The Importance of Complete Signoff Methodology for Formal Verification

Mahesh Parmar[1] Iain Singleton[2] Geogy Jacob[3]

[1]Synopsys (India) Pvt. Ltd. B Wing, 3rd-5th Floor, Tower A, RMZ Infinity, Old Madras Road, Bangalore, 56001, Karnataka, India, mparmar@synopsys.com
[2]Synopsys Inc, 690 E Middlefield Road, Mountain View, CA 94043, USA, isingle@synopsys.com,
[3]Synopsys Inc, 690 E Middlefield Road, Mountain View, CA 94043, USA gjacob@synopsys.com

Introduction: Formal verification in the industry has been growing rapidly in the past few years as a means for ensuring correctness in the design of integrated circuits. The surge in popularity of formal can be attributed in part to two major factors: the development of user-friendly formal verification tools and the ability of formal to produce exhaustive verification results. As design complexity continues to grow, and the applications of integrated circuits become more diverse, there is an ever growing need to achieve higher confidence in design verification. Traditional verification methodologies such as dynamic simulation cannot provide an exhaustive level of confidence due to the sheer number of input combinations that need to be tested. The mathematical proofs obtained through formal model checking allow users to gain confidence that features of designs will always work as expected, however formal is not without its own challenges.

Due to the exhaustive nature of formal it is held to a higher standard than simulation-based testing. From a management perspective, investment in formal often means to them that everything has been completely tested. The successful application of formal however is only as good as the methodology used. To ensure thorough completeness of a formal verification task it is important to go through a process described here as formal signoff. The formal signoff process can be broken down into two main areas:

- dealing with bounded proofs
- ensuring verification completeness.

A bounded proof refers to an assertion, given to a formal verification tool, which in the time allotted has not been able to conclusively say whether a property will always hold true or not. It provides a depth from the initial state up to which has been exhaustively explored without any failures detected, but beyond this is unknown. Verification completeness refers to ensuring that all possible features of the design have been tested by the assertions that have been written. It also refers to checking that the results of these assertions are valid by verifying there are no overconstraints on the design. Such overconstraints are dangerous as in the presence of an overconstraint assertions may pass, masking real issues in the design. In this paper we present a methodology for a complete formal verification signoff and use a real case study to highlight the dangers of not going through this process.

The case study we present is a Branch Prediction Unit (BPU), a sub-block within the Instruction Fetch Unit (IFU) of an advanced high performance, low power microprocessor developed for the high-end embedded market. The BPU is used to reduce the branch penalty in highly pipelined processor designs. This implementation uses dynamic prediction with global history, utilizing the history of previous branches and the current branch in order to make the prediction. The BPU features a dual bank memory and each memory word can have up to 4 instructions aligned to 16 bit boundaries. The BPU supports up to 2 branches for a 64 bit word and up to four branches can be analyzed every clock cycle. The BPU receives information from the Execution Unit (EXU) whenever a branch is executed and uses this information to update branch memory. This information is used to make predictions in the future. A block diagram of the BPU is shown in Figure 1 below.

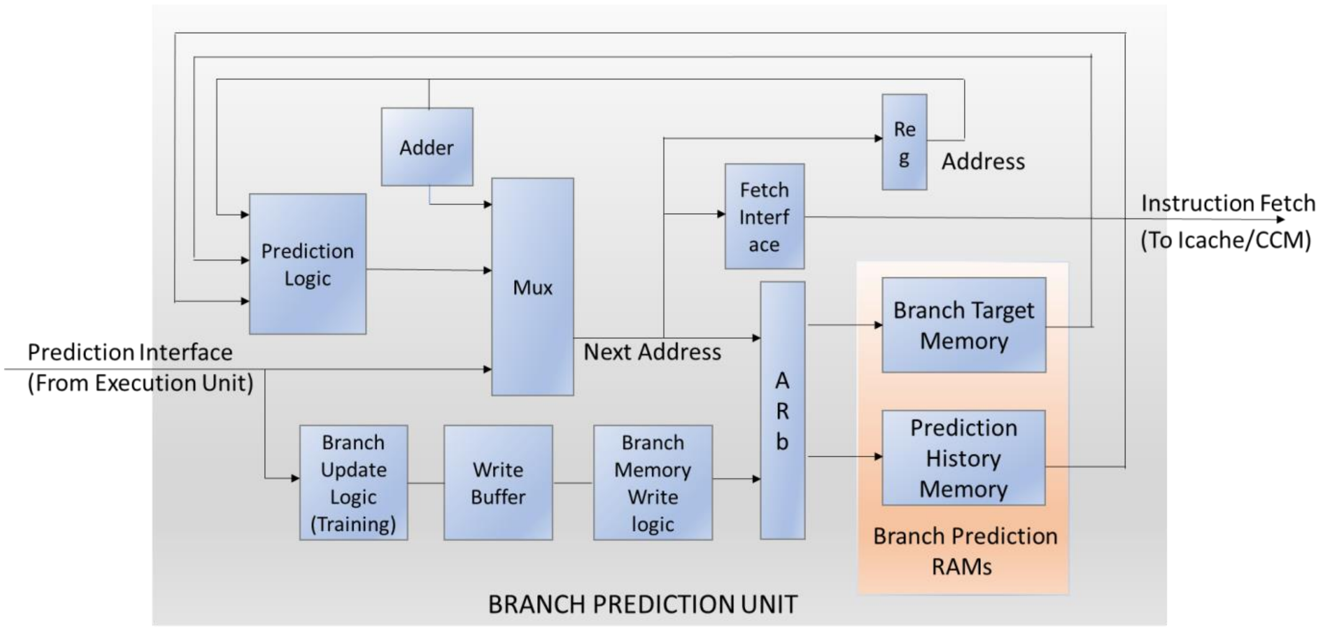

Adder
Re g
Address
Fetch Interf ace
Instruction Fetch (To Icache/CCM)
Prediction Logic
Mux
Branch Target Memory
Prediction Interface (From Execution Unit)
Next Address
A R b
Prediction History Memory
Branch Update Logic (Training)
Write Buffer
Branch Memory Write logic
Branch Prediction RAMs
BRANCH PREDICTION UNIT

Figure 1: Block Diagram of BPU

The BPU had undergone significant simulation testing but due to its complex nature, the likelihood of potential missed bugs remained high. As a result, the management decided to apply formal verification to see whether applying formal would be enough to complete verification and gain additional confidence. The initial stage of the formal work done on this was with the plan for thorough and complete verification. The closure criteria used was:

- Designer review of properties and results
- Human analysis of acceptable bounded depths and capping the formal verification exploration at this depth

While designer review is an important part of verification signoff it is not enough on its own to produce enough confidence in formal verification. Relying purely on human judgment vs a metric driven approach is prone to error and use of automation and thorough process is an important part of performing quality formal verification. After the verification was deemed complete on this design it was passed over to us where we applied our defined verification methodology to analyze the quality of the work.

Methodology: The originally completed work contained a total of 141 assertions, 19 cover properties and 40 constraints. More than 80% of properties were bounded. Our signoff methodology first begins with thorough analysis and experimentation with the bounded properties that were left at the end of the verification run. There are several advanced techniques which help to either improve the bounds of assertions or convert them into full proofs. This paper does not explore these in detail however and instead focuses on some of the simpler techniques that can be deployed for bounded proof signoff. For our case study we first performed a bounded analysis on the assertions that had been written. This technique involves generating a set of cover properties on the design and performing a reachability analysis using shortest path formal engines. By doing this, it is possible to get an idea of the sequential depth of the design and a better understanding of how many cycles need to be reached to achieve some level of confidence in a formal signoff. When we performed this analysis on our case study we found that the required bounds were in fact above those that had been deemed sufficient by the human analysis carried out previously. The previous analysis was done simply by looking at the design and working with the design team to understand the likely sequential depth of the design. While this is a useful activity, using formal techniques to calculate the true depth of cover points in the design can provide a more accurate insight. As the cover points told us that the previously reached depth was not sufficient we experimented with different formal engines and some abstraction techniques which allowed us to increase the explored bound. Doing this enabled us to find RTL bugs that existed beyond the previously signed off bound.

In addition to thorough analysis of the bounded proofs, it is important to signoff the quality of the overall formal testbench and completeness. For verification completeness we adopted a state-of-the-art flow to quantify the areas in the design which had truly been tested. This encompassed a flow as follows:

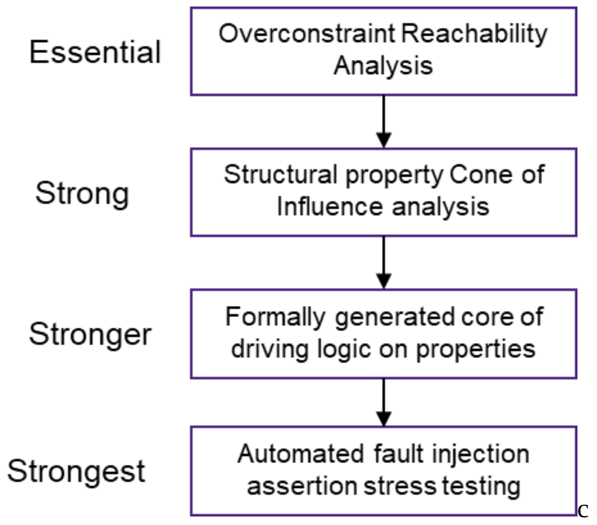

Essential
Overconstraint Reachability Analysis
Strong
Structural property Cone of Influence analysis
Stronger
Formally generated core of driving logic on properties
Strongest
Automated fault injection assertion stress testing
c

Figure 2: Formal Signoff Flow

Each step of this flow is important.

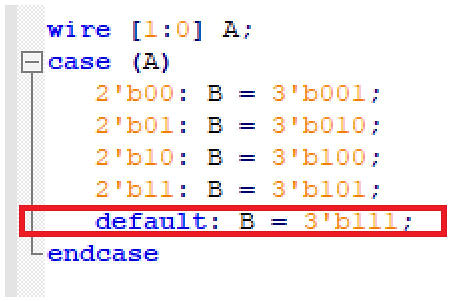
The first stage of the signoff flow incorporates important overconstraint reachability analysis. In contrast to dynamic simulation where the inputs of the design are driven, formal property verification, by its nature, will automatically check all possible interleavings of input combinations. This is part of the exhaustive power of formal verification, however not all inputs to the design are legal. As a result, it is essential to write a set of constraints to prevent illegal behavior from driving the design. There are two possible problems that can arise when incorporating constraints:

Underconstrained Design – The set of constraints provided do not prevent all illegal behavior and problematic inputs may drive the design

Overconstrained Design – The set of constraints provided is preventing some legal behavior and certain input combinations may not be tested.

An underconstrained design is an easier problem to catch. Underconstraints will allow illegal input behavior and as such cause false failures on the set of assertions being tested. These false failures are naturally debugged and uncover the underconstraint as the root cause. By contrast, an overconstrained design is a much harder problem to catch. Overconstraints prevent certain legal behavior and therefore mean that assertions may prove, or remain inconclusive, under a restricted set of behaviors. When a property is proven, it is assumed that the behavior holds true, but an overconstraint could be masking a real design bug.

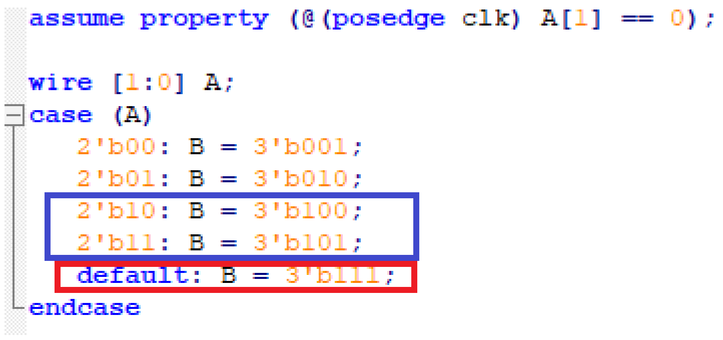
Overconstraint reachability analysis is the process of checking whether all possible cover points in the design are reachable under a given set of constraints. This is performed by doing a formal reachability analysis on a set of generated cover properties. If some of these properties are shown to be unreachable it means, there are areas in the design which cannot be reached and may be prevented from doing so due to constraints. It is of course true that simply because an area of the design is unreachable, it does not necessarily confirm the presence of an overconstraint. There may also be areas in the design which are structurally unreachable due to design construction. Classic examples of this would be default assignments in case statements. In Figure 3 below, the default statement is unreachable by the formal tool, even without constraints in place.



```
wire [1:0] A;
case (A)
    2'b00: B = 3'b001;
    2'b01: B = 3'b010;
    2'b10: B = 3'b100;
    2'b11: B = 3'b101;
    default: B = 3'b111;
endcase
```

Figure 3: Structurally Unreachable Line Target in Design

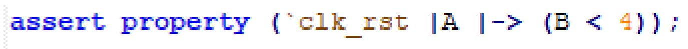
It is therefore important to allow the formal model checking tool to perform an analysis to understand whether targets are unreachable due to constraints or would be unreachable without constraints. Some tools have the ability to perform this analysis automatically. An alternative approach would be to run the analysis with and without constraints and identify the difference. Figure 4 below shows the same simple case statement with an added constraint. In this case there are areas of the design unreachable both with and without constraints.



```
assume property (@(posedge clk) A[1] == 0);

wire [1:0] A;
case (A)
    2'b00: B = 3'b001;
    2'b01: B = 3'b010;
    2'b10: B = 3'b100;
    2'b11: B = 3'b101;
    default: B = 3'b111;
endcase
```

Figure 4: Unreachable Targets with and without Constraints

In Figure 3 we see an overconstraint has been added which ties the MSB of A to 0. This means that we now have areas of the design unreachable with constraints, in addition to the structurally unreachable target. These overconstraints could easily mask a bug in the design. For example, the assertion in Figure 5 below will prove in the presence of this constraint, but it actually masks a bug in the design.



```
assert property (`clk_rst |A |-> (B < 4));
```

Figure 5: Assertion Giving False Positive in the Presence of Overconstraint

As such running overconstraint analysis allows us to identify and remove overconstraints in the design. In the IFU case study we worked on, performing this analysis identified several overconstraints in the design. Once these overconstraints were removed, we were able to identify RTL bugs in the design that would have otherwise been missed. More details in the Results section.

While overconstraint analysis is an important step of checking that the design setup is correct, it does not check whether there are enough assertions in the design, or the quality of the assertions. There are multiple steps in our formal signoff flow to check the quality and quantity of assertions. Each step provides a stronger level of confidence but is slightly more intensive in terms of the runtime cost. By following the flow in order, we can identify low hanging fruit very quickly, before progressing to more intensive analysis.

The first step in reviewing the assertions is to perform a Cone of Influence (COI) analysis. Every property defined in the formal environment has a structural fan-in of logic that traces back to the primary inputs of the design. If we overlay all the COI's of all the assertions in the design, it is possible to see if there are any areas of logic that do not fan-in to any assertion. The presence of logic within the COI of an assertion does not necessarily mean that this logic has been tested thoroughly, however if there is logic outside of the COI of all assertions it means that it has most certainly not been tested.

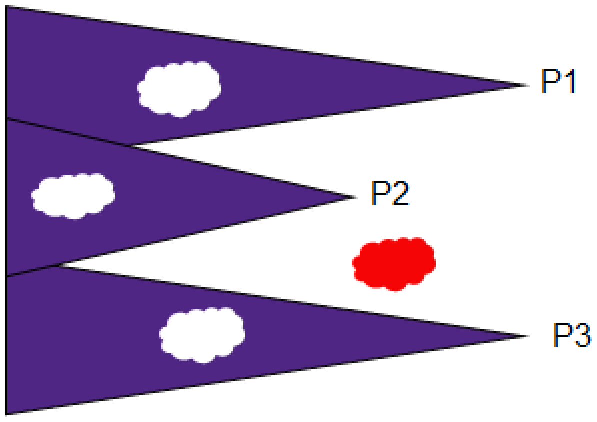
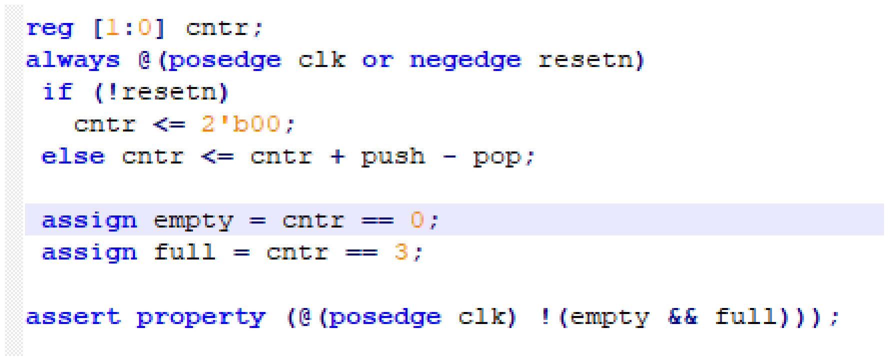


Figure 6: Visual Representation of Logic Outside of COI of 3 Properties

Figure 6 above shows a visual representation of the analysis where there are 3 assertions, whose COIs overlap one another, but there is a small red cloud of logic sitting outside of the COI of all these properties. Due to the fact COI analysis is a structural check, we run this as the first part of our signoff flow. The structural nature of the analysis means that it will always have a quick turnaround time, even on large designs, and the verification holes it finds are just as valid as holes found by more intensive analysis. Due to the coarse nature of the check however it means that there could be many more verification holes that have not been found by this analysis. In our case study of the IFU our COI analysis provided a report that approximately 87% of the design space was within the COI of the assertions, meaning that 13% of the design remained untested. This was untested in the previous revision as no formal signoff techniques had been applied and only human review had determined if there were enough properties.

Although COI is a powerful tool for identifying holes in the verification, the structural nature of it means that it is not as accurate an analysis as other methods. After running COI analysis and fixing the holes, it is then important to perform an analysis of the formal core of the assertions. The formal core can be defined as the subset of the COI that was used by the formal engines to prove the properties.



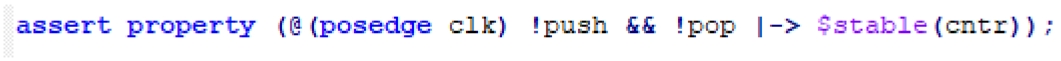
```
reg [1:0] cntr;
always @(posedge clk or negedge resetn)
 if (!resetn)
   cntr <= 2'b00;
 else cntr <= cntr + push - pop;

 assign empty = cntr == 0;
 assign full = cntr == 3;

assert property (@(posedge clk) !(empty && full));
```

Figure 7: Inside COI Outside Formal Core

The example RTL in Figure 7 shows the assignment of the register cntr and an assertion relating to empty and full flags. In a COI based analysis, the cntr would be shown to be inside the COI of the property as it is structurally connected. There is however, no relation between the actual assignment of this register and the proof of the assertion. If we were to replace the cntr with a floating input the property would still prove. As such the cntr is shown as outside of the formal core, although inside of the COI. Formal core based analysis can be performed by commercial formal model checking tools to inform the user of what was involved in the proof (or bounded proof) of an assertion. Formal core analysis is a stronger, more accurate metric than COI, but because the analysis is formal it usually has a longer runtime than COI. Because the COI holes are a subset of formal core, it is still recommended to run COI first.

The final step in the signoff flow is to run fault injection/mutation analysis. This is an important step in determining the final signoff confidence on a formal environment. This fault injection technology used was Synopsys VC Formal FTA. While formal core analysis can identify that *something* has been tested about logic in a design, it cannot tell you whether multiple features of that logic have been tested. Fault injection analysis is the step of artificially inserting bugs into the design and checking whether the assertions falsify in the presence of these faults. If all assertions prove in the presence of a fault, this indicates a verification hole as we were unable to catch the issues. If, however at least a single assertion fails in the presence of a fault then this is a sign that we can detect the fault, and an indication of robustness.

Referring to the small design in Figure 7, we could add the assertion below to ensure that cntr appears in the formal core.



```
assert property (@(posedge clk) !push && !pop |-> $stable(cntr));
```

Figure 8: Additional Assertion to Improve Formal Core

The assertion in Figure 8 checks something about the cntr register directly and ensures that if push and pop are both low, the cntr remains stable. This means that cntr would be in the formal core of this assertion as it is involved in the proof – without cntr, the property would fail. The assertion however only checks one thing about the cntr. It does not check the increment and decrement conditions happen correctly, only that there are no changes when push and pop are low. By running a fault injection analysis, we could inject faults to remove the else condition or flip the +/- signs and none of these faults would be caught – the assertion would still prove. By running fault injection analysis, we can validate the quality of assertions. Fault injection analysis however can again be more runtime intensive. In general, it is quicker to falsify a property in formal than it is to prove a property (though there are exceptions). This means that the stronger your formal environment is when you enter the fault injection phase, the better the performance that will be seen. Equally, if there are areas of the design that are not supposed to be tested, then they can be found using other techniques and excluded from the fault injection. As a result, it is much better to step through each stage of the formal signoff process, increasing the confidence as the steps are performed. In our case study, an additional 17% of the design was found to be untested on top of the 13% found using COI analysis.

Results: By applying our methodology we were able to identify initially that the bounded depth of 15 which was considered sufficient was in fact not large enough to be used for signoff. By running a bounded analysis and experimenting with cutpoint based abstraction and bug hunting techniques, we identified 33 assertions, out of the original 141 previously signed off at a bounded depth of 15 which subsequently failed between 16 – 20 cycles. This highlighted the importance of looking futher into bounded proofs.

In addition to the falsifications that were found at these greater bounded depths we were able to use overconstraint analysis to ensure that there were no overconstraints in the setup. Using overconstraint analysis has allowed us to prove that of the 1110 extracted cover points, none of them are overconstraining the design.

In total we were able to identify 4 RTL bugs that were missed by the initial formal verification process. Of these bugs, all of them were found as a result of discovering the bounds explored were not sufficient and using techniques to improve the bound. All of the bugs are related to incorrect branch prediction which would have caused serious problems if they had made their way into Silicon.

Finally, on top of the new RTL bugs we discovered using the already existing assertions, our thorough sign-off flow identified that approximately 30% of the design in total was as yet untested. This was identified through the use of mutation analysis which showed that 30% of 2150 injected faults in the design could not be caught with the current assertions. After identifying the verification holes, we were able to add 8 new assertions, requiring 3 constraints in order to cover the verification holes.

	Original Signed Off Design	**With Our Signoff Methodology**
Total #Properties	141	149
Signoff Methodology	Bounded at 15 Designer review	Full fault detection with FTA
Additional found bugs	N/A	4

This paper highlights that it is not enough to rely on a technology alone. While formal verification is a very powerful tool, it is important for management and engineering teams to understand that correct methodology is equally important. Without the application of a properly defined signoff flow it is easy to under-verify a design and miss bugs. While this is true of any verification methodology, it is possible to gain a false sense of security with formal verification due to its exhaustivity and the belief that it will catch all issues.