Design Guidelines for Formal Verification

Anamaya Sullerey
Juniper Networks
2530 Meridian Parkway
Durham, NC 27713

Abstract-**Improvement in capacity and usability of EDA tools has helped in pushing the formal verification envelope. Capacity still remains the major limiting factor in the scope of formal verification deployment. Formal verification experts employ a variety of techniques to overcome this challenge. Design and implementation choices made by the designers greatly influences the effectiveness of these techniques as well as ease with which they can be applied. Most designers are not exposed to the formal verification process. This paper proposes design guidelines that facilitate application of formal verification on large blocks.**

I. INTRODUCTION

Formal verification adoption has accelerated in the past few years, becoming quite widespread. One common use case of formal verification is design bring-up, a process that guarantees basic design sanity. The designer is often responsible for this activity. This is then followed up by a thorough formal verification exercise. A formal verification engineer is typically responsible for this subsequent activity. Formal verification engineers have a good understanding of design complexity and tool capacity. They are also aware of the abstractions and optimizations that can be applied to make formal verification properties tractable. This knowledge is obtained through the hands-on experience of working on a variety of formal verification problems as well as their educational background. On the other hand, most designers are not sufficiently exposed to the challenges of formal verification and the methods that are employed to solve these challenges. This technology is new to them and quite different from the more commonly used simulation based verification techniques.

The effort required for formally verifying a design depends on design complexity and design coding style. It is frequently noticed that designs with similar complexity can require very different formal verification efforts. Some of the attributes of a formal verification friendly design are similar to the attributes of a good design, but not all fall under this category. Two clean implementations of the same design can have different formal verification effort requirements. Design structure and quality of code is an important issue for simulation, but it is critical for formal verification since formal verification has tool capacity limitations. While it is very important to prove a property, the time that the tool takes for the proof is also important. Shorter proof time allows for quick iterations for bug fixes or for constraint fixes.

There exists a large quantity of good literature on handling complex design problems that is targeted towards formal verification engineers. This paper, however, is targeted towards the designers. It elaborates key principles that allow designs to be more formal verification friendly. Such designs allow formal verification abstractions and optimizations to be applied with ease. The examples presented along with the design guidelines in this paper are kept simplistic so that the design guidelines can be easily explained. Most seasoned designers would follow the correct approach in these cases. These examples, however, are only used to illustrate the effect of design decisions on formal verification. Most cases in real-world designs will be more complex and not as black and white.

The proposed guidelines in this paper are distilled from formal verification work done over multiple projects, as both design and formal verification engineer. They are also derived from literature covering the techniques applied by formal verification engineers to address complexity. One project in particular, an Ethernet switch function [1], provided the knowledge that became the key source for these guidelines. In this case study the designer and formal verification engineer engaged early in the project and enabling formal verification was a key design criteria. This approach resulted in a very successful formal verification exercise.

II. DESIGN GUIDELINES

A. *Functional design paradigm*

Functional design is a term that has been commonly used in the context of software programs. Wikipedia describes a functional design as where “each modular part of a device has only one responsibility and performs that responsibility with the minimum of side effects on other parts" [2]. Another design approaches is event driven design, which is centered on performing certain actions in response to observed events. A similar approach is data driven design that centers on performing certain actions based on observed data. These terms have their origins in software engineering but the concepts behind them can be extended to logic design.

Designers use a mix of these three approaches to construct designs. A subset of the functionality of any block maps clearly to a set of well-defined functions. For this subset, a designer typically uses the functional design approach. Other parts of the functionality may be better implemented using the event driven approach. In these cases the blocks are not typically constructed as a set of functions, but rather as a set of actions in response to the specific input events and input data. While these two approaches may yield very similar designs, minor differences between the two may be critical from a formal verification point of view.

The most useful technique to overcome capacity issues in formal verification is "*divide and conquer"*. In this approach the functionality of any big or complex design block is partitioned into a hierarchy of smaller parts and these parts are verified independently. A functional design approach naturally results in a hierarchy of sub-blocks. Since the leaf blocks are smaller and less complex, it is easier for these to be tested individually. The "*Assume guarantee propagation"* technique can then be utilized to prove overall design correctness. This is a commonly used technique in which assumptions made at the input of a sub-block are verified as assertions at the output of the sub-block that drives them. While the various guidelines being proposed in this paper are distinct, in most design scenarios there is a correlation between them. Following one of them helps the cause of another. In this respect, having functionality divided into smaller sub-blocks with clearly defined functions for each sub-block is the most crucial guideline.

Consider ingress path functionality for Ethernet traffic in a typical networking chip. A MAC (Media Access Controller) receives ingress data and offloads most of the Ethernet protocol. The MAC sends these Ethernet packets to a packet parser. The packet parser then inspects the headers of the various networking layers in order to classify the type of processing required. The format of theses headers and layering of protocols is flexible thereby requiring support for a multitude of variations in parsing. A minimum packet length is also expected for all legal packets. A packet smaller than the minimum length is called a runt packet. All runt packets are discarded.

A typical simplified version of the data transfer protocol at the input and output of a MAC has an SOP (start of packet indicator), an EOP (end of packet indicator), an error indicator that is valid at EOP, a data bus, and a valid signal that qualifies all other signals. A packet starts with an SOP indication and ends with an EOP indication. Valid data transfers from different packets are not interleaved. This is referred to as the packet framing rule. Often the MAC is a design block bought from an external vendor. It is assumed that there can be scenarios where the MAC does not follow the framing rule. A MAC can potentially send multiple SOPs without any EOP or vice versa. The ingress data path functionality is required to discard data in such cases. The design terminates the current packet by setting EOP along with the error indicator. Subsequent data is discarded until a new legal packet boundary is detected.

An event driven design approach looks at the input combinations as a cross product of all legal packets, runt packets, and packets with framing errors. The resulting state machine discards runt packets, discards illegal packets, and parses packets in parallel. Fig. 1 shows the state diagram of an implementation following an event driven approach. In this example the data width of the bus is set to 16 bytes and minimum packet size is defined as 40 bytes. Packet data is parsed up to a certain length based on the information in the packet header and the rest of the packet is considered pass-through. States and arcs in blue are related to runt packet filtering. States and arcs in red are related to handling packets with framing error. States and arcs in the black are related to parsing network headers.

In this implementation, a property covering a framing rule at the output of the parser is a function of all packet formats due to an implementation choice. A property that checks the correctness of the parser is also a function of illegal packet and framing rule error scenarios. There are cases where a packet is a runt packet and also has framing error. In such cases the packet needs to be discarded and a new packet boundary needs to be determined. This makes properties related to framing checks dependent on the length of the frames and vice versa. Actions related to each state, which are not covered in this state diagram, have similar dependencies.

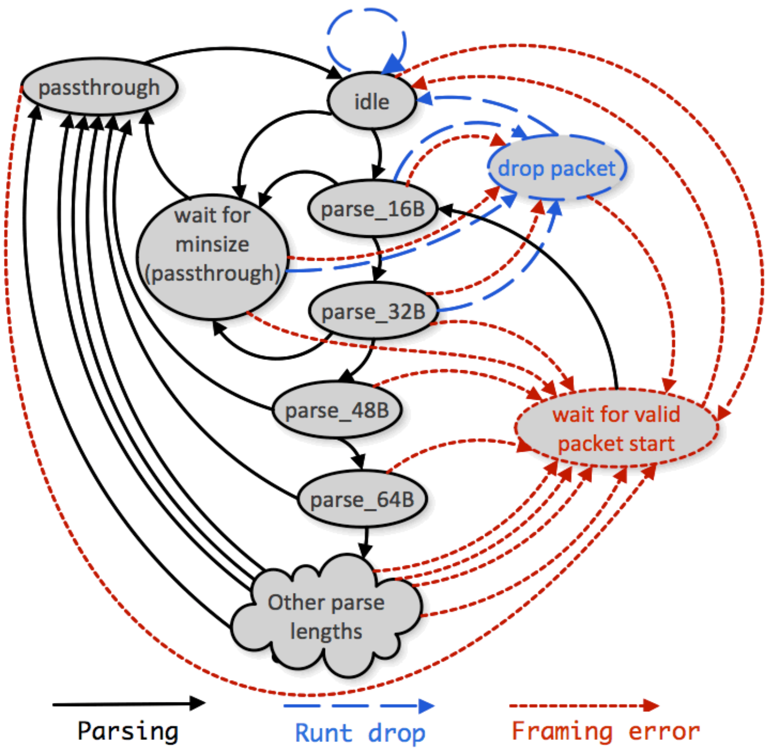


Figure 1: State space diagram for event driven un-partitioned logic

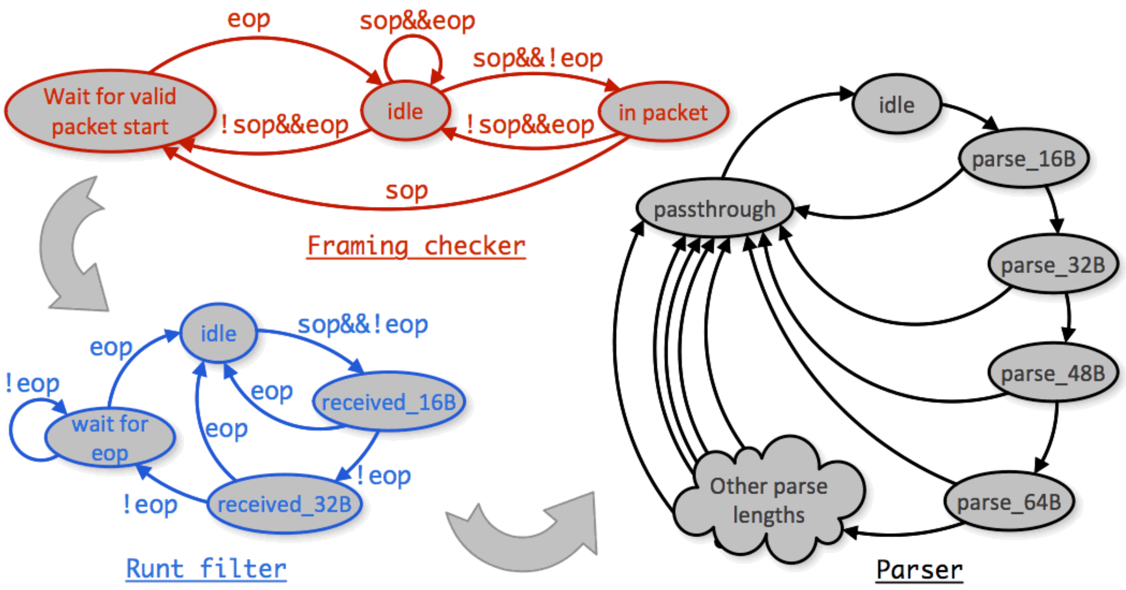


Figure 2: Functional driven partitioned logic

In a functional design approach the functionality for this logic is divided into three stages working sequentially. Fig. 2 shows the state machines of the block stages of the logic partitioned functionally. The first stage fixes any SOP-EOP framing rule issues with received packets. The second stage drops all runt packets. With this design architecture, the input to the third stage is then clear of any protocol errors. The third stage finally does the packet parsing. SOP-EOP framing rule can be easily formally verified at the output of the first stage. Runt packet drop checks can be easily verified at the output of the second stage and are further simplified based on the assumption that framing rules are followed at the input of second stage. Packet parsing rules are also much easier to verify at the third stage assuming clean packets at the input of the third stage.

B. *Clear and succinct interface definitions*

For formal verification, input constraints of a block are captured in assumptions and output constraints are captured in assertions. These constraints are sometimes fully expressed as SVAs (System Verilog Assertions) and are sometimes expressed with the help of test-bench functionality expressed as synthesizable RTL. The state space for a formal verification property proof includes the state space of the DUT and the state space of these constraints. Clean interface definitions enable the ability to express constraints in a simple and succinct manner. Besides reducing state space, clean interface definitions also makes model and constraint generation simpler.

Design documentation is targeted for simulation and describes the design at the block level. Since formal verification is often done at sub-block levels there is usually no documentation available that describes sub-block interfaces. Messy interfaces require a lot of interaction between the designer and formal verification engineer in order to create correct constraints.

Clean interface definitions avoid transfer of information between blocks through implicit means. Often this issue is introduced during implementation phase due to new requirements or bug fixes, but can also be introduced due to poorly defined interfaces during the architecture phase of design. Here is an example that demonstrates implicit information transfer: Consider a memory subsystem block that is connected to a chip interconnect bus that accepts 5 different command types. Table 1 shows the allowed data lengths and read (R) and write (W) attributes with each command type along with the clients (C#) that use the command. Eight different clients use this memory subsystem. Furthermore, some clients share memory space and some have a dedicated region. During performance evaluation it is determined that one particular client (C5) that only uses the cmd2 with 1KB access, needs higher priority processing. No other clients use this command.

TABLE 1: MEMORY SUBSYSTEM COMMAND DESCRIPTION

Command type	**1B-32B**	**64B**	**128B**	**128B-1KB (64B increments)**
cmd0	RW (C1,C2)	RW (C1,C2)	-	-
cmd1	RW (C3)	RW (C3)	RW (C3)	-
cmd2	R (C4)	R (C4)	R (C4)	**R (C5)**
cmd4	-	-	RW (C6)	-
cmd5	-	-	-	R (C7,C8)

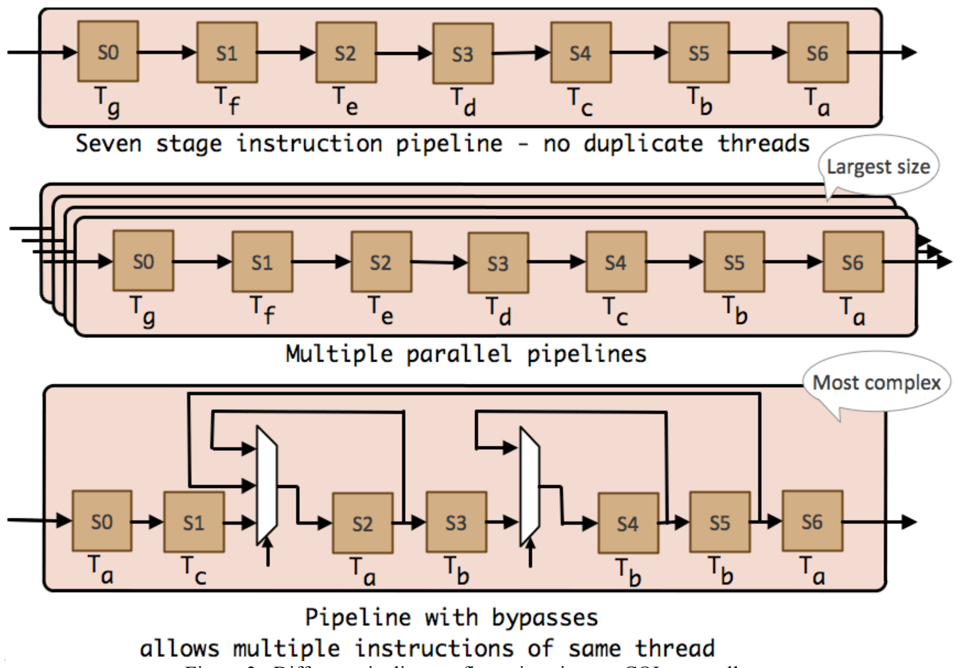
One way to address this requirement is to interpret priority from command type and data length. This approach does satisfy the immediate needs of the system, however, it adds implicit information dependent on command type, data length, R/W access, and client. This approach is not future proof since going down this path can lead to very confusing dependencies after only a few sets of changes. It also requires more constraints and more interactions between the designer and the formal verification engineer. The correct approach is to add a priority signal in the interface. This signal reflects the actual design requirement change and is also self-explanatory. Adding explicit priority signal future proofs the design from other clients having similar priority access requirements.

C. *Keeping state space as a design consideration*

The state space of a block is one of the key design considerations for formal verification. If the state space is too large, the block should be partitioned into smaller blocks. However, some problems are intrinsically complex and cannot be broken down beyond a point. In such cases, close coordination between the formal verification engineer and designer during the design phase is highly desirable.

Perception of the size of the state space of a particular design and the formal verification tools capacity is an acquired skill. For this reason, first hand use of a formal verification tool by designers is a very productive exercise. Formal verification can help significantly during the bring-up phase of the design, i.e.

initial cleanup after the design is coded. Some tools provide design exploration features that help kick start formal verification with minimal setup. This is a good way to be introduced to the technology.


S0
S1
S2
S3
S4
S5
S6
T_g
T_f
T_e
T_d
T_c
T_b
T_a
Seven stage instruction pipeline - no duplicate threads
Largest size
Multiple parallel pipelines
Most complex
Pipeline with bypasses
allows multiple instructions of same thread
Figure 3: Different pipeline configurations impact COI unequally

The total number of gates, flops and memory elements in the design do not translate directly to the state space of a property. Rather, the size of the COI (Cone of Influence) more directly influences the state space of a property. COI includes all primary inputs, all combinatorial and sequential logic, and memory elements that influence that property. Even the size of the COI does not always provide a complete picture. It is the size of the COI and the connectivity within COI that determines complexity. Consider three variations of a seven-stage multithreaded instruction pipeline that is shared by large number of threads as shown in Figure 3. In the first scenario, the high-level architecture assures that a particular thread cannot have instructions in two stages of the pipeline simultaneously. In the second scenario, the implementation has multiple copies of the previously described pipeline in parallel. In the third scenario, there is a single pipeline that allows multiple instructions from the same thread. The first and third cases have a similar number of gates, however, the complexity in the third case is much higher than the first. The second case has a much higher number of gates than the third case, but it is still less complex than the third case. The bypasses introduced in the third case may not have a high impact on COI size, but the overall complexity of the pipeline has multiplied.

D. Symmetry

Formal verification tools exploit symmetry in designs to reduce the state space. Many designs have replication of functionality. These designs can benefit from symmetry-based optimizations [3]. If designs are almost symmetric with some exceptions, the exception cases can sometimes be isolated into a separate logic partition leaving the bulk of the design symmetric. Symmetry also helps reduce overall effort required to code constraints and prove the property as it reduces the amount of unique logic that needs to be verified [4].

Consider a block that receives packets from three sources with bus widths 512, 256, and 128 as shown in Fig. 4. The design adds a 512 bit header to each packet and then passes the packets to one of the two possible processing engines. In the first implementation the header is inserted prior to the spray FSM. Then the spray FSM selects the source, does the data width conversion and passes it to one of the outbound interfaces. In this design, all of the header insert blocks and code within spray engine with respect to ports is unique. An alternate implementation first converts 512 bit and 128 bit inputs to a 256 bit bus. This makes the internals of the spray FSM very symmetric with respect to the ports. The header insert blocks, which are now placed after the spray FSM, are also identical. It is easier to write properties for the second implementation. The spray FSM in the second case is data-independent, i.e. data passing through is not modified, which makes data bits in the 256 bit bus symmetric. This allows reduction of the 256 bit bus to a single bit signal for the purpose of formal verification.

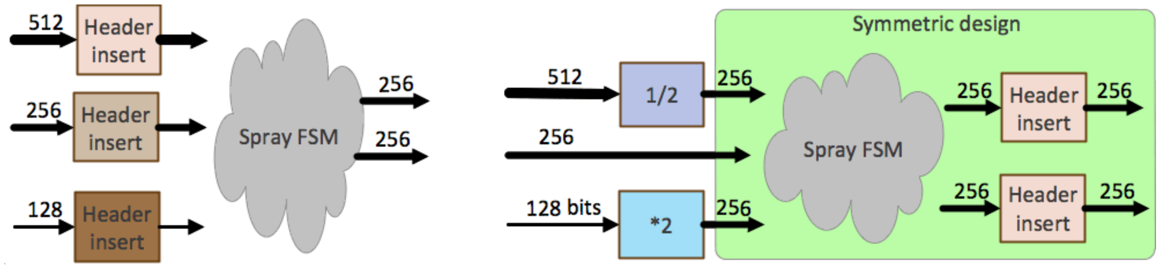


Figure 4: Asymmetric versus symmetric implementation

E. *Parameterization*

Designs can often be scaled down without affecting key aspects of the design that need to be verified. This can be done through parameters or Verilog pre-processors common in many design environments. Proving correctness of a scaled down design often provides significant value and is very close to proving the correctness of the original design. There are cases where the properties can be proven in an unmodified design, but it takes hours to prove them. In such scenarios, scaled down versions can be used to formally verify the design and constraints much more efficiently.

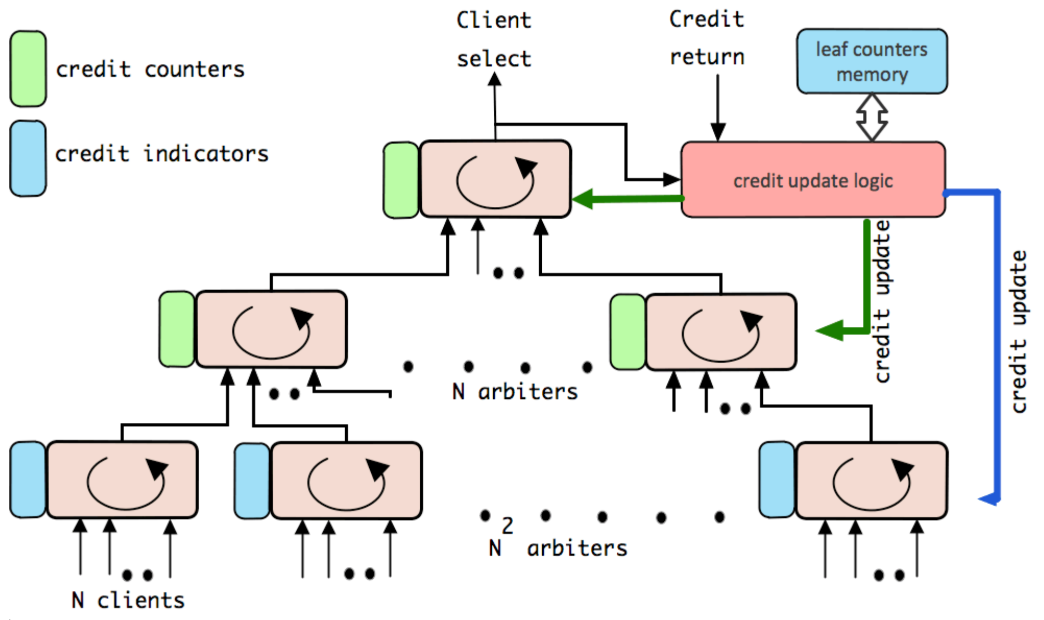


Figure 5: Scheduler block diagram

Consider a scheduler with 4K clients as shown in Fig. 5. The basic building block of this scheduler is a 16 client round robin arbiter. This scheduler is implemented in a tree structure with three stages. First stage has 256 of these arbiters, the second stage 16, and the last stage has 1. Each client has an 8 bit credit count associated with it. A request from a client is entertained only if that client has non-zero credit. A single credit is consumed for a grant and credit is returned back based on external events through a shared credit return bus. The credit counts for individual clients are stored in memory to save area. Arbiters in the first stage of this scheduler store a bit per client that indicates non-zero credit. Arbiters in the second and third stage store total credits for all clients under each requester port. Since this implementation operates over multiple cycles, there are mechanisms to block requests from various nodes while credit decrement operations after any grant for those nodes are being performed. Properties for this arbiter have a very large COI.

This arbiter has a very regular structure. The number of clients at each stage can be parameterized. The size of the credit count for the clients can also be parameterized. A significantly scaled down version of the design can be generated with the number of clients of the basic building block arbiter set to 4 and a credit count width of 3. This smaller scale version preserves the overall structure of the design and all interactions between the stages. There is a greater chance to prove properties for this implementation than the full-scale design and the proofs will take much less time. Once the scaled down design is clean, parameter values can be dialed up until the point where the formal tool can just handle the complexity.

Designers need to watch for logic that has large COI but is regular in its structure or function. Such logic has the potential to be tested using its scaled down version. Both large FIFOs and wide counters [4] are also good candidates for these optimizations. Besides a few boundary conditions, they are very regular in their function. The boundary conditions are full and empty for a FIFO and rollover for a counter. When reducing the design sizes, these boundary conditions are preserved but the state space is reduced.

F. Capturing design invariants

Designs blocks have invariants around which the code is structured. Many of these invariants are internal to the design blocks and have little or no direct correlation with high-level specification. The following are some examples of these design invariants:

- A particular bit vector is one hot
- A request to an internal arbiter will get a grant within N cycles
- An operation in some command FIFO implies availability of operands in a particular data FIFO
- A certain FIFO can never be full
- Some internal events can not be generated in a particular power state

It is a good practice to add assertions for these design invariants. These assertions are often simpler to prove than end-to-end properties. Tools automatically use the proof results of one property for the proof of other properties. These assertions are very good intermediate points for the proof. They help guide the tool during proof of more complex properties.

G. Code structure

Formal verification of large designs often requires replacing parts with models. This may be in form of replacing modules or introducing a cut point with assumptions. Ease of these modifications depends on code structure. While any particular code structure does not prevent one from making these changes, it does have the potential for making it a very laborious and error prone process.

Consider a pipeline that is implementing cryptography protocol. The design has a pipeline control aspect and a cryptography aspect. These are quite orthogonal functions. The cryptography algorithm is implemented over multiple stages. The cryptography function in this case is verified through a software model in simulation. The pipeline is responsible for fetching keys, providing inputs to the cryptography function at the correct time, and storing cryptography state. For formal verification of the pipeline, the cryptography functions can be replaced with other much simpler functions. If all of the design code is located together, then the formal verification engineer needs to modify this code and continuously keep it in sync with the production code. This approach is inefficient and error prone. The correct approach is to isolate the cryptography portion of the design in a separate location. The formal verification engineer can then easily replace this module with a simplified model.

The following are some other guidelines for coding that help formal verification:

- Instantiating memories outside of logic. Allows easy identification and replacement of memories.
- Have a process in which properties related to a set of signals traversing through the design hierarchy can be available at any level of the hierarchy. This allows for elaborating the design at any desired level.
- Create expressions composed of meaningful intermediate terms. They make the code more readable and allow quick setup for partial proofs (proofs done for a subset of input space) by adding cut points at these intermediate terms.

H. Error isolation

Many designs handle very large sets of independent contexts. The state for these contexts is stored in memory or flip-flops. A few contexts are processed at a time in a pipelined fashion. An example of a context can be a cache line in a memory subsystem or a queue-id or packet-id in a queuing system. There are legal defined states for each context. For each of these states there is a set of legal defined operations. The design operates on a context in response to external stimuli or an internally generated event. Many such contexts are handled in parallel in the design. The processing of contexts is completely uncorrelated with each other.

For example, a multiprocessor cache using a MESI protocol will have cache line in either the modified, shared, exclusive or invalid state. Read operations are allowed in all but the invalid state. There are similar constraints are on other operations such as write. As such, the input constraints require knowledge of the state of the context and any model for formal verification also needs to keep track of the state of the context. Since the total number of contexts can be very high, the total amount of internal state and formal model state required results in tool capacity issues. Formal verification of this class of designs concentrates on proving correctness of one context [1]. The tool is allowed to pick any context during reset and the constraints are targeted to this chosen context. Behavior of other context is left unconstrained. Internal design memories storing context state are replaced with models that track only a single memory location. This technique cuts down the size of the design drastically by eliminating memories and a large number of constraints.

In this approach the behavior of the other contexts is completely unconstrained. The formal verification tool is allowed to create erroneous cases for these contexts. This approach can only work if the design insures that the error conditions in one context do not affect the state of any other context.

III. Adherence to Design Guidelines

These guidelines are essentially good design and coding practices in the context of formal verification. While examples of good implementation can be provided, it is not possible to have strict rules that define a good implementation.

Designers should be encouraged to follow the guidelines through the following approach:

1. Providing and maintaining good literature on the design guidelines.
2. Providing examples of formal verification friendly designs that were successfully verified.
3. Encouraging designers to do formal verification, especially for initial design bring-up.
4. Having a formal test plan [5]. Designers should be involved in the formal test plan review.
5. Involving formal verification engineers early in the micro-architecture process. They can provide early feedback on possible formal verification challenges.
6. Checking adherence to the design guidelines for formal verification in various reviews. Table 2 shows the reviews that are helpful in covering each of the recommended guideline.

Table 2: Reviews for the design guidelines for formal verification

Guideline	Review
Functional design paradigm	Micro-architecture
Clear and succinct interface definitions	Interface
Keeping state space as design consideration	Micro-architecture, Formal test plan
Symmetry	Micro-architecture, Formal test plan
Parameterization	Formal test plan
Capturing design invariants	Assertion and Coverage
Code structure	Code
Error isolation	Functional specification, Micro-architecture

Acknowledgments

I am grateful to Sanjeev Singh, Jonathan Sadowsky, David Talaski, and Erik Seligman for providing valuable feedback regarding various drafts of this paper.

References

[1] B. A. Krishna, A. Sullerey, A. Jain, “Formal Verification of an ASIC Ethernet Switch Block”, FMCAD, Oct 2010.

[2] Wikipedia contributors. "Functional design." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 26 Jun. 2014. Web. 22 Jan. 2015.

[3] C. N. Ip, D. Dill, “Formal Methods in System Design”, Vol 9, 1996.

[4] A. Datta, V Singhal, “Formal Verification of a Public-Domain DDR2 Controller Design”, VLSI Design, 2008.

[5] H. Foster, L. Loh, B. Rabii, V. Singhal, “Guidelines for creating formal verification testplan”, DVCON, 2006.

[6] E. Seligman, R. Koganti, K. Maheswaran, R. Naqib, “Bringing Formal Property Verification Methodology to an ASIC Design”, DVCON, 2006

[7] L. Benning and H. Foster, “Principles of Verifiable RTL Design”, 2nd Edition, ISBN 0-7923-7368-5, Kluwer Academic Publishers, 2001