2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Discovering Deadlocks in a Memory Controller IP

Jef Verdonck, Emrah Armagan – u-blox AG, Leuven, Belgium, {jef.verdonck, emrah.armagan}@u-blox.com
Khaled Nsaibia, Slava Bulach – u-blox AG, Thalwil, Switzerland, {khaled.nsaibia, slava.bulach}@u-blox.com
Pranay Gupta, Anshul Jain, Chirag Agarwal – Oski Technology, Inc., Gurgaon, India, {prgupta, anshul, chirag}@oskitech.com
Roger Sabbagh – Oski Technology, Inc., Ottawa, Canada, rsabbagh@oskitech.com

***Abstract*—The risk of deadlocks is one of the areas that is not well addressed by dynamic testing. Simulation does not provide the tools to target deadlocks directly, so finding deadlock scenarios generally happens by chance. On the other hand, formal verification is particularly well suited to verifying a wide range of forward progress properties of designs, such as absence of deadlock, live-lock, and starvation. In this paper, we present a formal verification methodology that has been shown to predictably discover deadlock in RTL designs. The methodology is applicable in the early phases of IP development and design integration. We share results from the application of the method on an industrial memory controller IP.**

Keywords—deadlock, formal verification, memory controller

I. Introduction

Deadlock in IC design occurs when there is a circular-wait condition. It may happen when there is communication between blocks. For example: block A is waiting for block B to send a response, block B is waiting for block C, and block C is waiting for block A. It may also happen when there is resource sharing amongst blocks. For example: blocks A, B and C each require lock on two resources to make progress. Each has one resource allocated and there are no additional free resources available, so they are all stuck in a wait state.

There are several challenges to be addressed when verifying a design for absence of deadlock through traditional dynamic testing:

1. Absence of a method to directly target deadlocks
   There are no known test stimulus generation methods that are specifically designed to target deadlocks. Discovery of a deadlock in simulation is generally considered to be an unforeseen effect of testing for some other functional requirements of the design, or in short, a “lucky find”.

2. Simulation resistance of deadlocks
   Deadlock scenarios often occur in rare cases that involve several preconditions occurring in sequence with specific timing. Consequently, they may arise only in extreme corner-case conditions, making them resistant to discovery through random testing. Moreover, it also increases the difficulty of knowing all the possible test sequences and combinations of events that might expose deadlock cases.

   Even emulation and FPGA prototyping, which help improve test runtime and coverage, still suffer from these fundamental limitations. The coverage obtained through these approaches have strong dependencies on the availability of use-case descriptions, which are often insufficient and simplified as compared to the real-world scenarios.

3. Coverage is not a guarantee
   Also, time-to-market pressure limits the time allotted to closing coverage and there exists the risk that some important coverage items may be missed or waived. Finally, even though coverage metrics report no coverage holes, it does not guarantee that the entire legal state space has been covered.

Modern IC designs are becoming increasingly complex, with a high degree of parallelism and concurrency, which increases the risk of deadlock states, and makes it harder for traditional methods to find them.

II. Formal Verification Methodology

The formal verification Capability Maturity Model (CMM) [1] defines five distinct levels of formal verification capability. Level 3 of the CMM is ABV formal, which augments simulation with formal analysis and is usually associated with bug hunting. However, it is focused on local properties, such as designer RTL assertions added for specific structures like arbiters, FIFOs and interface handshake signals. Hence, Level 3 capability is not necessarily sufficient for discovering all sources of deadlocks, which can only be found with end-to-end checkers that are a key component of the Level 4 skill set. Level 4 formal [2] uses end-to-end checkers to replace block-level simulation testing with formal analysis, eliminating the need to develop a unit-level simulation testbench.

End-to-end checkers for deadlock model when the design-under-test (DUT) has the "ball". For example, when a block has received more input transactions than corresponding sent transactions at the output, and there is no back-pressure at the output side, then the design has the ball – it is expected to make forward progress with the input transactions and move the ball. In each state where the design has the ball, we expect to see a state transition within a finite time, even if there is no new activity at the inputs to the DUT. This type of check allows the formal analysis to directly target proving the absence of deadlock.

III. Case Study: Memory Controller IP

The design-under-test (DUT) in this case study is Memory Controller (MC) block in a mobile SoC. The architectural block diagram of the design is shown in Fig. 1. The MC is responsible for routing commands and data back and forth between the Arm AMBA AXI® [3] interface and the memory device.

The MC arbitrates between Read and Write commands received on parallel channels from different masters in the system, such as CPUs and DMA controllers. It supports multiple transaction types, such as normal bursts, unaligned bursts and wrapping bursts. It must account for out-of-order Read and Write address and data information received on the AXI bus. It also protects against Read-after-Write (RAW) hazards, by stalling younger Reads until Writes are completed. These complexities put the system at risk of deadlock.

In the following sections, we will discuss the strategies that we used to discover previously unknown sources of deadlock in the design.

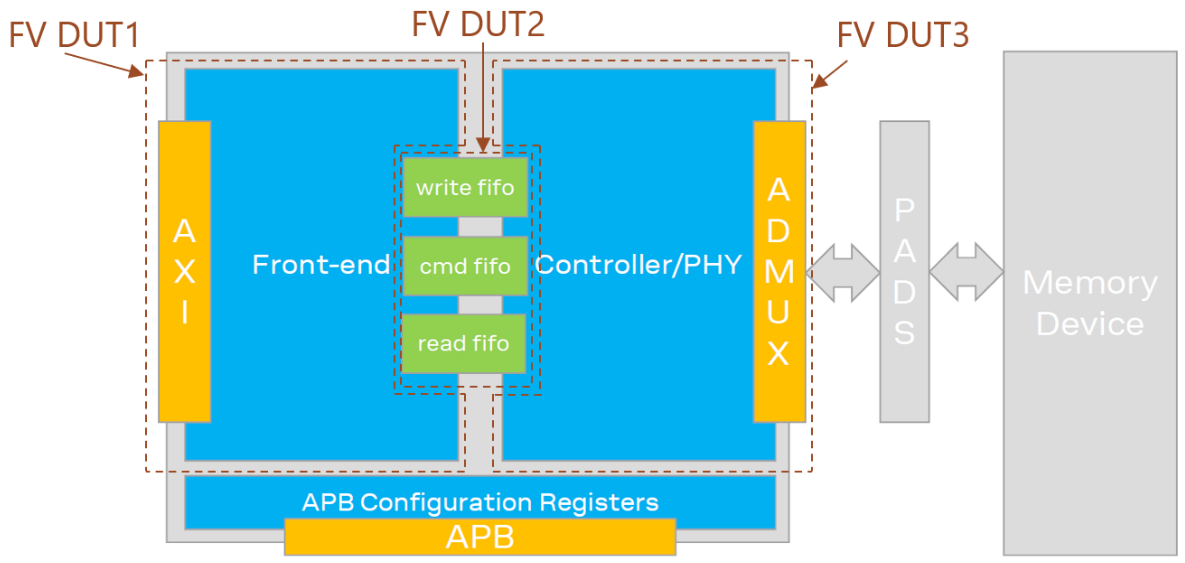

Fig. 1. Memory Controller high-level architecture

A. *Dealing with Complexity*

The MC is a large design and complex IP, consisting of over 10,000 lines of RTL code and over 40 design modules. To overcome complexity barriers and enable deep formal analysis, the design was decomposed into three parts, along logical boundaries, as shown in Fig. 1. The following sections describe the function of each of these parts and their deadlock worry cases that were identified in the verification planning process.

1. AXI Front-End (FE)

   The AXI FE is responsible for communicating with AXI bus masters to receive memory read and write commands, arbitrate between them, and send responses when they become available. It also has the RAW hazard protection logic.

   Deadlock worry cases include:

   - Read path stall due to false RAW hazards
   - Spurious read requests consuming resources that never get freed
   - Incorrect volume of read data signaled to the Controller/PHY – example: requesting less data than required would leave the AXI FE waiting for data that would never arrive

2. Asynchronous Queues

   The asynchronous queues handle the clock-domain crossing (CDC) of commands and data between the AXI clock and the memory clock domains. It consists of a Command FIFO, a Write Data FIFO and a Read Data FIFO.

   Deadlock can occur with data integrity failures in the queues, for example:

   - Commands are corrupted or dropped in the Command FIFO leaving the AXI FE stuck forever awaiting a response
   - Data fails to clear out of the Read or Write queues as expected, which can build up over time, leaving them full and unable to accept any new data, and the system deadlocked
   - Data is dropped while crossing the clock domain boundary, leaving the AXI FE waiting for read data or the Controller / PHY waiting for write data that will never arrive

3. Controller / PHY

   The Controller / PHY converts the AXI commands to memory interface signals with the required timing relationships.

   Deadlock worry cases include:

   - Commands are dropped before reaching the memory interface leaving the AXI FE stuck waiting for a response
   - Extra data is pushed into the Read FIFO, which builds up over time, leaving it unable to accept new data
   - Data is dropped before being pushed into the Read FIFO, which leaves the AXI FE waiting indefinitely for the transaction to complete

B. End-to-End Checkers

1) AXI FE

Examples of the end-to-end checkers planned for the AXI FE include:

- Forward progress checkers for Reads
  - If there are any pending read requests, then RVALID should assert within a finite time duration
  - If there are RAW hazard conditions, the RAW hazard should resolve within a finite time duration
  - There should be a finite number of Read requests in the pipeline of the design at any given time
- Command correctness checkers
  - There should be no spurious command sent to the Async CMD queue for an address which does not match to a transaction on the AXI interface
  - For a Read request, the number of words requested by the command sent to the Async CMD queue should be the same as the request received at the AXI interface

2) Asynchronous Queues

Examples of the end-to-end checkers planned for the Asynchronous Queues include:

- Command Queue
  - Forward progress – commands pushed in should be popped out within a finite time duration
  - Data transport – there should be a one-to-one relationship between commands pushed in and commands popped out
- Write Data Queue
  - Forward progress – data pushed in should be popped out within a finite time duration
  - Data transport – there should be a one-to-one relationship between data pushed in and data popped out
  - Pop interface protocol – when a pop is requested from the queue, all of the data for the corresponding command should be available at the pop side of the queue
- Read Data Queue
  - Data transport – there should be a one-to-one relationship between data pushed in and data popped out

3) Controller / PHY

The Controller / PHY formal environment included a model for the external memory, which modeled the responses to commands arriving at the memory interface.

Examples of the end-to-end checkers planned for the Controller / PHY include:

- Forward progress on the round-trip Read data
  - For every Read command received at the CMD Queue interface, the corresponding Read data is pushed to the Read Data Queue interface within a finite time duration
- Read data transport

  Compare the data from the memory to the Read data that is pushed into the Read Data Queue. Check that:
  - No spurious Read data is pushed into the Read Data queue
  - Max number of pending Read data words in the system at any given time is not exceeded
- Forward progress on Write data
  - If there is pending Write data in the system, then data will be sent on the memory interface within a finite time duration
- Write data transport
  - No spurious Write data written to the memory interface
  - Max number of pending Write data words in the system at any given time is not exceeded

C. Constraints Validation

By default, formal verification explores every possible input combination, so constraints are required to filter out the illegal input space. An important step in the methodology is validation of the constraints to ensure bugs are not missed due to unintentional over-constraints. There were two primary methods of constraints validation that we used in this case:

1. Simulation of Constraints

   Constraints at the primary interfaces to MC were run as assertions in simulation. A violation of these assertions would indicate that the system is using the MC in a way that formal verification did not test for, which is a red flag. This would mean that the constraints would have to be updated and the formal analysis run again to check for any new failures.

2. Cross Proofs of Constraints

   Constraints on signals that go between formal DUTs were proven against the neighboring block that drives the signals, in an assume-guarantee approach.

D. Bugs Found

1) AXI FE

One example of a forward progress issue which was discovered in AXI FE is depicted in Fig. 2. This issue is related to the RAW protection logic. The sequence of events that leads to blocking of Read commands is as follows:

1. AW channel receives a write request (AW0)
2. W Channel has not yet received all the data for the AW0 request, therefore progress of the AW0 request is blocked until all write data for AW0 is received
3. In the meantime, AR channel receives a read request (AR0) at an address overlapping with the address range of the AW0 request
4. A RAW Hazard is detected between AW0 and AR0. AW0 is still waiting for the remaining write data on the W channel and AR0 is waiting for the hazard to resolve (i.e. AW0 write request to move forward)
5. AW channel then receives another write request (AW1), that also has address range overlapping with the AR0 request
6. After receiving the AW1 request, W channel receives all data for the AW0 request. Therefore, validating AW0 to move forward in the design.
7. After the AW0 request moves forward, the RAW hazard is again detected. This time between AW1 and AR0, even though the AR0 request is an older read. Detecting the RAW Hazard again will block the AR0 request until the RAW Hazard is resolved.

The consequence of this issue is that MC would deny Reads from progressing as long as Writes are focused on the same address region. This is due to RAW hazards detected again and again with younger writes. Furthermore, when the RAW condition eventually resolves and the Read is processed, it would get a response with data from the younger writes.

It may happen that the master which is sending the transactions to MC is waiting for the older Read to finish before sending the Write data for the younger write. In that case, a circular-wait condition would be created. Since the older Read is blocked due to the younger Write and the younger Write is blocked due to Write data not being available to MC, then the data-path is stuck in deadlock.

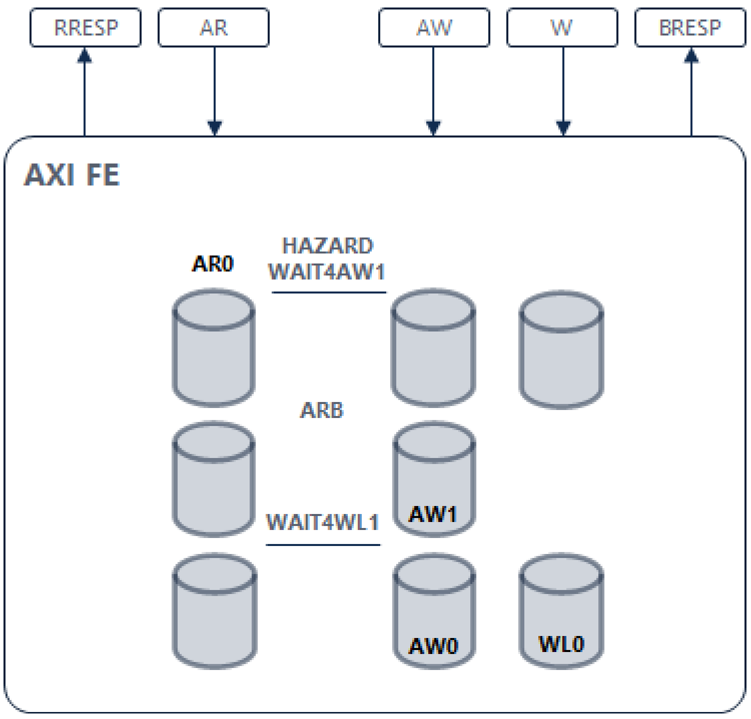

Fig. 2. Read-after-Write hazard deadlock scenario

2) Asynchronous Command and Data Queues

One of the properties of the Async Write Data Queue is that when a pop is requested to Write Data Queue, all of the data for the corresponding command should be available at the pop side of the queue. This property failed with the following scenario, as depicted in Fig. 3:

1. A write command (WR0) is popped from Command Queue with length of 5 data words
2. After accepting the write command the Controller / PHY sends a pop request to Write Data Queue for fetching the corresponding data for the command
3. A maximum of 2 data words can be fetched from the Write Data Queue at any given time, therefore a pop request is sent for fetching 2 out of the 5 data words for WR0
4. However, the occupancy of the Write Data Queue on the pop side is only 1 data word in this case
5. Because of the violation of the interface protocol, the Controller / PHY is expecting 2 data words to already be available on the bus and samples 1 word of real data and 1 word of garbage data at the pop interface
6. The Write Data Queue pop side pointer is then updated with the balance of the 4 words in the burst
7. Later, the Controller / PHY reads the remaining 3 words, which leaves 1 word orphaned in the Write Data Queue after completion of the command

There are two critical impacts of this bug:

- Firstly, there is a data integrity issue for the Write data being written to the memory, which contains one word of garbage data
- Secondly, if this scenario happens repeatedly, the Write data queue will eventually fill up with orphaned data, and it will stop accepting any new data and the Write data path will become deadlocked

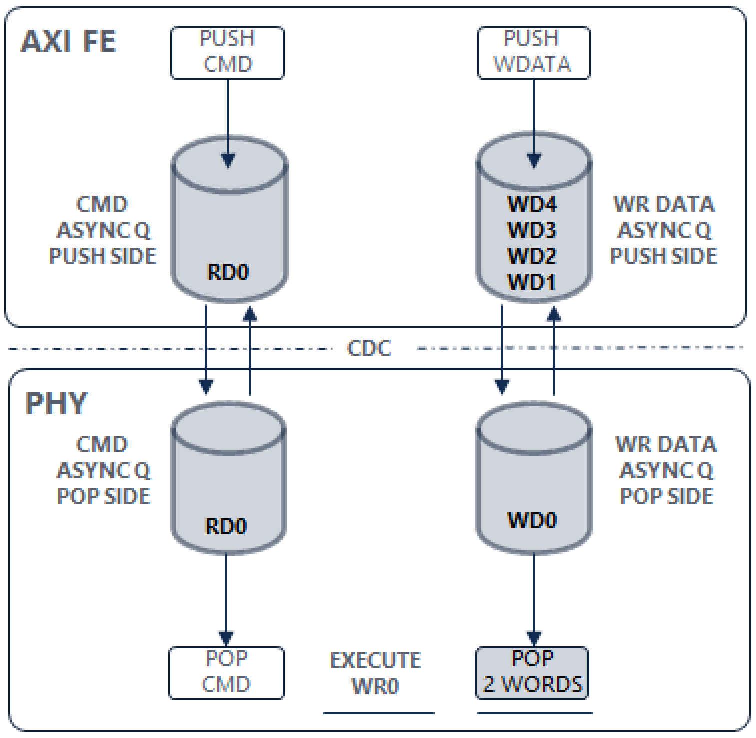

Fig. 3. Orphaned Write Data in Queue leading to Write deadlock Scenario

E. *Effort*

A big advantage of formal analysis for deadlock detection is that with reasonable effort and suitably accurate assumptions about the environment, we can achieve high quality results in a relatively short time. The formal results were obtained for the MC in about 1 month of engineering effort. These deadlocks had not been discovered after many months of dynamic testing.

IV. Conclusions

Formal deadlock verification is an effective and pragmatic method for discovering deadlocks in complex SoC IP blocks. For the MC design, we used Level 4 formal techniques to find deadlock bugs that were undetected through many millions of cycles of dynamic testing.

Acknowledgment

The authors would like to thank the MC designers Andrew Evans and Jaime Murillo.

References

[1] B. Bailey, "Adding Order And Structure To Verification", https://semiengineering.com/adding-order-and-structure-to-verification/
[2] S. Neb, C. Agarwal, D.K. Gupta, R. Sabbagh, "Formal Verification of a Highly Configurable DDR Controller IP", DVCon Europe 2018
[3] https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf