arbiter results. There is a fair possibility of an independent input(s) to level 2 arbiter. In “Fig. 6,”, this is represented as Ch 1 (Logic 9). When there is an actual packet from the design, the reference model performs logical checking (Logic 8) and compares it with the result from the level 2 arbiter. The Arbitration Model 3 provides results using the "get_expected_packet" API. If there is a mismatch between the results (encoded ID or channel information), the checker implemented in the reference model flags an error.

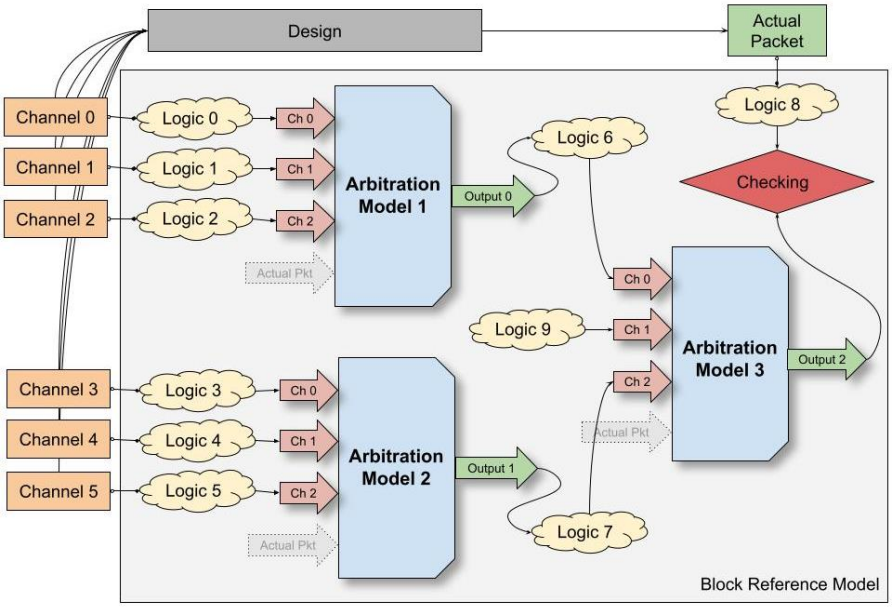

Figure 6. Multiple level of Arbitration Integration

Below are the snippets of single level arbitration model integration steps,

1. Instantiate arbitration module with number of input channels and channel enumeration typedef.

```
26   /**
27    * @brief : CXL arbitration module enum typedef
28    * @var   : arb_cxl_channel_e
29    */
30   typedef enum {H2DREQ_, S2MNDR_, S2MDRS_} arb_cxl_channel_e;
31
32   /**
33    * @brief : CXL arbitration module instance
34    * @var   : cxl_arb
35    */
36   ei_arbitration_c #(3, arb_cxl_channel_e, string) cxl_arb;
```

Figure 7. Model Instance Example Code

2. Creating instance of arbitration module.

```
45   /**
46    * @brief    : New constructor
47    * @function : new
48    */
49   function new(string name = "", uvm_component parent = null);
50     super.new(name, parent);
51     this.name = name;
52     cxl_arb = new({name,"_cxl_arb"});
53   endfunction : new
```

Figure 8. Model Constructor Example Code

3. Setting weight using “set_weight” API.

```
85   // Settng weightage
86   foreach(cxl_arb_weight[i]) begin
87     void'(cxl_arb.set_weight(  i
88                              , cxl_arb_weight[i]
89           ));
90   end
```

Figure 9. Setting Weight API Example Code

4. Giving input packets to arbitration module using “push_packet” API.

```
125  /**
126   * @brief    : Get input packet and push into arbitration module input queue
127   * @function : write_cxl_arb_input_port
128   */
129  virtual function automatic void write_cxl_arb_input_port(ei_uvm_sequence_item pkt);
130    ei_cxl_base_packet_c input_pkt_;
131
132    if($cast(input_pkt_,pkt)) begin
133      if(input_pkt_.link_channel inside { ei_cxl_enum_pkg:: H2DREQ,
134                                          ei_cxl_enum_pkg:: S2MNDR,
135                                          ei_cxl_enum_pkg:: S2MDRS }) begin
136        void'(cxl_arb.push_packet(  (   (input_pkt_.link_channel == ei_cxl_enum_pkg:: H2DREQ) ? 0
137                                      : (input_pkt_.link_channel == ei_cxl_enum_pkg:: S2MNDR) ? 1
138                                      :                                                         2)
139                                  , input_pkt_.link_channel.name()
140             ));
141      end
142    end
143  endfunction : write_cxl_arb_input_port
```

Figure 10. Input API Example Code

5. Getting the output packet and checking it with actual using “check_actual_packet” API.

```
147  /**
148   * @brief    : Get output packet and check it with actual packet
149   * @function : write_cxl_arb_output_port
150   */
151  virtual function void write_cxl_arb_output_port(ei_uvm_sequence_item pkt);
152    ei_cxl_base_packet_c output_pkt_;
153
154    if($cast(output_pkt_,pkt)) begin
155      if(output_pkt_.link_channel inside { ei_cxl_enum_pkg:: H2DREQ,
156                                           ei_cxl_enum_pkg:: S2MNDR,
157                                           ei_cxl_enum_pkg:: S2MDRS }) begin
158        void'(cxl_arb.check_actual_packet(  (   (output_pkt_.link_channel == ei_cxl_enum_pkg:: H2DREQ) ? 0
159                                              : (output_pkt_.link_channel == ei_cxl_enum_pkg:: S2MNDR) ? 1
160                                              :                                                          2)
161                                          , output_pkt_.link_channel.name()
162             ));
163      end
164    end
165  endfunction : write_cxl_arb_output_port
```

Figure 11. Output API Example Code

V. RESULTS AND ANALYSIS

We have used this generic model to perform 4-5 block-level verifications on a complex SoC project. Currently, there are 12 arbiters, with more expected to be added. The implementation of a single arbitration in the testbench requires approximately 400 lines of code. With 12 arbiters, the code length increases to 4800 lines. By utilizing this model, we were able to reduce the overall coding effort by 91%. Such a model can offer several benefits, such as it can help quicker and efficient development of verification reference model, reducing coding errors and improving test coverage. Additionally, it can facilitate verification without white-box testing, which can save considerable time and effort during the verification process.

Below is the snippet of coverage bins generated from project which used this arbitration model,

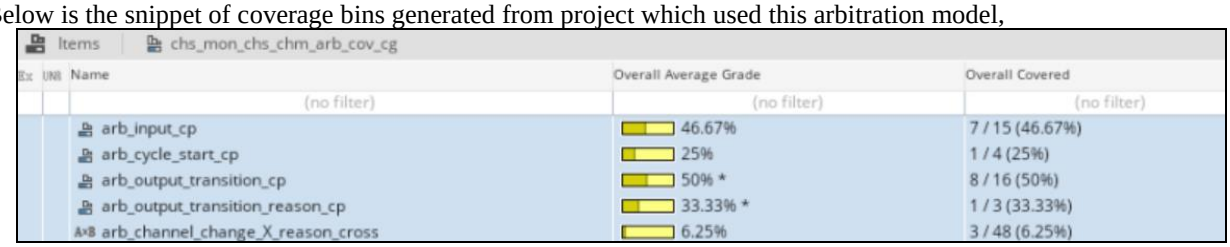

Figure 12. Arbitration Model Cover Points

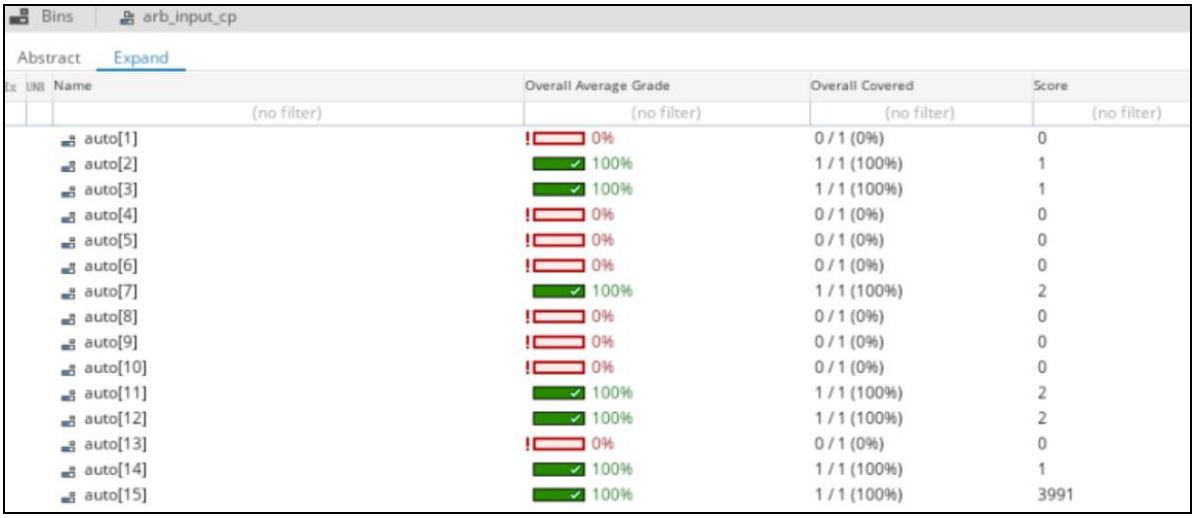

Figure 13. Input Cover Point Bins

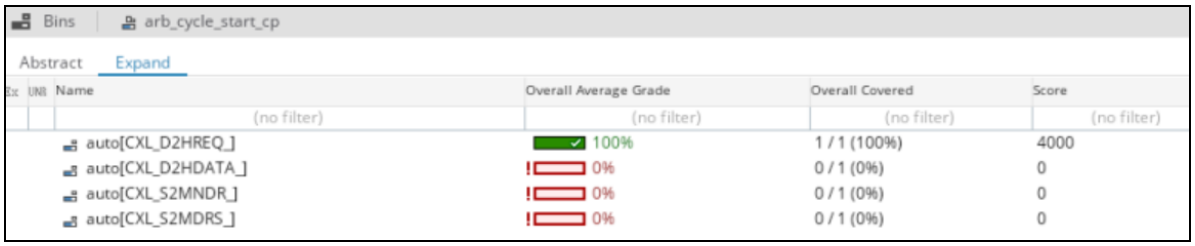

Figure 14. Cycle Start Cover Point Bins

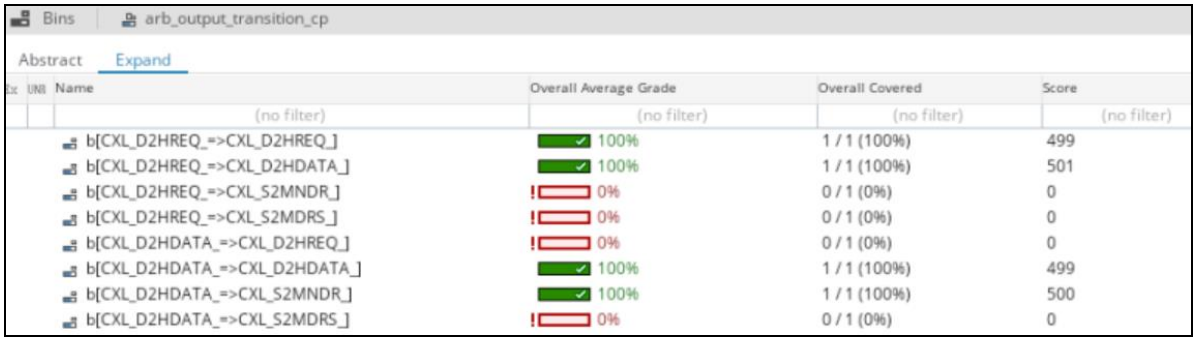

Figure 15. Output Transition Cover Point Bins

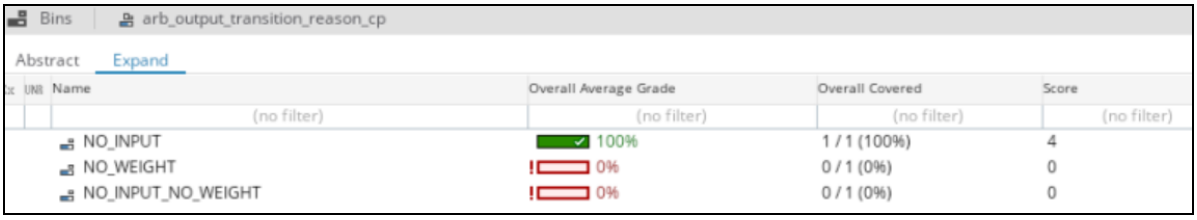

Figure 16. Output Transition Reason Cover Point Bins

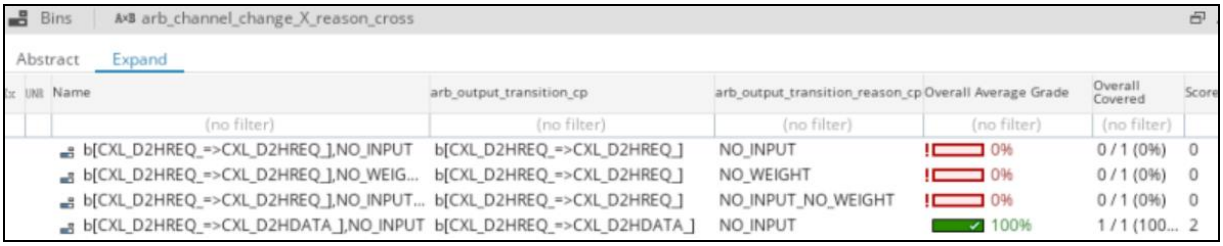

Figure 17. Output Transition and Reason Cross Bins

VI. LIMITATIONS AND FUTURE EXPANSION

When a mismatch occurs in the arbitration result, debugging can be challenging, often requiring waveform analysis to identify the issue. In the arbitration model described, the error message currently provides the expected and actual results. However, it would be beneficial to include additional details in the error message to aid in debugging efforts. This can involve providing insights into why the current result is expected and explaining why the previous channel is not selected as output. By incorporating such details into the error message, the debugging effort can be significantly reduced. The coverage information (Output Transition Reason) can be utilized in the error message to provide more context and insights into the arbitration outcome.

Currently, the arbitration model supports the Classical Weighted Round Robin (WRR) algorithm and single arbitration output. However, there is potential to enhance the model's capabilities to support the Interleaved WRR algorithm and multiple arbitration outputs. By expanding the model's functionality, it becomes more adaptable to diverse arbitration scenarios, catering to a wider range of design requirements.

VII. CONCLUSION

The generic arbitration verification model presented in this paper offers a solution to the challenges associated with verifying multi-level arbitration designs. By providing a reusable and adaptable model that can be integrated into any verification environment, the model reduces code repetition and simplifies maintenance and expansion efforts.

Furthermore, the model facilitates efficient bug detection and untested scenarios through its in-built coverage capabilities.

The paper emphasizes the significance of adopting a comprehensive arbitration verification model in the overall design verification process. By utilizing this model, verification teams can enhance their testing methodologies, improve coverage, and reduce the risk of undetected issues in complex multi-level arbitration systems. The model's flexibility and ability to address various arbitration challenges contribute to its value as a critical component of design verification efforts.

REFERENCES

[1] Katevenis, M.; Sidgiropoulos, S.; Courcoubetis, C. (1991). "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip". IEEE Journal on Selected Areas in Communications. 9 (8): 1265–1279

[2] Z. Fu and X. Ling, “The design and implementation of arbiters for Network-on-chips,”in 2010 2nd International Conference on Industrial and Information Systems, vol. 1, July 2010, pp. 292–295.

[3] Accellera Systems Initiative. (2017). SystemVerilog 3.1a Language Reference Manual.

[4] Accellera Systems Initiative. (2017). Universal Verification Methodology (UVM) 1.2 Language Reference

[5] C. Spear, SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. New York NY: Springer, 2006.