Every Cloud – Post-Silicon Bug Spurs Formal Verification Adoption

Blaine Hsieh
Faraday Technology Corp.
Hsinchu City, Taiwan
blaine_h@faraday-tech.com

Stewart Li
Mentor Graphics Corp.
Hsinchu City, Taiwan
stewart_li@mentor.com

Mark Eslinger
Mentor Graphics Corp.
Fremont, CA 94538
mark_eslinger@mentor.com

***Abstract*- Formal verification has many useful applications in System on a Chip (SoC) design and verification flows, and had not been adopted in the verification flow of our recent chip. Unfortunately we experienced a post-silicon bug. This led us to consider formal verification and as a first step reproduce the post-silicon bug using formal techniques. This paper will detail our experience applying formal techniques to our problem and the benefits of using formal verification. Furthermore, how we continued to apply formal verification to our subsequent project and the improvements in overall verification we observed.**

I. INTRODUCTION

It's the news everyone hopes ***not*** to receive. Their verification project, the one that they've been meticulously laboring over for months, has come back from the fab and a silicon bug has been found. Then the introspection begins. How could this happen? What can we do differently in the future? That's exactly the position we found ourselves in a little over one year ago. A post-silicon bug was discovered in a DDR3 controller block. The design had been extensively verified with advanced techniques, including constrained-random simulation and a coverage-driven verification methodology. But a unique corner case bug had escaped verification and was seen on silicon in the lab. A sequence of write commands to specific memory bank and row combinations would cause a DDR3 protocol violation related to pre-charge timing. That was the turning point at which we started to explore formal verification. We decided to put formal to the test on the buggy version of the DDR3 design. We wrote the properties to capture the bug scenario on the DDR3 bus and constrained the design's other interfaces with the appropriate assertion protocol monitors. Then we ran Questa Formal and employed some advanced techniques to get to deep design states and reproduce the bug. Using this same approach, we were able to verify the bug fix. The DDR3 design exercise proved to us that formal verification had the tool capacity and methodology to handle the complexity of our designs and it allowed us to become familiar with the use model. Simulation has been the mainstay of our RTL verification methodology for decades and that's not expected to change any time soon. However, we then realized that formal verification could also play a part in our strategy. After seeing the ability of formal to reproduce our post-silicon bug, we then continued to roll it out as part of our standard verification process.

Since then we have deployed formal on multiple projects with outstanding results. In this paper, we plan to present our approach to deploying formal to complement traditional simulation based methods. We've used formal in both bug hunting and design assurance applications [1]. For the bug hunting application, we will show how formal verification enables us to find bugs earlier and easier. In design assurance applications, we'll describe how we can improve verification efficiency, while boosting the confidence in design quality for certain blocks.

II. THE POWER OF FORMAL VERIFICATION

As this paper is being written, formal verification is a fairly standard flow for many companies. Even so there are still verification teams and engineers where its usage is not fully utilized and/or understood. A brief overview will be given to ensure everyone is on the same page with techniques which will be described later in this paper. Formal verification is another tool which can be effectively used to minimize bugs in a design and help ensure the highest quality silicon goes out to the end customer. Like any form of verification formal has its strengths and weaknesses. Most engineers are familiar with simulation and its application in their verification flow. If we look at any design under test (DUT) we find there is a state space which describes its behavior. If you compare the strengths and weaknesses of formal and simulation you'll find that they dove-tail nicely, making these techniques complimentary with one another as explained below.

Simulation is a vector based technology which given an initial state will carve a path through the state space of a design using a depth-first approach. The strength of simulation is that you can describe virtually any behavior and explore deep into the state space of the design. The weakness of simulation is that there isn't enough time to fully explore the state space of the design. This is one of the main reasons functional bugs end up in silicon. This can be illustrated using a trivial example. Consider a 32 bit comparator. To exhaustively check the functionality of this comparator you would need to run 2^{64} vector combinations through it. Running a vector per ns through your "fast" simulator it would take well over 500 years to get through all combinations. Obviously designs are much more complex than this. Proper test planning can mitigate the effect and it illustrates the point that no matter how much planning is done, it is impossible to cover every scenario.

Formal verification is a technology which uses mathematical methods to do an exhaustive analysis using a breadth-first approach. The strength of formal methods is that given an initial state, it can explore all possible input combinations and states of a design under test (DUT) and ensure the desired behavior. This allows formal to find corner case bugs which are often missed by simulation. The weakness of formal is that due to its mathematical nature, the problem is geometric in nature. The state space can become to big which causes the algorithms to reach a point where further progress isn't possible or the memory resources of the machine the software is running on have been maxed out. One of the ways to mitigate the state space issue will be described below.

Formal verification is predicated on the usage of Properties which can be asserted, assumed, and covered. System Verilog Assertions (SVA) is a standard assertion language which we used in our flow. These properties can describe both expected and unexpected behaviors of the design. This is one of the ways to apply formal to post-silicon bugs: describe the bug behavior you are seeing, write a property which says it can't happen, and let formal show you how it can. Since formal verification doesn't require vectors it can be used early in the design and verification flow. It can find the corner case bug states that may slip through directed or even random testing. Formal verification can be applied in 3 primary ways (the ABCs of formal):

A) *Assurance*[2]
   The main goal is to "prove" all properties. Once proven, there is no input stimulus which can violate the behavior of the design with respect to the property. Formal results can take one of 3 possibilities, a proof, a counterexample, or an inconclusive [3].
B) *Bug Hunting*
   This is an artifact of assurance and can be the sole objective as well. Post silicon bug hunting also falls in this category. The goal is to find and fix as many bugs as possible.
C) *Coverage Closure*
   The goal is to determine if a coverage statement/bin/element is reachable or unreachable.

When using formal verification, as mentioned above, the state space of the DUT can exceed the capacity of the methods being used. The way this usually exhibits itself is that the cycle depth of analysis will begin to slow down as it takes longer and longer to go a cycle further into the design. The limit reached may be 10's or 100's of cycles depending on the type of problem being explored. If you are doing a post-silicon bug hunt and you know the bug is beyond the cycle depth you have analyzed to, other steps need to be taken. When this happens various techniques can be used to minimize the state space to allow deeper exploration of the design. While this paper won't cover all these techniques, we'll cover one called "goal posting" as shown in Figure 1 below.

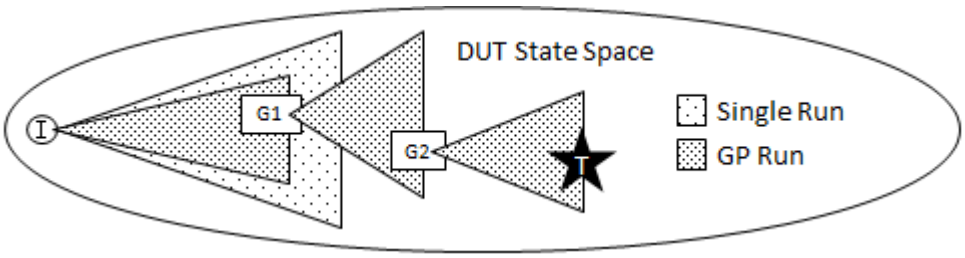

Figure 1. A goal-posting formal run compared to a single initial formal run to hit a desired target.

If we can find a cover point which is in the depth of analysis that formal can hit then we can use that point as an initial state from which to explore deeper into the design. In Figure 1 above the initial formal run only

analyzed so far into the design and was not able to reach the final target. Intermediate goals, typically specified as a cover statement, were selected which allowed multiple formal runs to explore deeper into the design and hit the desired target. Using this technique a user can leap frog from point to point exploring deeper into the design until the final goal is achieved. This is an excellent bug hunting technique, especially for post-silicon bugs.

III. Formally Reproducing the BUG On our DDR3 design

DDR3 designs are highly configurable and complex from a verification standpoint. Proper initialization is needed to configure the design for proper operation. The serial nature of the design also makes it more challenging for applying formal techniques. Part of the exercise was to use normal assertion based techniques in reproducing the bug to build internal expertise and give us a good idea of traditional formal usage.

A. DDR3 Basics

DDR3 memory controllers are complex designs from a design/verification standpoint. Initialization is a challenge in that there are many configuration registers to be programmed to setup the design for proper function. Normal style DRAM signals are used: CKE, CS[1:0], RAS, CAS, WE, BA[2:0], A[15:0]. A typical write (burst 4) to precharge transaction is shown below in Figure 2:

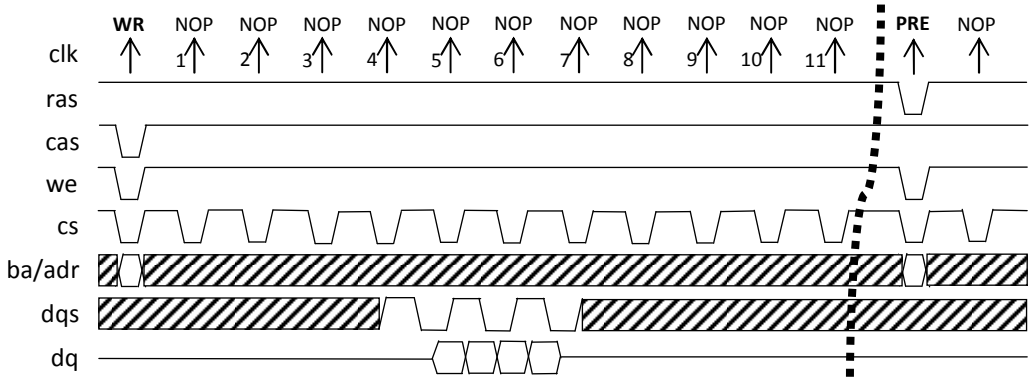

Figure 2. A DDR3 write to precharge timing diagram[4]

The operations are typically described using a truth table. Some of the basic functions are shown below in Table 1:

Table 1
DDR3 Partial Command Truth Table [4] (Note: H = Logic 1, L = Logic 0)

Function	CKE (prev.)	CKE (current)	CS#	RAS#	CAS#	WE#
Bank Precharge(PRE)	H	H	L	L	H	L
Bank Activate (ACT)	H	H	L	L	H	H
Write (WR)	H	H	L	H	L	L
Read (RD)	H	H	L	H	L	H
NOP	H	H	L	H	H	H

Based on these commands, access to the DRAM is achieved. There are also timing considerations which must be met between the various access types and when refreshes happen depending on the part and technology. There is timing specifications related to RAS and CAS latency as well as other timing configurations which are typically specified with respect to clock cycles. This type of timing check was where some bugs got through our verification process.

B. DDR3 Design Setup

Initially we ran on the whole DDR3 design. This design has two complete DDR3 output busses (DDR3_0 and DDR3_1) with dual chip selects. The configuration registers are programmed through an APB interface. There are 2 AHB slave interfaces and 6 AXI3 slave interfaces. Picking the level of hierarchy you run at can make a difference in ease of constraining the design. We used formal IP to constrain the AHB and AXI interfaces. When doing a targeted bug hunt minimization of the state space will make the formal runs more efficient. We disabled 4 of the AXI interfaces and ignored one of the DDR3 output interfaces since it was a duplicate of the other one. We also disabled

the APB interface so that the design could not be reconfigured during the formal run. We focused all our attention on the post-silicon bug hunt target, so all decisions and actions were toward that end.

C. DDR3 Bug

The initial post silicon bug we looked at involved a write to precharge timing violation. For this particular design it is required to wait at least 11 cycles after a write to execute a precharge. In our design we were witnessing a scenario where the precharge was happening 7 cycles after a write command to the same bank when looking at the DDR3 interface. Figure 3 below shows the timing of the write to the precharge of the same bank address from the formal firing (note that there were other transactions which had happened before this scenario which aren't shown):

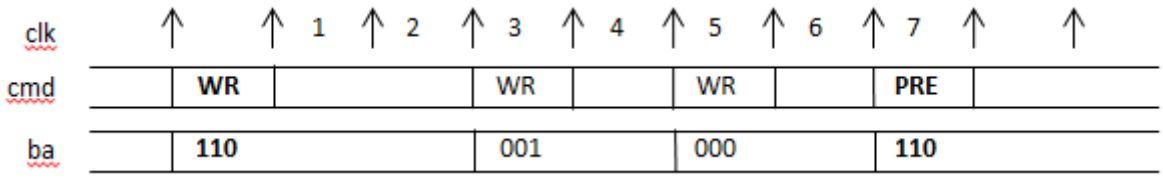

Figure 3. Waveform showing the violation of write to precharge timing (should be greater than 11 clock cycles).

Subsequently more work was done on a sub module of the design as described below on continuation of this analysis. The bug was triggered when the following internal conditions happened in the design:

1) tWL+r+tWR >= 15 in DDR3 mode
    a. for this case tWL+4+tWR = 7 + 4 + 9 = 20
2) At least 4 consecutive write commands with different bank addresses happened.
    a. For example bank 0, 1, 2, 3
3) After 4 write commands, another bank "B" write command with a different bank from the previous 4 write commands is pushed into the DDR command queue when the command queue is empty.
4) Another command with bank "C" is pushed into the command queue, and bank C is the same as one of the previous 4 write commands but with a different row.

As you can see the corner case scenario found with formal is non-trivial. As stated earlier, the number of variations of commands that can happen is more than what most verification teams can test in the time allotted to them using traditional simulation based methods. This is true no matter how sophisticated the simulation environment is.

D. DDR3 Initialization for formal

One of the keys to applying formal techniques is initialization of the DUT. Your proofs are only as good as the starting state. Normally this is the traditional power on reset state which is accomplished by asserting the reset signals for some period of time. For the DDR3 design one must go many steps further in that configuration registers need to be written to. The process of initialization of a DDR3 design can take 1000's of cycles. This is typically too complex to do in the normal formal initialization that happens during a formal run. Since we had simulation data available we decided to leverage the heavy lifting the simulation team had done to initialize the design. We were also fortunate in that the initialization sequence employed in the design has a signal, init_ok, which is asserted once the initialization of the design is complete. It was then a simple matter of viewing that signal in the waveform viewer and finding the time stamp when it was asserted. We then used the waveform file and that time stamp to initialize the DUT for running formal. This technique can also be used for doing bug hunts deeper into the design than you might normally achieve from the standard "power on reset" initial state.

E. Assertions used for the bug hunt

Standard SVA and modeling layer code were used with a bind statement to connect to the DUT for this bug hunt. The assertions used for the formal check were very simple. When writing assertions for formal it is best to keep them short and simple. In addition, modeling layer code is heavily used to make the writing of assertions easier to read and maintain as well as to keep them simple in nature. We used renamed logic for command decoding and for use in the modeling layer code. Some of the command decoding logic is shown below in Figure 4.

```
parameter PRECHARGE = 7'b11_0010_0;              // precharge truth table values
parameter READ      = 6'b11_0101;                // read      truth table values
parameter WRITE     = 6'b11_0100;                // write     truth table values
parameter ACTIVE    = 6'b11_0011;                // active    truth table values
reg pre_cke;
always @(posedge ddr_clk) pre_cke <= ddr_cke;    // create delayed cke value
wire [6:0] ddr_cmd = {pre_cke,ddr_cke,ddr_cs,ddr_ras,ddr_cas,ddr_we,ddr_addr[10]};
wire precharge     = (ddr_cmd      == PRECHARGE); // precharge command
wire active        = (ddr_cmd[6:1] == ACTIVE);    // active    command
wire write         = (ddr_cmd[6:1] == WRITE);     // write     command
wire read          = (ddr_cmd[6:1] == READ);      // read      command
```

Figure 4. Modeling layer code to simplify the assertions and command transactions.

With post-silicon bug hunts there is the advantage that you know what the bug is and can write a very specific target assertion to catch it. In the more general sense of bug hunting you don't know if a bug is there are not and have to rely more on functional specifications of the design. Once the bug has been isolated and the block/design in which it resides has been determined a property which describes the bug scenario can be written, then specify that the scenario can't happen and let formal find a way that it can. In this case we knew the requirement was a minimum of 11 cycles from a write to a precharge so we could write the property per that spec. The knowledge that there was a bug that violated that specification, we expected the assertion would fire. What we didn't know was what the scenario was that would cause this bug condition to happen. It hadn't occurred in all our simulation scenarios. This is where the power of formal can be used to find these corner cases. Non-determinism (ND) is a mathematical term used to describe various behaviors with formal algorithms. It is basically a fancy way of saying: "let formal figure out what to do". We know there is a write to some bank and a precharge to that same bank that violates the timing requirements. That is what we want to check. What we don't know is what transactions have happened before that sequence or even if it makes a difference what bank is used. We'll use ND and let formal pick when to start that sequence and what bank address to use. Initially we set things up and ran with the assertion we had written. The result was inconclusive. We then employed the goal-posting technique described above to help formal out a bit. In these scenarios there is always some experimenting that goes on before you arrive at the final solution. In the normal run there were too many scenarios. In the goal-posting run we found that once an active command had been issued, from that point we could fire the desired target. This employed a 2 step goal-posting run. One formal run to get to the first active command, then using that as the new start point a 2nd formal run to provide the full counterexample of the bug. Figure 5 below shows the modeling layer code to keep track of the write formal picked and the cover statement and final target assertion that was used:

```
// Let Questa Formal pick the wr cmd variables, when and what bank addr
wire start;                          // ND: formal picks when to check
reg my_wr;                           // record when the write happens
reg [2:0] my_ba;                     // ND: formal picks the bank address
// precharge with same bank addr as write
wire same_pre = precharge && (ddr_ba_addr == my_ba);
always @(posedge ddr_clk or negedge reset_n)
if (!reset_n) begin
        my_wr <= 1'b0;
        my_ba <= 3'b000;
end else
        if (start && write && !my_wr) begin
                my_wr <= 1'b1;
                my_ba <= ddr_ba_addr;
        end else begin
                my_wr <= my_wr;
                my_ba <= my_ba;
        end

// First goal (active command)
cov_gp: cover property (@(posedge ddr_clk) !my_wr && active );

// final bug target (write to precharge timing violation)
a_wr_to_pre_bug: assert property (@(posedge ddr_clk)
                                        $rose(my_wr) |-> (!same_pre)[*11] );
```

Figure 5. Modeling layer code and the cover statement and final target assertion used in the DDR3 post-silicon bug hunt.

For the above scenario, from the first active command formal can do any number of commands, at some point it will decide when to "start" the check. When it chooses to start the check that fact is captured in the "my_wr" register and the bank address is captured. The assertion starts checking on the rising edge of "my_wr". If there is a precharge to the same bank address within the 11 cycle requirement the bug is found. For this bug it happened at 7 cycles.

F. Formal Results

As the design is being setup for the formal run, initial constraints and design initialization are put in place. One of the tests for determining if the setup is correct is if various commands can be executed on the DDR3 outputs. The bug property was first run starting from an initial state and formal was able to get 22 cycles into the design. At this point the assertion was still inconclusive and the process killed as it wasn't making much progress past that point. With other experiments it was determined it took 13-16 cycles to get to the first write command. If you add 11 cycles to that it would require progress into the design in the range of 25-30 cycles to cover the bug scenario. This is when we made the decision to use goal-posting. An active command is needed before any read or write can happen. We played around with various scenarios and determined our first goal being the first active command. From this point formal was able to apply a sequence of AHB/AXI transactions which caused a series of active and write commands before issuing the write command which caused the counterexample reproducing the bug seen in silicon. The counterexample was 38 cycles deep (8 to the 1st goal active command and 30 to the assertion firing). The whole process of reproducing the post-silicon bug completed in under an hour.

G. Continuation of project

The post-silicon bug hunt was done on the whole design initially. The block, shown in Figure 6 below, which contained the bug was later extracted and additional formal runs were done at this level. This replicated scenarios where a designer or verification engineer would apply formal to the block in a more traditional verification environment.

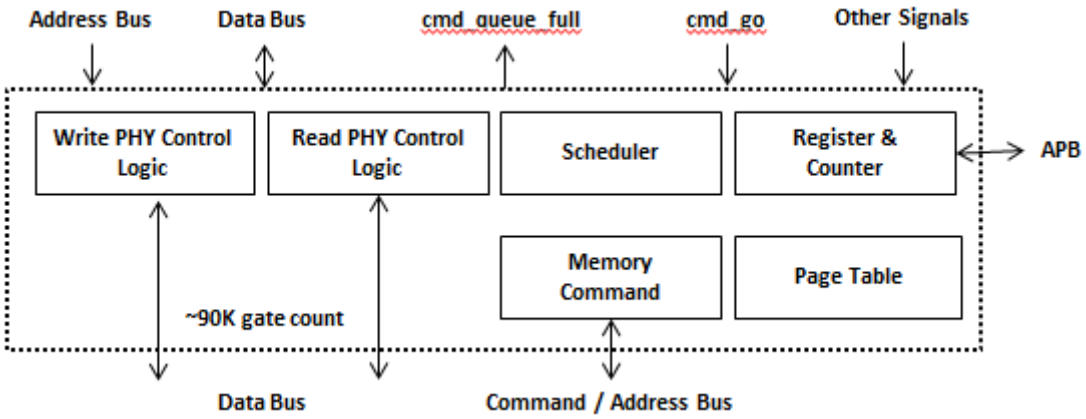

Figure 6. Block diagram of extracted buggy module

This setup required more traditional constraint and initialization as the interfaces were custom internal busses/signals. The address (row, bank, column) and data busses were left random. The cmd_queue_full and cmd_go signals were constrained with SVA such that if the queue is full there is no "cmd_go" signal. The other signals, many of which were data from the configuration registers were tied off to 0/1 as appropriate. Assertions were added to check that the output commands conformed to the DDR3 spec, specifically to check another timing issue and a re-ordering issue which had been seen. The timing issue required goal posting and had a firing of 71 cycles in 1.5 hours. The other bug was fired in minutes at a depth of 12 cycles. We could then also use this same setup to test the rtl code with the bug fixes (the 1st bug fix we had a bounded proof and the 2nd one a full proof). With these successes under our belt we decided to roll formal out on our next project on several blocks.

IV. Applying Formal on Other Blocks For Assurance and Bug Hunting

With our success at applying formal to the DDR3 design we decided to roll formal out on other blocks in subsequent projects. We'll detail our experience on 3 of those blocks below.

A. CPU BIU module

The Bus Interface Unit (BIU) is an internal module of the CPU. It accepts requests for memory accesses from the CPU core and executes instructions and data accesses through the external system bus. Figure 7 below shows a block diagram of the BIU module:

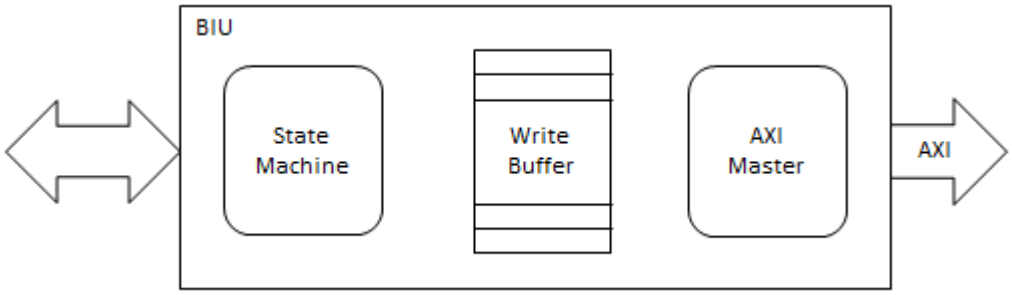

Figure 7. Block diagram of the BIU module

Each of the interfaces is either an AHB or AXI interface. Since the project schedule prevented running formal at the whole CPU level we decided to focus our efforts on the BIU module. Below is an example of some of the checks that we did on the BIU.

In the BIU there is a write buffer. It buffers the write data and feeds the data to AXI master. Under certain special conditions it is possible for the AXI bus to be deadlocked. For example, the logic controlling the write buffer doesn't handle the full signal properly so the buffer can overflow and cause the AXI bus to be stuck without a BReady signal.

We built the environment for the property checkers and modeling layer code. We can then verify all possible cases to ensure there are no side effects in the BIU module. Below in Figure 8 are two examples of assertions checking for AXI transaction timing and prevention of overflow of the write buffer. Signal definitions are shown below:

Signal Definitions:

BValid/BReady	: AXI response signals
WBfull	: Full status of internal write buffer
dc2_regW	: Write enable of internal write buffer

```
bch_halt_bug1: assert property (@(posedge AClk) disable iff (!AResetn)
                                        $rose(BValid ) |-> ##[0:1] BReady       );
bch_halt_bug2: assert property (@(posedge AClk) disable iff (!AResetn)
                                                  !(WBfull && dc2_reqW)      );
```

Figure 8. Some of the assertions used in verifying the BIU module

B. Interrupt Controller

The Interrupt controller is a centralized resource for supporting and managing interrupts in a system that includes at least one processor. The interface for programming the register file is an AXI interface. Figure 9 below shows a block diagram of the Interrupt Controller:

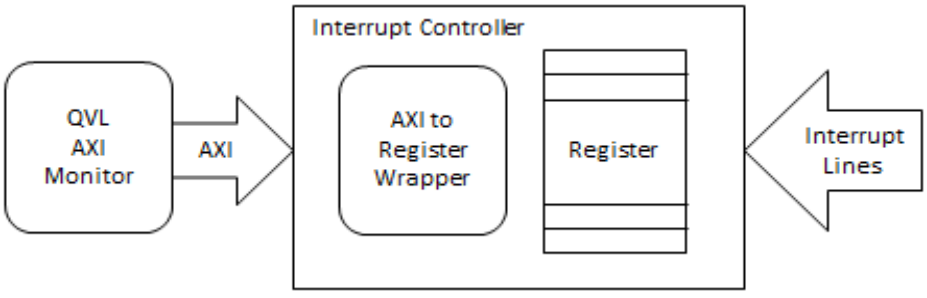

Figure 9. Block diagram of the Interrupt Controller module

When running formal we observed that an AXI write action will fail if there is a wait state during the write operation. When the AXI master inserts a wait cycle, the address is updated to the next address (offset=A+4). This means the write data and address are not in sync (Address[A+4]=DataA). This causes the data to be written to the wrong address. This was not a situation which had originally been accounted for.

We used the QVL (Questa® Verification Library) AXI monitor to find bugs and then validate them until all assertions were proven. Below in Figure 10 are two example assertions for checking the read and write operation of the configuration/status registers.

Signal Definitions:

W_active/R_active	: Active signal of AXI write/read combination
decode_addr_w/decode_addr_r	: Internal signal of decoded address
active_waddr/ active_raddr	: Expected write/read address

```
compare_waddr : assert property (  @(posedge AClk) disable iff (!AResetn)
                                W_active |-> ##1 (decode_addr_w == active_waddr)   );
compare_raddr : assert property (  @(posedge AClk) disable iff (!AResetn)
                                R_active |-> ##1 (decode_addr_r == active_raddr)   );
```

Figure 10. Some of the assertions used in verifying the Interrupt Controller module

C. MAC Controller

The MAC controller has AHB master capability and is fully compliant with the IEEE 802.3 100 Mbps and 10 Mbps specifications with the MII/RMII interfaces. Figure 11 below shows a block diagram of the MAC Controller module:

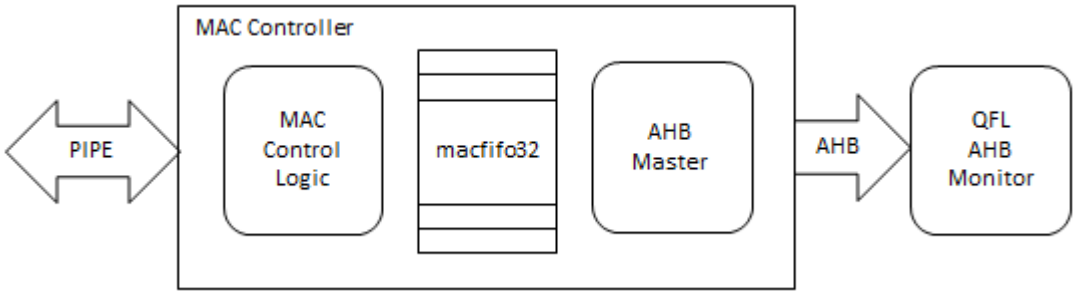

Figure 11. Block diagram of the MAC Controller module

We had a similar situation with the MAC controller that we did with the DDR3 controller in that we witnessed a bug in our FPGA. We observed an Ethernet transaction hang in our FPGA lab environment. It took us many days to find the root cause, the MAC controller AHB master issuing an illegal HTRANS signal after the AHB slave un-split the transfer. This behavior created the wrong packet on the Ethernet channel which resulted in stuck data, locking up the interface. Unfortunately our simulation flow using directed test patterns wasn't good enough to cover the corner case condition. We didn't have the time to modify the simulation environment and turned to formal due to its ability to cover the state space for reproducing the issue as well as verifying the fix.

The MAC controller is very suitable for testing with formal verification because it includes many functions like AHB master/slave, DMA, MAC TX/RX and MII/RMII. It is difficult and impossible to verify all functions even with rigorous test bench. We approached the problem from a number of angles. One was to focus on the AHB master and split it from the system. Using formal verification and the Questa Formal Verification IP (QFL) we were able to verify the AHB master interface to ensure correct behavior. We also targeted other underflow/overflow conditions within the design which were related to the problem we had encountered. Figure 12 below shows some of the assertions used in verifying the AHB master FIFO address pointer logic. Between using the QFL library and our assertion sets we were able to reproduce the bug and verify its fix.

Signal Definitions:

ahbm_fifosp	: FIFO pointer of AHB master, represents the free space of the macfifo32
ahbm_st_idle	: The idle state of AHB master
invalid_fifosp	: Checks that the range of free space is between 0 and 6, if 7 it is an overflow condition
invalid_fifosp2	: Checks that the free space stays at 6 when the AHB master is in the idle state

```
// assert fifo without overflow
invalid_fifosp : assert property (  @(posedge hclk) disable iff (!hreset_n)
                                                      (ahbm_fifosp <= 6) );
// assert fifo without underflow
invalid_fifosp2: assert property (  @(posedge hclk) disable iff (!hreset_n)
                                          ahbm_st_idle |-> (ahbm_fifosp == 6)  );
```

Figure 12. Some of the assertions used in verifying the MAC Controller module

D. Results of the above formal runs

Our experience with using formal in the above blocks has been excellent. After our experience with the DDR3 controller, extending and expanding on that experience we were able to apply formal on these 3 other designs. Table 2 below details some of the results we obtained. Two of the blocks we were able to apply formal in parallel with our simulation flows. The 3rd block we applied formal as part of a post-silicon debug and bug find/fix flow:

Table 2
Design Assurance Test Case Results

Design Block	Simulation Results	Formal Results	Time Savings (%)
CPU BIU Module	30 man-days to coverage closure	10 man-days to proven properties	66%
Interrupt Controller	20 man-days to coverage closure	9 man-days to proven properties	55%
MAC Controller	No testbench	7 man-days to proven properties	N/A

As you can see from the table, formally verifying the block saved us a fair amount of time and also gave us confidence that our designs were bug free for the functionality we were testing with our assertions. When you look at the assertions which we have used throughout the process you'll notice that they are fairly straightforward. The simplicity of these checks makes them easy to write and extend to similar situations. Even though they are simple they can uncover complex corner case scenarios. The other useful thing about writing assertions and verification of them using formal techniques is that not only can you describe expected correct behavior of the design, we could also describe undesired behavior or conditions of the design we didn't want to have happen. These descriptions of undesired behavior with assertions is similar to the method used in post-silicon debug and when proven that these conditions don't exist provide a high level of confidence that our designs will operate correctly in all conditions encountered in the field.

V. CONCLUSION

As mentioned above, finding a post-silicon bug is never fun. Looking on the positive side of things, this opened up a new verification horizon for us in exploring how formal verification could help us in finding and fixing this post-silicon bug. The success we had with using formal verification gave us the confidence to apply it in other design blocks on our next project before tape out and on another block after tape out applying techniques we had learned with the DDR3 bug hunt. All these experiences have made us believers in the power of formal to enhance our verification methodology. Not only do we find more bugs faster and output higher quality designs, it has set us on an assertion based verification path which extends to all our verification methods. The original bug hunt provided many lessons in the application of formal verification including:

1) Application of formal covering the ABCs of formal for both assurance and bug hunting
2) Keeping assertions simple and sequentially short (making them easy to understand)
    a. Using modeling code to assist in this
    b. Making it easier for others to apply in similar situations
    c. Describing both desired and undesired behavior
    d. Using predefined formal verification IP whenever possible
3) Leveraging simulation data for DUT initialization when designs have complex initialization sequences
4) Using various techniques to resolve inconclusives
    a. Formal techniques such as goal posting and other abstraction techniques
    b. Picking the level of hierarchy to run at
        i. For simplification of the state space
        ii. For simplification of constraining the DUT
5) Measuring and recording metrics of our results and experiences

Even though our original post-silicon bug was an unfortunate event, it was the root cause of us expanding our verification methodology to include formal verification. We have expanded on that initial experience and now it is a valued part of our flow. Initially the "cloud" of a post-silicon bug cast a shadow on our verification methods and now we've found our silver lining with formal verification!

REFERENCES

[1] R. Narayan, "The future of formal model checking is NOW!" DVCon paper, March 2014.
[2] H. Foster, "Planning for formal ABV success", Verification Academy, Courses: Assertion-Based Verification
[3] Questa Formal PropCheck User Guide, Mentor Graphics, November 2014.
[4] JEDEC STANDARD, "DDR3 SDRAM Standard", JESD79-3F (Revision, July 2010), July 2012