2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Unleash the Power of Formal for Post-Silicon Debugging

Jan Hahlbeck, Shreya Upadhyay,
NXP Semiconductors Germany GmbH,
Hamburg, Germany
(jan.hahlbeck@nxp.com, shreya.upadhyay@nxp.com)

***Abstract*—This paper shows how we adopt formal property verification to accelerate post-silicon debugging. By using cover properties to re-create bugs rather than a traditional simulation based testbench, we take advantage of the full state space exploration of formal tools. The used flow is based on seven steps, which can be also applied by beginners and new adopters of formal verification, as just a subset of skills is required to set up the environment. For many bugs a single cover property is sufficient to re-create and debug them. The implemented setup can be also used to prove the absence of the bug after a design fix or to test a workaround. By means of an example, we showcase that we were able to re-create a hard to find corner case deadlock in one of our designs. The bug escaped the UVM based pre-silicon verification and got initially observed by the post-silicon validation team. After the implementation of a cover property with respect to the provided information, the formal tool consumed just a few minutes of execution time to provide the formal trace of the observed deadlock.**

***Keywords*—Formal Verification, Formal Property Verification, Post-Silicon Debugging**

I. INTRODUCTION

As supplier of modern System-on-Chip (SoC) designs for radio and audio processing in the automotive domain, we are faced with an increasing design complexity on one side and an aggressive time-to-market schedule on the other side while keeping the product quality high. It happens more frequently that pre-silicon verification teams are not able to complete the full-fledged verification anymore within the project schedule and furthermore they need to take well thought decisions to prioritize their efforts. Undetected corner case design deadlocks are a nightmare for design and verification engineers, as they can seriously harm the product usability at customers end and may lead to a loss of trust. To catch these type of bugs in the pre-silicon verification phase we require highly experienced engineers and precisely written test specifications in combination with a well selected verification methodology. Still there is a chance that critical deadlocks escape the pre-silicon verification and are still present in silicon. To mitigate this risk in multi-spin projects, post-silicon validation is heavily used to run long term tests which are not feasible to be executed at that extend in a simulation based environment. As soon as the post-silicon validation team is able to bring the design into an unexpected state, the corresponding test case can be handed over to the pre-silicon verification team which tries to re-create the issue in their environment. Many engineers still tend to use a traditional simulation based testbench to re-create the bug, even though the power of formal verification is known to verification engineers. For corner case bugs, which occur after hours on silicon it might create a huge effort in terms of correct constraint randomization and a high number of executed test runs to eventually run into the faulty behaviour. Using formal to accelerate the debugging of post-silicon issues [1] and as part of our initiative to push formal verification as methodology [2] by providing a hybrid verification environment we started using Formal Property Verification (FPV) to tackle post-silicon bugs in a manner which is suitable even for beginners as most of the work relies on assumptions and cover properties rather than complex assertions. In an example we show that we were able to catch a rare corner case deadlock in one of our designs, which took up to 20 hours to appear on silicon. By using just a single cover property we unveiled this bug within just 73 seconds. The only additional effort which was required to set up the formal testbench was adding Assertion-based Verification IPs (ABVIPs) and some easy to implement fairness assumptions. All assumptions and cover properties are implemented by using System Verilog Assertions (SVA). As formal FPV tool we use Cadence Jasper FPV [3].

II. FORMAL BASED POST SILICON DEBUGGING FLOW

Using formal property verification for post-silicon debugging is known as "reactive FPV" [4] and it takes advantage of the ability to explore the entire space of all reachable design under test (DUT) states. Instead of implementing the stimuli, it is possible to describe the target state. The formal tool is capable to generate an evidence, namely a formal trace, of how to reach this state. The main effort of this approach is to implement the observed faulty behaviour as SVA sequence with respect to all reported boundary constraints. Our process to re-create a bug follows the "UNEARTH" guide [5] which consists of seven phases:

1. **U**nderstand the problem
    a. The post-silicon validation team needs to deliver all kind of information, which is known about the faulty test, like the test implementation, initial analysis results, SoC/IP documentation and all known register settings.
2. **N**ail down the formal property
    a. Implement a single cover property which includes all provided constraints.
3. **E**nvironment build for FPV
    a. Either re-use an existing FPV testbench or create a new testbench toplevel module and instantiate the corresponding module(s) as DUT. Instantiate ABVIPs for standard protocol interfaces like AMBA AHB, APB or AXI and configure and connect them properly. ABVIPs ensure valid stimuli according to the protocol specifications.
    b. Custom interfaces, which are not following a standard protocol, must be modeled manually by adding custom SVA assumptions.
4. **A**ssess the reachability
    a. Several iterations may be required to get the FPV environment in the right mode based on the initial trace e.g., when the tool used some automated blackboxing or other optimizations.
5. **R**egulate the environment assumptions
    a. Recommendation is to start with an underconstrained environment and slightly add assumptions based on the feedback of initial runs.
6. **T**ap the details from sample simulation waveform
    a. If required the FPV tool can be initialized based on a pre-recorded simulation wave to get the DUT already in a "warm" state
7. **H**arness the full power of FPV
    a. Once the bug got discovered the existing cover can be used to prove the absence of the bug after the RTL got fixed.

All seven steps are not strictly sequential, some of them might go in parallel or it is required to go back and forth. Just as in the regular FPV flow, it will need several iterations to get everything right. The flow can be used for any kind of functional bugs, for instance to debug deadlocks, livelocks, or any other kind of unexpected design behaviour.

III. SHOWCASING THE APPROACH

In the following section, we showcase how we applied the post-silicon bug hunting flow to re-create and debug a rare corner case deadlock which escaped the pre-silicon verification. As mentioned in the previous section, the steps of the UNEARTH approach are not mandatorily sequential, e.g. we prefer to set up the FPV testbench before implementing the property to have all hierarchical paths already in place.

A. Understanding the Problem

During the post-silicon validation phase of a signal processing SoC the validation team reported that one of their tests fails randomly after a runtime of 10 to 20 hours on silicon. In the corresponding test case they are exhaustively testing a multi-channel data arbiter which is part of a larger signal processing module. The test case selects two random channels of the arbiter and configures them with randomized settings. After configuration, the corresponding input data processing gets enabled to ensure input data to the arbiter channels. The arbiter collects the data in a FIFO and writes them into a target memory address by using an AXI manager. For every IP configuration eventually some output data is expected at the target memory address. In the mentioned test case, the DUT suddenly stops sending data. The following constraints are provided:

1. The suspicious module, the multi-channel data arbiter got already identified by the post-silicon validation team. The arbiter got verified by using a constrained-random UVM testbench and the verification activities passed already the sign-off.
2. The arbiter has 7 channels in total. Two random channels are enabled sequentially. The bug got observed with different configurations, e.g. channel 0 and 1, or channel 1 and 3. There seems to be no relation to any specific channel selection.
3. All control registers are known and valid.
4. Continuous data input is ensured on all configured input channels of the arbiter.
5. The data output interface of the arbiter is using the AXI protocol. The connected subordinate is ready to accept data on both AXI channels for address and data. The AWREADY and WREADY flags are asserted.

B. Environment Build for FPV

The second step after collecting all information about the unexpected behaviour is the setup of the FPV testbench. The multi-channel arbiter gets instantiated as DUT and all inputs and outputs are connected properly. The arbiter has an APB control interface to access internal registers, multiple custom data input interfaces and an AXI interface to write out data into a memory. Figure 1 shows the DUT with connected ABVIPs for the APB and AXI interface and additional fairness constraints on the input data ports to ensure a continuous input data flow. Without adding fairness constraints the formal tool could create a deadlock just by not sending any input data. The ABVIPs add assumptions to guarantee protocol compliant behaviour on the APB and AXI interface.

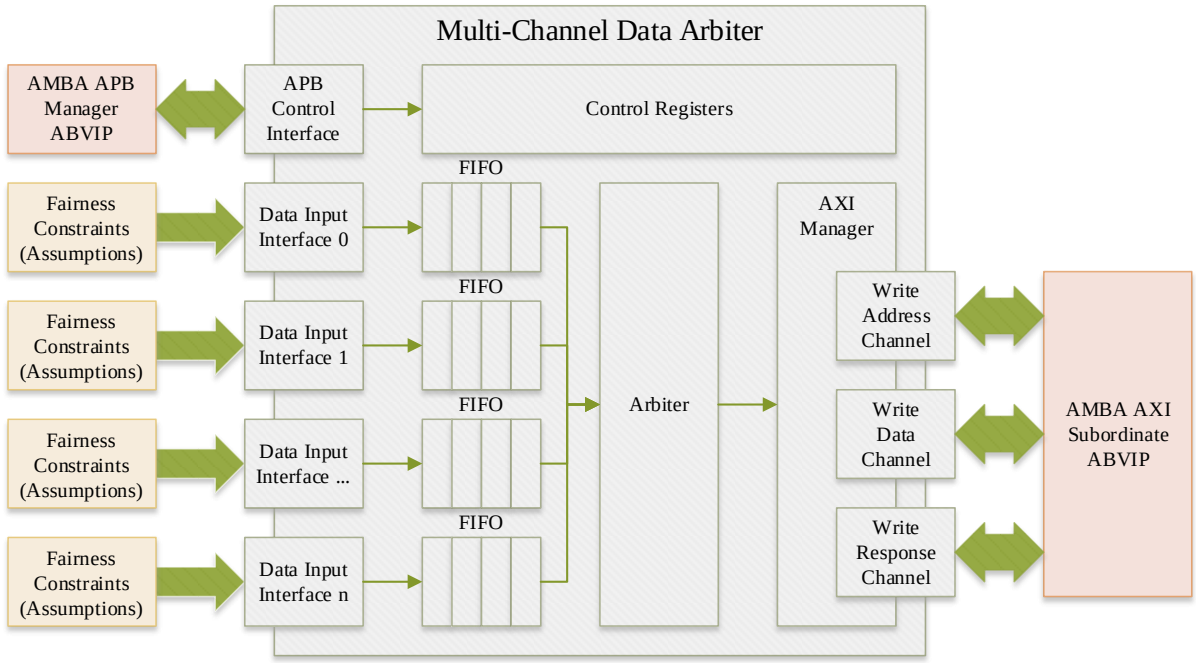

Figure 1: Design Under Test with attached ABVIPs and fairness constraints

C. Nailing down the property

Based on the post-silicon validation finding the SVA cover property can be implemented. To enable the maximum readability of these SVA properties, we take advantage of the SystemVerilog ‘let’ construct to create alias variables to avoid long and unreadable properties. Figure 2 shows the implemented auxiliary code for one channel of the arbiter. This gets duplicated for other channels as well. The configuration registers can be accessed directly via cross references into the DUT, as we don’t care how the registers are getting configured, we just want to ensure correct internal values in our formal trace. Two additional variables per channel are implemented to ensure the expected settings and to check whether they remain stable.

```
// Helpers to access internal DUT signals
let ch0_mode = DUT.<path_to_mode_control_register>;
let ch0_packet_size = DUT.<path_to_size_control_register>;
let ch0_axi_burst_len = DUT.<path_to_burst_len_control_register>;
let ch0_axi_target_addr = DUT.<path_to_target_addr_control_register>;

// Validation settings
parameter CH_PACKET_SIZE       = 24'd300;
parameter CH_AXI_BURST_LEN     = 8'h03;
parameter CH_AXI_TARGET_ADDR_0 = 32'hAFFE_0000;

// Arbiter Mode per Channel (Enabled / Disabled / Debug Mode)
parameter CH_DEBUG    = 2'b10;
parameter CH_ENABLED  = 2'b01;
parameter CH_DISABLED = 2'b00;

// Helper to ensure validation settings
let ch0_cfg_matches =
  (ch0_packet_size == CH_PACKET_SIZE) &&
  (ch0_axi_burst_len == CH_AXI_BURST_LEN) &&
  (ch0_axi_target_addr == CH_AXI_TARGET_ADDR_0);

// Helpers to ensure stable settings (mode excluded intendedly)
let ch0_control_regs_stable =
  $stable(ch0_packet_size) &&
  $stable(ch0_axi_burst_len) &&
  $stable(ch0_axi_target_addr);
```

Figure 2: Auxiliary code to prepare the property implementation for channel 0

Based on the prepared auxiliary code the cover property shown in Figure 3 got implemented. Out of the seven possible channels we selected only the first two to be activated. The property is splitted into three phases to ensure proper configuration with respect to the post-silicon validation findings and eventually ends in the deadlock situation:

- Phase 1
  - In the first phase we want to ensure that the two selected channels are disabled and the configuration matches. This ensures that the correct settings are used when a channels gets enabled, as the IP samples the control registers in the cycle where the channel gets enabled.
- Phase 2
  - In this phase the formal tool gets the freedom to do whatever is needed to reach phase 3. We have no knowledge about the number of cycles which are required to reach the next phase, therefore we keep it open ended. The formal tool is configured to find the shortest possible path to reach the target state. The only constraint is a stable configuration of channel 0 and 1.
- Phase 3
  - In the third phase, we want to see that the first two of total seven channels are activated and the remaining channels are inactive. In parallel, there shall be no AXI traffic even when the subordinate signals are ready. It is important to understand that we don't expect both channels to be started at the same time. The described scenario shall hold true for a configurable number of clock cycles, the DELAY. We started with a DELAY value of 50, but after initial tool runs we figured out that 50 cycles are too less due to internal FIFOs, which are still able to buffer data. Therefore, we added a generate loop to create multiple versions of the property with different delay cycles to sort out valid behaviour from a true deadlock.

```
parameter MAX_DELAY = 500; // Started with 100

generate
  for (genvar DELAY=50; DELAY < MAX_DELAY; DELAY=DELAY+50) begin
    show_me_the_deadlock : cover property (
      @(posedge axi_clk) disable iff (!axi_rst_an)
      // Phase 1: Config must match once when channels are disabled.
      //          Register settings are sampled when channels gets enabled.
      (
        ch0_mode == CH_DISABLED && ch0_cfg_matches &&
        ch1_mode == CH_DISABLED && ch1_cfg_matches
      )
      ##1
      // Phase 2: Control regs must be stable, we don't care what else happens
      (
        ch0_control_regs_stable &&
        ch1_control_regs_stable
      ) [*1:$]
      ##1
      // Phase 3: Two of seven channels are enabled, configuration remains stable.
      //          No traffic on AXI, subordinate is ready.
      (
        ch0_mode == CH_ENABLED  && ch0_control_regs_stable &&
        ch1_mode == CH_ENABLED  && ch1_control_regs_stable &&
        ch2_mode == CH_DISABLED &&
        ch3_mode == CH_DISABLED &&
        ch4_mode == CH_DISABLED &&
        ch5_mode == CH_DISABLED &&
        ch6_mode == CH_DISABLED &&
        !axi_m_awvalid && !axi_m_wvalid &&
         axi_m_awready &&  axi_m_wready
      ) [*DELAY]
    );
  end
endgenerate
```

Figure 3: Single cover property to describe the expected target state

D. Regulate the environment assumptions

To ensure continuous input data on the custom data interface, Figure 4 shows the implemented fairness assumptions for channel 0. These constraints are replicated for the remaining six channels. The assumptions are designed in a way that the data rate can be dynamically chosen by the formal tool, but always stays inside a lower and upper bound. The lower bound guarantees that the deadlock cannot be caused by missing input data, and the upper bound guarantees not to violate the specifications of the IP.

```
parameter SHIFT_REG_SIZE = 12;
parameter MIN_VALID_CNT  = 1; // Min datarate: 1 of 12 cycles valid
parameter MAX_VALID_CNT  = 4; // Max datarate: 4 of 12 cycles valid

bit [SHIFT_REG_SIZE-1:0] valid_0_reg;

always @(posedge sample_clk) begin // channel 0
  valid_0_reg[0] <= DUT.p_data_valid0;
  valid_0_reg[SHIFT_REG_SIZE-1:1] <= valid_0_reg[SHIFT_REG_SIZE-2:0];
end
```

```
assume_data_valid_0 : assume property (
  @(posedge sample_clk)
  $countones(valid_0_reg) >= MIN_VALID_CNT && $countones(valid_0_reg) <= MAX_VALID_CNT
);
```

Figure 4: Fairness constraints for custom data input valid signal of channel 0

E. Proof Results

Table 1 shows all the generated property variants and their corresponding runtime. In total nine variants of the property with different delay cycles got generated. It takes the formal tool 4 seconds to generate the first cover trace and 203 seconds to generate the longest trace. Up to a delay count of 200 cycles, the internal FIFOs are still able to buffer data, which is a valid behaviour and does not reflect the deadlock. The first trace which re-produces the deadlock starts at 250 delay cycles and 73 seconds tool runtime. Figure 6 shows an exemplary trace of the deadlock. It can be observed that channel 1 gets activated first, and some cycles later, the second channel 0. After ~150 and ~240 cycles both FIFOs are full and no data gets written via the AXI interface. A thorough RTL review pointed us to a lately introduced arbiter optimization for the single channel mode. The design engineer had not expected that this optimization is also active during the start-up of multiple channels. The formal tool hits exactly one critical cycle, where enabling of the second channel corrupts the arbiter logic. Fortunately, a workaround in the control flow can be used to avoid this corner case. This workaround got tested with additional assumptions and the formal tool was not able to generate traces anymore with delay cycles of 250 or more.

Table 1: Runtime of generated properties

Generated Property #	Delay in Cycles	Proof Runtime in Seconds	Deadlock found
1	50	4	No (Internal FIFOs still able to buffer)
2	100	44	No (Internal FIFOs still able to buffer)
3	150	48	No (Internal FIFOs still able to buffer)
4	200	57	No (Internal FIFOs still able to buffer)
5	250	73	Yes
6	300	96	Yes
7	350	126	Yes
8	400	162	Yes
9	450	203	Yes

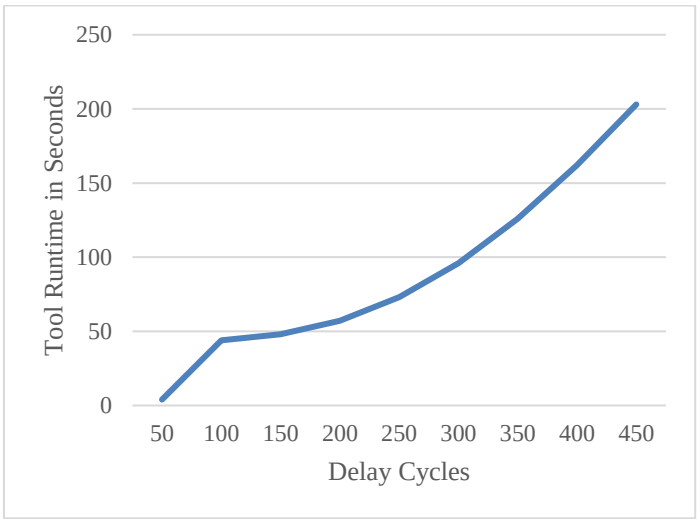

Figure 5: Tool runtime to generate a trace for different delay cycles

I. LIMITATIONS

The UNEARTH approach to re-create post-silicon bugs using FPV requires a good understanding of how formal traces for cover properties are being derived. As every – additional - cycle can dramatically increase the runtime to generate the trace, it requires a thorough understanding how to describe the shortest path to the faulty behaviour. Figure 5 shows the non-linear tool runtime to generate a formal trace over linear increasing delay cycles.

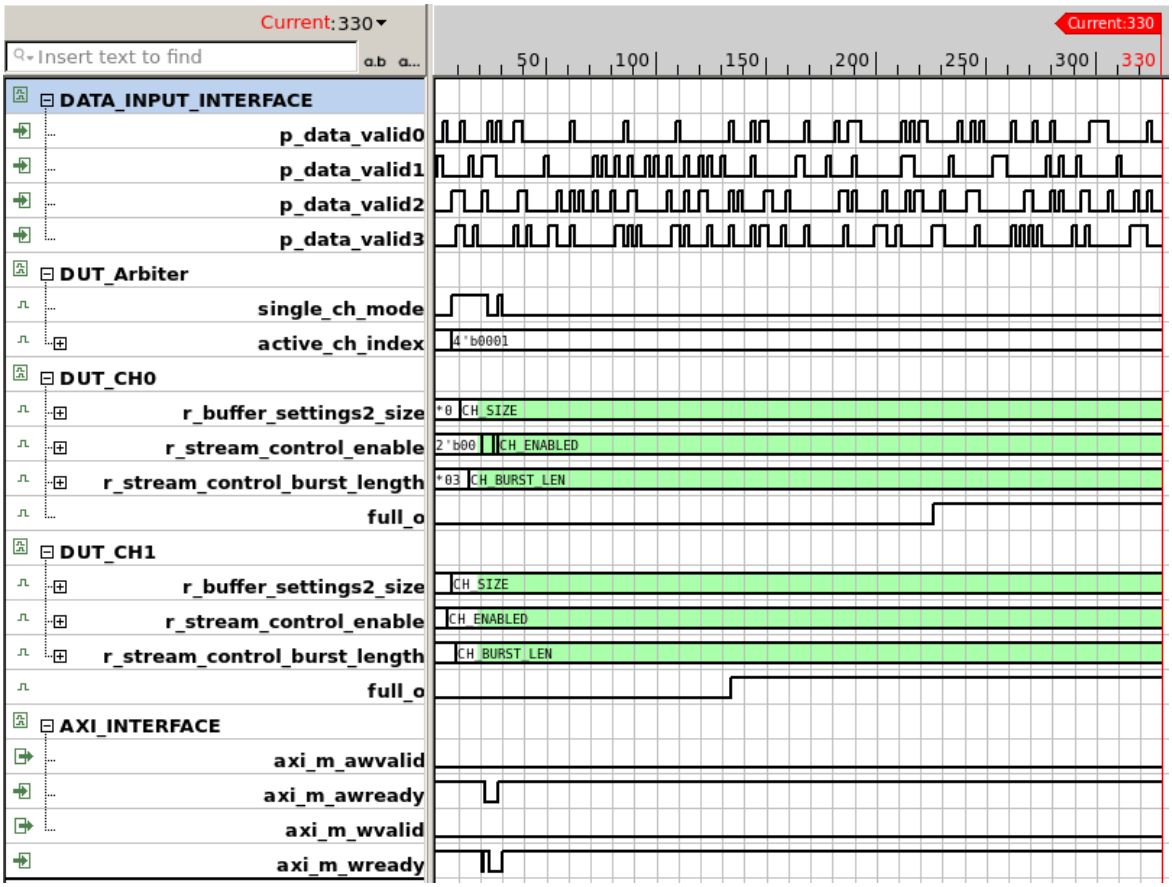

Figure 6: Formal trace of the deadlock with full FIFOs and no AXI traffic

II. SUMMARY

In this paper we demonstrated that FPV is highly suitable to analyze and debug post-silicon issues. By following the seven steps of the UNEARTH approach, bugs can be re-created in a structured way. We were able to catch a rare corner case deadlock in a multi-channel data arbiter which escaped the pre-silicon verification. A single cover property is sufficient to generate the formal trace of the deadlock. It took the formal tool just 73 seconds to re-create the deadlock. Based on this trace, we were able to identify the buggy RTL and successfully tested a workaround to avoid the deadlock in the final silicon. This greatly shows how powerfully formal can be used to re-create, analyze and fix deadlocks and other kinds of functional bugs.

III. REFERENCES

[1] J. Kasak, "Accelerating Post-Silicon Debug with Formal Verification," in *JUG*, 2018.

[2] Jan Hahlbeck, Steffen Löbel, Chandana G. P., "Towards a Hybrid Verification Environment for Signal Processing SoCs," in *DVCon Europe*, 2023.

[3] Cadence, "Jasper FPV App," [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html. [Accessed June 2024].

[4] Erik Seligman, Tom Schubert, M.V. Achutha Kiran Kumar, Formal Verification: An Essential Toolkit for Modern VLSI Design, Morgan Kaufmann, 2023.

[5] A. G. A. K. V. M. B. S. S. S. K. S. G. N. Anshul Jain, Never too late with formal: Stepwise guide for applying formal verification, DVCon, 2022.

[6] H. Foster, "Functional Verification Study," Wilson Research Group, 2022.