Deadlock Free Design Assurance Using Architectural Formal Verification

Bhushan Parikh, Shaman Narayana
Intel Corporation
5000 W. Chandler Blvd,
Chandler, AZ 85226

I. Introduction

The design of deadlock-free systems can be extremely challenging, and deadlock defects are notoriously difficult to detect. If a deadlock is detected in the silicon (Si), it can delay the time-to-market for the product. In the absence of a workaround, a possible re-spin could cost the chip manufacturer millions of dollars.

Constraint-based random verification (CBRV) at the system level (simulation/emulation) is a widely used methodology to uncover logic defects related to system-level deadlocks. However, covering all possible test cases required to prove the absence of a deadlock in such an environment can be difficult as it requires running many test cases targeting very specific conditions that may need to be present simultaneously in several parts of the design.

Formal verification methods have shown the most promise, but there are serious challenges with their application to finding system-level deadlocks due to the exponential complexity associated with exploring all possible design states at the system level, even after deploying well-known design reduction techniques [1].

Leveraging Architectural Formal Verification (AFV) can reduce the complexity and make it possible to formally prove the absence of system-level deadlock. In this paper, we will make it clear what needs to be done to guarantee the absence of the system-level deadlock using the AFV. We will use a case study of lossless hardware (HW) Compression (Intellectual Property) IP, as a system and show how the proposed method was applied including the decomposition and mapping of a complex system to small block-level properties.

II. Challenges associated with system-level deadlock sign-off

A. Overview of a lossless HW Compression IP

At a high level, the compression IP consumes uncompressed data as input and produces compressed data as the output. Compression is inherently complex due to a large input space and data dependencies [2]. For example, to verify that the IP can compress N bytes of data correctly, there are $(256)^N$ possible input combinations to cover. In our case, the value of N can be in the range of Giga (10^9). In addition, each input combination may exercise a unique combination of the control signals to meet the requirements of the specific compression algorithms.

B. Design Details

As shown in Fig. 1, the design is composed of eight blocks and the uncompressed data stream flows from Ingress FIFO to Match Logic to Entropy Generator to Symbol buffer to Symbol encoder to bits-to-byte packer, and Egress FIFO on the way out. We describe the functions of each of these blocks

Error Handling and Control Logic – This block receives a configuration packet from the user and ensures that the rest of the design adheres to the specified configuration (e.g., compression format). In addition, this block observes error signals from the other blocks and asserts the response signal (complete) back to the user for each submitted request. Before asserting the complete signal, this block must ensure that the rest of the design is in the *initial* state and ready to process the subsequent request irrespective of any error(s).

Ingress FIFO – This block receives input bytes of a request with the number of valid bytes in every cycle and an indicator if these are the last bytes of the request; if the Ingress FIFO does not have space to accept the input, it asserts the stall signal. This block forwards bytes of a request along with the last indication to Match Logic. Ingress FIFO also observes the number of bytes retired by the Egress FIFO block to ensure that the output size of the compressed stream matches the programmed value during configuration.

Match Logic – This block performs a search on the data received from Ingress FIFO and provides multiple match results in terms of length and distance pairs to the entropy block and the symbol buffer. For the data that did not have any match results, Match Logic forwards the data (also known as Literals) to the symbol buffer and the entropy block.

Entropy Generator – This block generates the format-specific entropy based on the symbols received from the Match Logic block. This is one of the complex blocks with multiple state machines interacting with each other. This block must ensure that the generated entropy codes are sent to the Symbol Encoder block before asserting the

entropy generation complete signal. Since the symbol encoder functionality relies on the entropy generation complete signal, the assertion of this signal for every request is one of the key requirements for the system-level requirement of no deadlock.

Symbol Buffer – As the name suggests, this block stores symbols (Literal, Match length, and Match distance) from Match Logic and forwards them to Symbol Encoder as required.

Symbol Encoder – This block converts the symbol from the symbol buffer to its respective entropy codes generated by the Entropy Generator Block.

Bits to Byte Packer – The output of the symbol encoder is bit aligned which is converted to byte aligned using this block.

Egress FIFO – This block is responsible for sending the compressed data back to the user once the data is available and requested by the user.

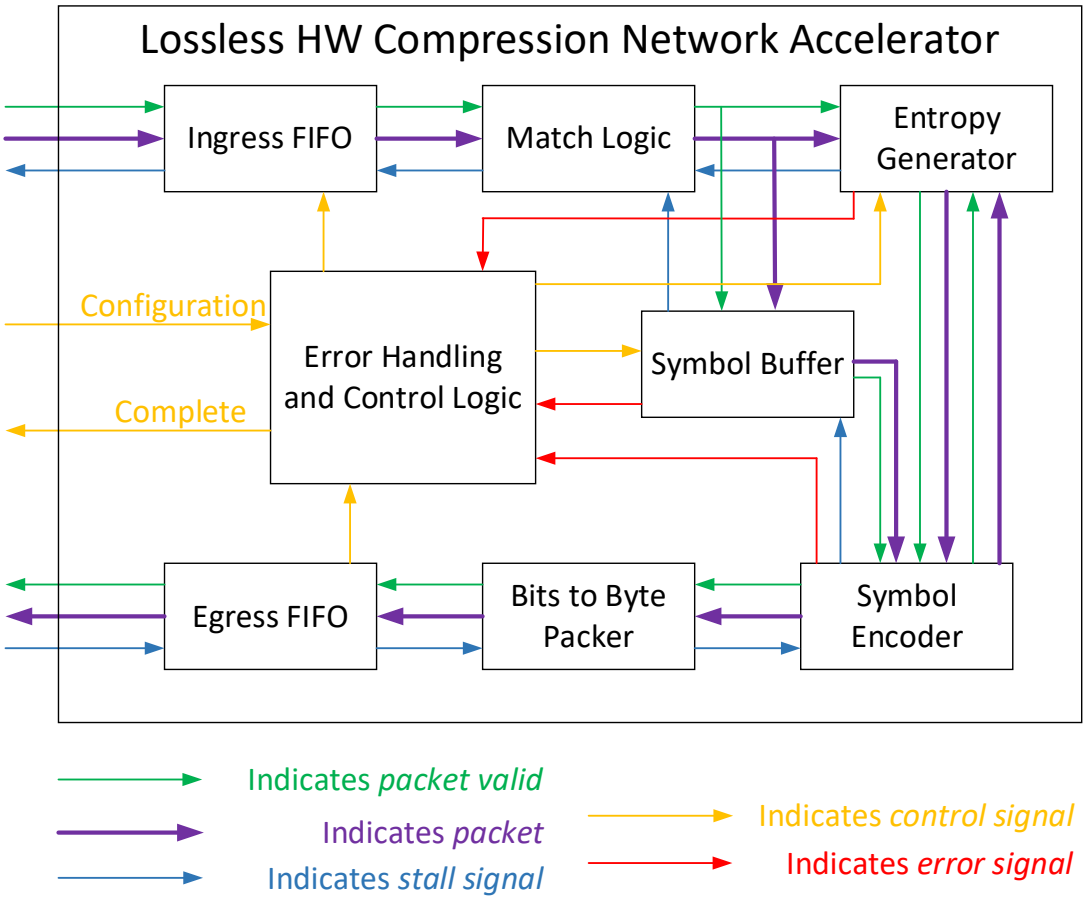

Figure 1. High-level block diagram of the lossless HW Compression IP

C. Problem Statement

A network accelerator such as lossless HW Compression IP must never encounter a deadlock situation. This is a very important reliability aspect. For example, if an attacker gets access to an uncompressed stream that can put the system in an unrecoverable (also known as hang or an error state where it always produces incorrect output) state, then they can bring down the entire network by constantly sending the bad uncompressed stream and thus making it unavailable for rest of the users. This is known as a Denial of Service (DoS) attack. Therefore, it is essential for the lossless HW Compression IP to complete its task and return to its initial state for any input data. This requires us to prove the following,

Lossless HW Compression IP must assert completion signal for every end of the request packet irrespective of any error

Figure 2. System-level requirement of no hang

D. Challenges Using the Traditional FV Methodology

Based on the previous experience with a prior generation of this IP, we already knew that the CBRV methodology will not be effective to cover the breadth of conditions to verify to prove the system-level requirement defined in Fig 2.

Hence, we started using the traditional FV methodology i.e., converted the requirement from Fig 2. in the assertion and tried to prove it using a commercial formal verification tool. As expected, the proof did not converge after a run of 24 hours. This was after reviewing the FV setup and tool settings with FV tool vendor experts ensuring no gaps.

To further quantify how impractical it is to prove the requirement defined in Fig 2 on our large RTL design with commercial FV tools and traditional FV methodology, we did some experiments as described in Table 1. Assert 1 is exactly the statement of the requirement defined in Fig 2, and the proof did not converge within 24 hours. To see how far we are, we ran reachability on coverage targets (Covers 1 to 3). As the table shows, even after running for 24 hours, we were not able to cover the reachability of the entropy generation complete case. Based on this data a monolithic approach is clearly infeasible for proving our system-level requirement. Fig 3. describes respective simplified properties for these cover and assert targets.

TABLE I
RESULTS USING TRADITIONAL FPV METHOD

SVA Property #	SVA Property	Status	Bound #	Time
Cover 1	Input packet is sent from *Ingress FIFO* to *Match Logic*	Covered	6	< 1 minute
Cover 2	*Match Logic* processed input and sent output to *Entropy Generator*	Undetermined	50	~ 24 hours
Cover 3	Entropy generation is complete	Undetermined	38	~ 24 hours
Assert 1	A completion signal is asserted for every end of the request packet	Undetermined	15	~ 24 hours

```
#cover_1
cover –name input_pkt_progress_to_match_logic
                    {in_fifo_vld_pkt_out & (in_fifo_pkt_out_len_bytes > 1)}

#cover_2
cover –name match_logic_processed_and_output_sent_to_entropy_gen
                    {$rose(match_logic_vld_pkt_out & start_entropy_gen)}

#cover_3
cover –name entropy_gen_complete
                    {$rose(entropy_gen_done)}

logic           end_of_request;
always_comb end_of_requeset =
                    packet_vld & (packet_payload[3:0] == END_OF_REQUEST_PACKET);

#assert_1
assert –name completion_signal_asserted_for_every_packet
                    {$rose(end_of_request) |=> s_eventually(completion)}
```

Figure 3. Simplified code for covers and assert properties described in the Table I

III. PROPOSED METHODOLOGY

As we have seen in the previous section, the traditional way of proving deadlock-free behavior is not practical. We will show how the same can be achieved by applying the compositional verification method often referred to as AFV, illustrated in Fig. 4.

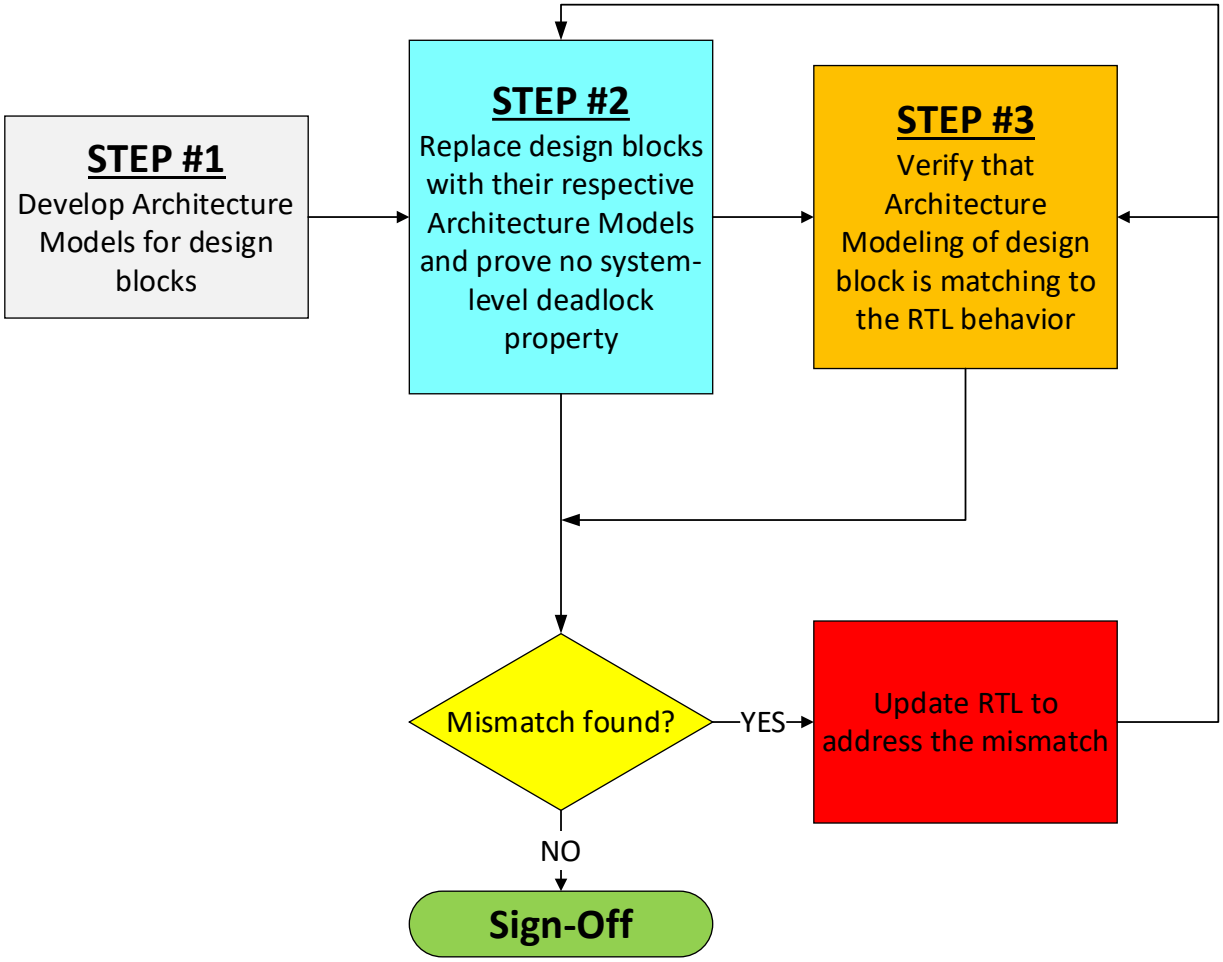

Figure 4. Architectural formal verification steps

A. Develop Architectural Models (AMs) for Design Blocks

The first step is to develop Architectural Models for the design blocks. This is achieved by manually crafting a set of System Verilog (SV) properties describing the system-level requirements for the respective design blocks. The design blocks of interest in our lossless HW Compression IP are Ingress FIFO, Match Logic, Entropy Generator, Symbol Buffer, Symbol Encoder, Bits to Byte Packer, and Egress FIFO.

Taking Ingress FIFO as an example to illustrate step 1, the system-level requirements of this design block are as follows:

- In the absence of an error, the Ingress FIFO should forward each of the incoming packets to the Match Logic 16 Bytes at a time
- In case an error is detected, the Ingress FIFO should send the end of the request packet to Match Logic and drop all incoming packets
- After the completion signal is asserted, the Ingress FIFO should drop all subsequent packets, except the end of the request packet, for a request until the configuration command is received

- If the number of bytes retired by Egress FIFO is bigger than the size specified in the configuration packet, then the Ingress FIFO should send the end of the request packet to Match Logic and drop all incoming packets

The above four system-level requirements can be represented by the following SV properties as shown in Fig. 5. It is important to note that all control-specific design requirements should be modeled in AMs.

```
module Ingress_FIFO (
                                //input
                                input [000:000] packet_in_valid,
                                input [128:000] packet_in_payload,
                                input [003:000] packet_in_len_bytes,
                                input [000:000] error,
                                input [000:000] completion,
                                input [000:000] output_bigger_than_expected,

                                //output
                                output [000:000] packet_out_valid,
                                output [063:000] packet_out_payload,
                                output [003:000] packet_out_len_bytes,
                                output [000:000] get_next_packet,
                                output [000:000] full
                        );

      localparam END_OF_REQUEST_PACKET = 15;

      assume –name forward_packets_if_no_error
                        {in_fifo_vld_pkt_in & ~error |=> ##[0:5] in_fifo_vld_pkt_out}

      assume –name drop_packets_if_errors_or_completion_or_biger_output
                        {$rose(error | completion | output_bigger_than_expected) |->
                              (~packet_out_valid & get_next_packet) s_until (packet_payload[3:0] == END_OF_REQUEST_PACKET)}

      assume –name no_input_packets_if_ingress_fifo_is_full
                        {full |-> ~get_next_packet}

      assume –name send_end_of_request_packet_if_error_or_bigger_output
                        {$rose(error | output_bigger_than_expected) |=> (packet_out_payload[3:0] == END_OF_REQUEST_PACKET)}

endmodule
```

Figure 5. Example of Architecture Model highlighting Ingress FIFO block.

B. Replace design blocks with AMs and prove system-level requirement(s)

After developing the AMs for each of the design blocks the next step is to replace the design blocks with their respective AMs and prove the system-level requirement(s).

Using the hand-written model developed in Step 1, we achieved the convergence for the system-level requirement defined in Fig 2. We used the same FV tool that we used for the traditional FV methodology.

Compared to the challenges faced that were discussed in Section II, the AFV methodology ensured that properties were proven quickly. In Section IV, we will describe the results in greater detail.

C. Verify that AMs match the RTL behavior

The last step is to prove the block-level properties using the traditional formal verification methodology. We leveraged different known strategies such as case-splitting, reset abstraction, etc. to reduce the design complexity and achieved full convergence for all block level proofs [3].

Since the proof achieved using the AFV relies on the accuracy of the AMs, it is important to resolve any mismatches between AMs and the RTL behavior found during the block-level verification step. In addition, if any changes to the AM(s) is(are) required then the system-level requirement shall be proved again with the updated AM(s) using the AFV.

IV. RESULTS

We leveraged AFV and successfully proved our system-level requirement captured in Fig 2. Table II below is the updated version of Table I using AFV. As shown, we were able to achieve full convergence for Assert 1 case in less than 30 minutes. In addition, other properties are also covered in less time which were uncovered using the traditional FV even after running for 24 hours.

TABLE II
RESULTS USING ARCHITECTURAL FORMAL METHOD

SVA Property #	SVA Property	Status	Bound #	Time
Cover 1	Input packet is sent from *Ingress FIFO* to *Match Logic*	Covered	6	< 1 minute
Cover 2	*Match Logic* processed input and sent output to *Entropy Generator*	Covered	75	< 5 minutes
Cover 3	Entropy generation is complete	Covered	84	< 10 minutes
Assert 1	A completion signal is asserted for every end of the request packet	Covered	52	< 30 minutes

Table III describes high-level categories of the bugs found by AFV.

TABLE III
RTL BUG CATEGORIES

RTL bug category	Details
Missing capturing of each error case in the control finite state machine	Error asserted for specific data combination led to deadlock
Incorrect signal sensitivity	The valid signal protocol was not followed
Missing assertion of a valid signal at the right time	Entropy generation is complete not asserted in a few corner cases
Counters not incrementing correctly	Some of the counters were not behaving as expected resulting in the incorrect output

Among the various bugs found in the design, there were two interesting corner cases:

A. Logic defect 1

Related to the assertion of the completion signal from the symbol buffer indicating the symbol buffer is empty. After the data is read out of the symbol buffer it should have asserted the symbol buffer empty signal. In addition, this signal should be sticky and not a pulse.

However, depending on the different format and entropy of the input data, the symbol buffer would assert the completion signal differently,

- It may assert it a clock cycle later
- It may assert it as a pulse and not a sticky signal

To analyze what would it take to identify these bugs using CBRV methodology, we continued to execute our CBRV strategy against the design without fixes for this bug. We were able to identify the bug related to the pulse vs sticky behavior of the completion signal within a week of the regression run; however, it took us approximately six months to find the issue where the completion signal is asserted a clock cycle later.

B. Logic defect 2

Related to corrupted configuration packet.

In this case, the control FSM was not accounting for a case where the configuration packet is corrupted in a way where it programs some of the fields with valid but not Plan of Record (POR) values.

In addition to this, it also highlighted a weakness in the design where the entropy generator was not asserting completion signals for all supported valid values. We would have not been able to find this issue in our CBRV as it was never targeted as part of the CBRV test plan.

IV. SUMMARY

System-level deadlock logic defects are extremely elusive, especially for complex designs such as our Compression IP. However, using the AFV methodology described in this paper, we achieved the formal proof for the system-level requirement of no deadlock in only four weeks. In addition to finding 100% logic defects related to the no deadlock requirement, through the step of comparing AMs with RTL we found logic defects in other areas as well.

The AFV can be implemented at an early stage of RTL development, and it is not architecture specific. Furthermore, it builds upon existing block-level FV methods, which are familiar to FV engineers. The AFV requires creating block-level models and building proofs using these properties. This is an iterative process, which requires a thorough understanding of the architecture. A collaborative effort between the designers and FV engineers is key for its successful execution.

We hope that this paper will encourage more engineers to apply the AFV methodology.

REFERENCES

[1] Adnan Aziz, Vigyan Singhal, and Robert K Brayton, "Verifying interacting finite state machines: Complexity issues," University of California at Berkeley, 1993

[2] Bhushan Parikh, Shaman Narayana, Buck Lem, David Cassetti, "Accelerating the IP Design Cycle with Formal Techniques Beyond Everyday FPV," DVCON USA, March 2021.

[3] Erik Seligman, Carl Dreyer, Ken Haren, Raman Nayyar, "Zero Escape Plans: Tying Together Design, Simulation, and Formal Methods for Bulletproof Stepping Validation" DVCON USA, February 2008.