FORMAL TECHNIQUES TO VERIFY FUNCTIONALITY OF DIGITAL MEMORY DECODER

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE

MASTER OF TECHNOLOGY
IN
VLSI DESIGN AND EMBEDDED SYSTEM

Submitted by:

Gaurav Kasana

(2K21/VLS/07)

Under the Supervision of

Dr. Sonam Rewari (Assistant Professor)

Mr. Lokesh Gautam (Assistant Professor)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY, DELHI

(Formerly DELHI COLLEGE OF ENGINEERING)

Bawana Road, Delhi- 110042

May 2023

ACKNOWLEDGEMENTS

First and foremost, I would like to extend a deep sense of gratitude to my project advisor **Dr. Sonam Rewari** and **Mr. Lokesh Gautam** for their very valuable guidance, monitoring and motivation throughout the course of this project. It was their immense knowledge, stimulating suggestions, patience and encouragement that helped me coordinate this project in a descent manner.

I am grateful to **Dr. O.P. Verma**, Professor, Head of Department, Electronics and Communication Engineering (ECE) and the faculties of Electronics and Communication Department for all their help in completing the project. I also thank my family and friends for their support throughout the work.

I am deeply grateful to the members in Intel India Technology Pvt. Ltd especially **Mr. Suresh Kumar Varanasi, Mr. Manu Mathews,** and **Mr. Anjan Bhowmik**, for their continuous support and helping me understand the methodologies and workings of various tools in Intel.

DECLARATION

I declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this thesis. I also declare that I have adhered to all the principles of academic honesty and integrity and have not misinterpreted or fabricated or falsified any idea/data/fact/source in my submission.

Place: DTU, Delhi
Date: 31/05/2023

Signature:
Name: GAURAV KASANA
Roll No: 2K21/VLS/07

DELHI TECHNOLOGICAL UNIVERSITY

Department of Electronics and Communication Engineering

CERTIFICATE

This is to certify that the Thesis entitled, **"FORMAL TECHNIQUES TO VERIFY FUNCTIONALITY OF DIGITAL MEMORY DECODER"** submitted by **Mr. Gaurav Kasana** to the Delhi Technological University (DTU) towards partial fulfillment of the requirements for the award of the Degree of Master of Technology in **Electronics and Communication Engineering** (**VLSI and Embedded System**) is a bonafide record of the work carried out by him under my supervision and guidance.

Dr. Sonam Rewari
Supervisor
(Assistant Professor**)**

Mr. Lokesh Gautam
Supervisor
(Assistant Professor)

Place: DTU, Delhi

Date: 31/05/2023

TABLE OF CONTENTS

ABSTRACT

This report discusses the formal property verification of physical digital memory decoders, which are a crucial component of most digital integrated circuits. We present a formal verification technique that is suitable for verifying the correctness of physical layer digital memory decoders. The method is based on a formal language for describing the system, which is then verified using Formal Property Verification techniques. To handle the complexity of the model, we also introduce a novel abstraction technique that reduces the number of states in the model while preserving its behavior. This abstraction technique enables more efficient verification of the model, reducing the time and computational resources needed to verify its correctness. We discuss the results of the verification process and compare the results with those obtained using traditional simulation techniques.

LIST OF ABBREVIATIONS

IP	Intellectual Property
FV	Formal Verification
JG	Jasper Gold
FPV	Formal Property Verification
SV	System Verilog
SVA	System Verilog Assertions
SoC	System on Chip
RTL	Register Transfer Level
HDL	Hardware Description Language
GUI	Graphic User Interface
DUT	Design Under Test
PSL	Property Specification Language
SPEC	Specification
IMP	Implementation
CEX	Counter Example
CPU	Central Processing Uni
I/O	Input/Output
FSM	Finite State Machine
Clk	Clock

LIST OF FIGURES

List of Table

CHAPTER 1

INTRODUCTION

The capacity of integrating gates into chips is increasing at very fast rate, and due to this System-on-Chip (SoC) design is getting more and more popular. However, verification has been a serious problem in the SoC design. The traditional verification methods suffer from huge numbers of test vectors and are becoming inefficient. Formal verification mathematically analyses all the possible design behaviors, instead of using specific values to compute the results. It does not actually simulate all the possible values but uses computationally extensive mathematical algorithms to prove the design behavior. Ideally a design can be verified by simulating all possible inputs and check results for each of these combinations. But this is not practical as the computation complexity is huge, and it will take long time to complete. Hence, the current industry verification scenarios are based on targeted simulation where some predefined stimulus is applied to cover most of the interested areas of the design. But there is a high probability that some of the areas can remain uncovered and hence some corner case bugs can escape. Formal Verification is a method which is close to the ideal verification paradigm and thus it ensures almost 100 percent coverage. To handle the computation overhead of such process, different heuristics and algorithms were developed which would handle complex designs efficiently. There were different EDA tools developed and thus formal verification be- came a practice, which is feasible to adopt to current industrial scenarios.

Fig. 1 shown below says that formal analysis is just part of the solution to a verification problem. It does not directly replace any other methodology or tool, but it can make significant savings, for example, use of more formal techniques can drastically reduce the amount of simulation which needs to be done. Formal verification is an umbrella word for a group of methods that use static analysis based on mathematical changes to establish the accuracy of hardware or software behavior. This is in comparison to dynamic proof methods such as simulation.

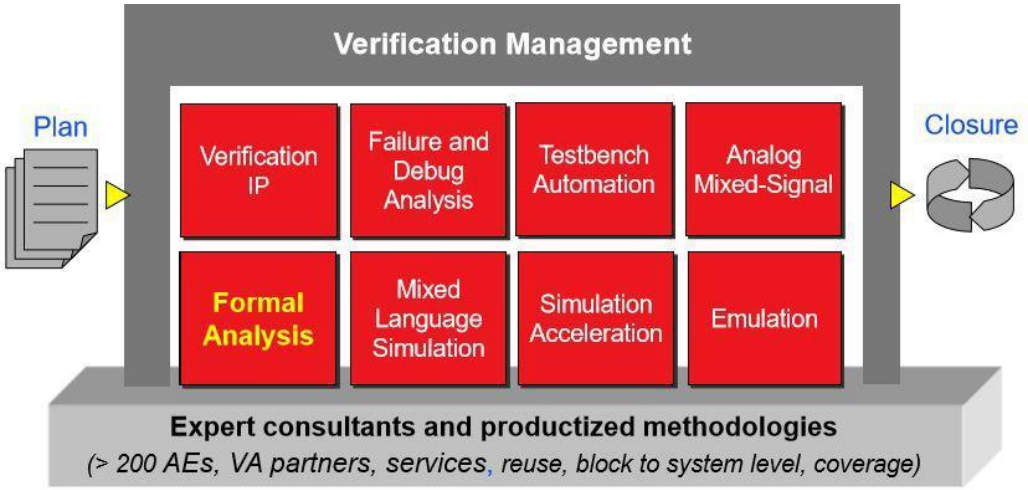


Figure 1.1: Expert consultants and productized methodologies

1.1 Literature Review

The initial development in the formal verification were in the field of computer science where it is used to find the correctness of a software program. There were different algorithms used for this purpose with varying performance and complexity. Then later these techniques advanced into hardware description languages and RTL (Register Transfer Level). Further research in this area led to rapid enhancement of efficiency in running formal algorithms on complex designs such as microprocessor components. One of such algorithms is called as Binary Decision Diagram (BDD). Since the design complexity was increasing over time new challenges were faced and some of the research proposed better and efficient way of employing formal methods in SoC level. The most prominent advantage of formal methods over simulation is that it can used to get early verification closure. As the formal methods are advanced, different languages used to write the properties. As a standard language System Verilog Assertions (SVA) [3] is matured in parallel, which enabled the easy and standard way of writing properties. To test whether the design so made meeting all the specifications as required, in general, two techniques have been followed viz. traditional design flow and formal verification. In traditional design flow only, probabilistic assurance can be ascertained. Whereas Formal verification provides a holistic view pertaining whether design meets all or parts of specifications as it uses complex and rigorous mathematical reasoning.

1.2 Objectives

The objectives of the project are listed below:

- **Getting Familiarized with the formal verification paradigm:** The formal methods are relatively new and very less engineers have explored the possibilities of it. Hence, this work intended to understand the possibilities of formal methods and applicability of it in the current industrial scenario.
- **Getting hands on knowledge on industry standard tool Jasper Gold:** Even though many EDA vendors provide different formal tools, Jasper Gold from cadence is the most adopted one in industry. It has many formal apps built-in which helps in the complete validation of the design unit. Hence, proper understanding of the tool is necessary for the successful and timely completion of formal projects.
- **Exercising an IP and module level formal verification and identifying the challenges:** Most of the work done by engineers previously was in a block level or module level. In this project an IP level formal verification [7] has been tried out with SEC. This is more challenging as the complexity of the design is a critical parameter as far as the formal method is concerned.
- **Identifying the advantages and drawbacks of formal methods:** To verify a design formally, the time and effort required is less compared to simulation. But formal has its own disadvantages especially when the design size is large. This project is aimed at identifying the challenges especially in the context of end-to-end IP verification and improving the methods if possible.

1.3 Motivation

The traditional VLSI design verification methods such as simulation and emulation have disadvantage such as low coverage on the design. But formal methods can provide almost 100 percentage coverage and thus it is the most reliable verification method. This full coverage is ensured by covering all the possible states of a design it can undergo.

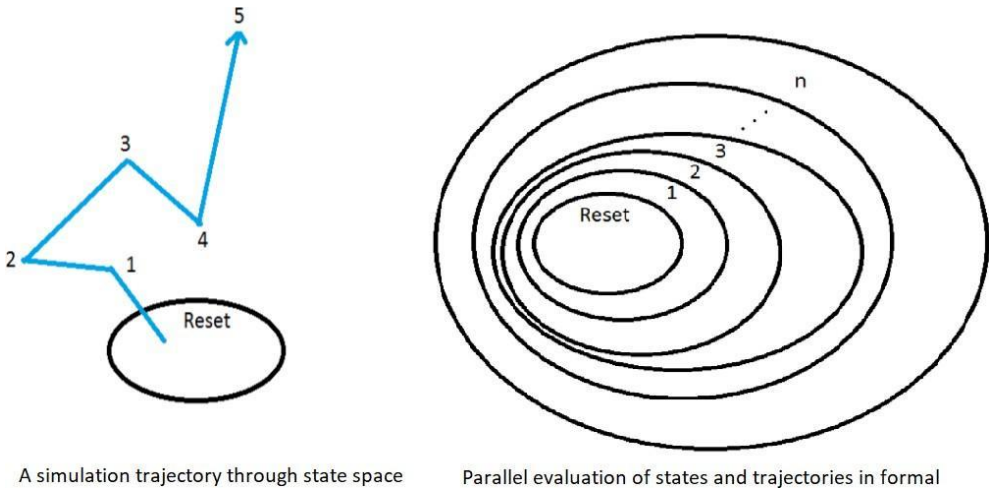


Figure 1.2: State Space Analysis [5]

Fig 1.2 shows how formal and simulation-based verification covering [5] each state space. Formal starts with Reset state and will cover all possible value in each state parallel and there is less chance for missing bugs, whereas in simulation, based on test case it will jump for one point in state space to another point in another state space. In simulation, there is high chance of missing state space, that may lead to masking of bugs.

But the computation required for this will increase exponentially with the design size and this was limiting the adoption of formal techniques in industry. But recently, lot of researches took place in this direction and software tools have been developed which would handle this overhead of computation efficiently [7]. If a VLSI industry has a proper formal verification strategy the quality of the products released will increase. Also, the adoption of formal techniques will considerably reduce the resources required to complete the verification process. Hence the study of formal verification methods is important to have a good future verification strategy.

1.4 Methodology

The project started with analyzing the current industry standard verification methods. Most of the verification happens with simulation-based methods, where the verification engineer simulates the design with different inputs until a reasonable coverage is obtained.

However, we have simulation-based verification environment for Debug IP which cannot be ruled out suddenly. We are using Formal verification for the things which could not be achieved through simulation and bringing up formal verification for the IP step by step. It is not possible to simulate the design with all possible cases of input and hence these methods cannot guarantee a complete bug free design. Here the formal methods outperform simulation as these methods will give complete proof for the design that it never violates a particular rule. But since these methods are relatively new, considerable amount of time had to spend in understating its theoretical background and methodology. An industry standard tool named Jasper Gold from Cadence is chosen for this project. Initially, an understanding about the methodology as well as the tool has. Also, frequent communication was there between the tool vendor as well.

CHAPTER 2

THEORETICAL BACKGROUND

Strong knowledge of background and theory is essential to have a deep in- sight into the details of the project. It helps us to get a better understanding and relevance of the work. It is also important for us to know the areas where the project work can be applied. Formal method is a semi-automatic way of proving the correctness of the design by using mathematical techniques. A powerful tool is needed to do these complex operations and there by the understanding about the formal tool is necessary. All the formal tools use some intelligent algorithms to convert design written in HDL into a mathematical form and evaluates the correctness of it.

2.1 Cadence Jasper Gold

Different EDA (Electronic Design Automation) companies have developed different formal tools. Out of those tools one of the most accepted one in industry is the Jasper Gold from Cadence. This is the standard tool recommended for formal related activities within Intel. The Jasper Gold is a verification platform from Cadence which contains different formal applications which can be used in different formal scenarios. This platform contains different Apps which are designed as targeted solutions to address different challenges in both design and verification phase of design cycle. Depending upon the role and the challenge that an engineer faces, he/she can choose any of the built-in Apps to find the solutions.

2.2 Jasper Gold for Design

A design engineer can use the Jasper Gold platform to increase the productivity. The basic use-case is the Formal Property Verification App for unit testing. It is not required and

designers don’t need to write test bench code, instead he can directly load his module or block to the Jasper Gold GUI (Graphic User Interface), and the tool itself will drive the inputs. It is possible to constrain the input values if needed and the user can interactively debug the design by editing the waveforms. This live waveform editing is the handiest feature which helps in easy debugging of the design. The platform includes formal- based technologies dedicated to better meeting designers’ needs for register-transfer level (RTL) signoff. Designers benefit from richer functional checks and formal-powered intelligent debugging to reduce violation noise. The Jasper Gold Superlint App and Jasper Gold Clock Domain Crossing (CDC) App improve design quality and reduce IP development time when compared to existing solutions with static rules-based checkers. With these applications, designers can signoff robust, reusable, and CDC-clean RTL code to the verification and implementation phase, shortening overall time to market and significantly improving design quality.

2.3 Jasper Gold for Verification

The Jasper Gold platform provides a wide range of formal verification apps, which will considerably increase the productivity. The very basic one is the formal property verification app, which checks for the valid behaviors of the design mathematically. The tool converts the design into mathematical space and evaluates the properties over this space. For the custom design the user must provide these two properties needs to be proven. But for connectivity or control and status register verification, the tool itself can create standard set of properties which is very convenient and less time consuming for the user. These apps can be easily used by verification engineers and formal verification specialists. Formal apps do exhaustive verification without writing the test benches, hence achieves significant savings in the time required for verification. In simulation-based methods considerable amount of time is spent on creating the test benches and bringing up the verification environment. This time can be saved in formal, and hence it is possible to find out bugs in early stages compared to simulation. The below list shows the all the available apps in the Jasper Gold platform. The users can choose between the

apps listed below depending upon their need.

- Jasper Gold Formal Property Verification App
- Jasper Gold Sequential Equivalence Checking App
- Jasper Gold Design Coverage Verification App
- Jasper Gold Coverage Unreachability App
- Jasper Gold X-Propagation Verification App
- Jasper Gold Control and Status Register App
- Jasper Gold Connectivity Verification App
- Jasper Gold Superlint App
- Jasper Gold Behavioral Property Synthesis App

2.4 Jasper Gold Formal Property Verification App

This is the classical and most used app in the formal environment. It fully verifies block level properties and high-level requirements. It enables exhaustive and complete verification and provides quick bug detection as well as end-to-end full proofs of expected design behavior. With its powerful analyzing capabilities and ease of use, the app is ideal for early-stage bug hunting as well as for ensuring the highest possible confidence in design functionality via end-to-end full proofs. Another important feature which makes this app more useful is the live waveform edit feature which allows the user to edit the waveform on the fly which will speed up the debug significantly. Also, there is a feature which called as quiet trace which gives minimum signal transitions to reach to a counter example. These capabilities make the Jasper Gold Formal Property Verification App ideal for early-stage bug catching. Additionally, the included state space tunneling technologies can accelerate the proof convergence process for complex high- level properties. Also, the tool itself comes with different complexity

methods which help in convergence of complex properties. Key features of this app are listed below

- High-performance formal engines exhaustively prove complex assertions utilizing advanced formal techniques.
- Formal scoreboard supports verification of data-path designs, tracking end to end data integrity by detecting dropped, duplicated, or corrupted data.
- The app supports System Verilog Assertion (SVA) or Property Specification Language (PSL) properties, Verilog or VHDL designs under test (DUT).

2.5 System Verilog Properties

The formal engines convert the design under test into a mathematical model and evaluates the rules or behaviors that the system should obey on top of it. The rules or behavior that should be obeyed by the design has to be given by the user. This is usually specified in System Verilog Assertions (SVA) language [3]. The properties/sequences are used for the following.

- Assertions
- Assumptions
- Covers

Assertions:

Assertions are the system Verilog constructs which has checks for the behavior of a design, and it is supposed to be true at all valid states of the design. If at any valid state the assertion fails, then it indicates the invalid behavior of the design. The assertions can be a simple Boolean expression, or it can be a complex temporal definition of signals. In formal property verification environment, the assertions are treated as proof targets and the tool will try to prove mathematically that the design never violates these assertions. In FPV terminology, the failure of the assertion is called as counter example [10] and

usually, the tool will provide a waveform with the failed scenario.

Assumptions:

By default, the FPV tool treats all inputs as un-constrained, and they are free to change the values randomly. But in some scenarios, this may not be the indented behavior. The inputs should be following some order and specification or in other words the inputs should be constrained to follow a behavior. These constrains are given by assumptions. The FPV tool will hold these assumptions as true always in the environment. Assumption is like assertion, assumption can be simple Boolean expressions, or it can be complex properties as well.

Covers:

Covers are the third and final category of SVA properties. A cover specifies a condition which the design might undergo. This is usually used to check whether some interested scenarios of corner cases have occurred at least once under current set of constraints. The FPV tool will try to see if the condition or statement given in the cover is executed at least once in the current environment and if it doesn't find any scenario then the cover is shown as fail. This implies that the current constraints are over-constraining the design or there may be issue in the design. Also, may be the condition given as the cover is not a valid one.

2.6 SVA language basics

The SVA properties can be written in different levels of complexity. The hierarchy is shown in Fig. 2.1

- **Booleans:** These are the basic statements with a simple Boolean expression.
  Example: - idle_req && timer_req
- **Sequences:** These are the Boolean expressions with a definite time dependency, or it specifies the conditions over a period. These sequences are associated with a clocking block to define the time unit.
  Example: - timer_req ##2 timer_ack
  This means that once the timer_req is high, timer_ack is going high in two cycles.

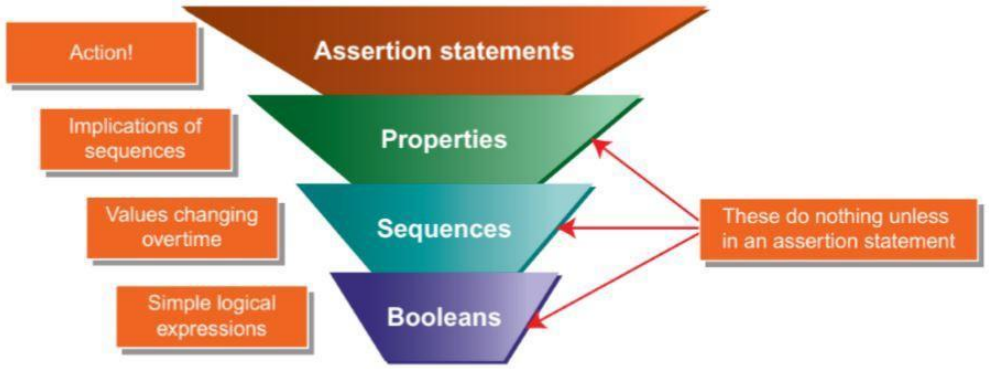


Figure 2.1: SVA Properties Hierarchy

- **Properties:** These are the sequences combined with operators such as implication operator. Usually it has two parts, the antecedent and the consequent. They relate to an implication operator. The consequent will be evaluated once the antecedent is true. There can be temporal dependency between antecedent and consequent as well.

  Example: - (timer_req && valid) -> ##2 timer_ack

- **Assertion Statements:** These are the statements written using assert, assume or cover phrase with an SVA property in it. This makes the FPV tool to treat that property as an assertion, assumption, or cover point.

  Example: - assert property (timer_req && valid) -> ##2 timer_ack

2.7 Jasper Gold Sequential Equivalence Checking App

2.7.1 Introduction to SEC

Equivalence checkers can be either sequential equivalence checkers or combinational equivalence checkers. Combinational equivalence checkers evaluate whether two designs, specification (spec) and implementation (imp), are logically equivalent. They check whether two designs have the same functional behavior at all external ports and in- ternal state elements. They require that the two designs are state-matching and perform equivalence checking only on the remaining logical cones.

In comparison, sequential equivalence checkers like the Jasper Gold SEC App check whether two designs have the same functional behavior at all external ports. They can work on both state-matching and non-state-matching designs, and they can work even in the presence of internal sequential differences.

As an example of the specification (sometimes referred to as the "Golden" model) and implementation design, consider a design (spec) that is introduced with new feature or clock gating changes for power optimization, which differ from the original spec, but should not impose any functional behavior change. The resulting design is the imp. The SEC App checks whether the spec and imp implement the same set of behaviors.

To perform this verification, the SEC App connects the two designs into a miter model. In this model, the boundary inputs of the two designs are connected in some sense (that is, by actual connection, assumptions, or temporal functional relations) to allow a comparison of their outputs.

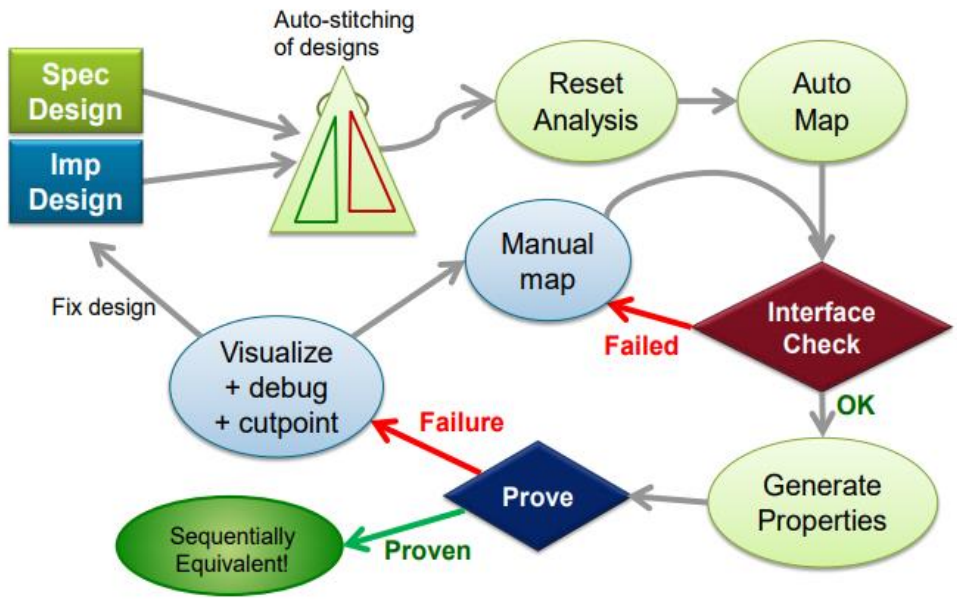


Figure 2.2: SEC workflow

Figure 2.2 explains workflow of SEC. Auto-stitching of designs i.e., spec and imp, Signal mapping and its interface check are done. The properties are generated and proved, failing assertions are debugged and fixed (fix is for either RTL or mapping, based upon failure reason). Once all the properties are proved, it is signed off and we can conclude

that the spec and imp are equivalent.

2.7.2 SEC App Use Models

The SEC App has a variety of use models. Any comparison activity can be considered a general SEC use case. Following are some popular use models.

- New feature addition
- Clock gating insertion
- Re-partitioning and pipeline retiming
- Code cleanup
- Parameterization
- CPU lockstep verification
- Reset optimization

CHAPTER 3

FORMAL METHODS FOR VERIFICATION

Formal verification starts with creating the design state space. The design state space consists of the values for each state element (flip flops, memory elements, and latches) plus the values of all possible input signals. Each state element or input can have values such as 0 or 1. So the total number of conceivable states for a design will be $2^{(n+i)}$, where 'n' is the number of state elements and 'i' is the total number of input bits. All these states collectively from a state space, which cover all the possible states a particular design can have at any time. Thus, the design state space is equivalent to simulating the design with all the possible input combinations. But even for small designs the state space will be huge and thus it is not possible to cover all these states using simulation. What formal engines will do is, that it will represent this state space using some efficient mathematical models and hence it will be feasible to check conditions or rules on all of the states. As the design operates it transits from one state to another on each clock edge. This can be seen as jumping from one state to another in the state space. So, all these jumps form a trajectory in state space. In normal simulation one run evaluate one possible such trajectories, and it is impossible to cover all such possible trajectories. But formal tools effectively analyze all possible trajectories in parallel and hence considerably increasing the performance and coverage.

3.1 Advantages of Formal Verification (FV)

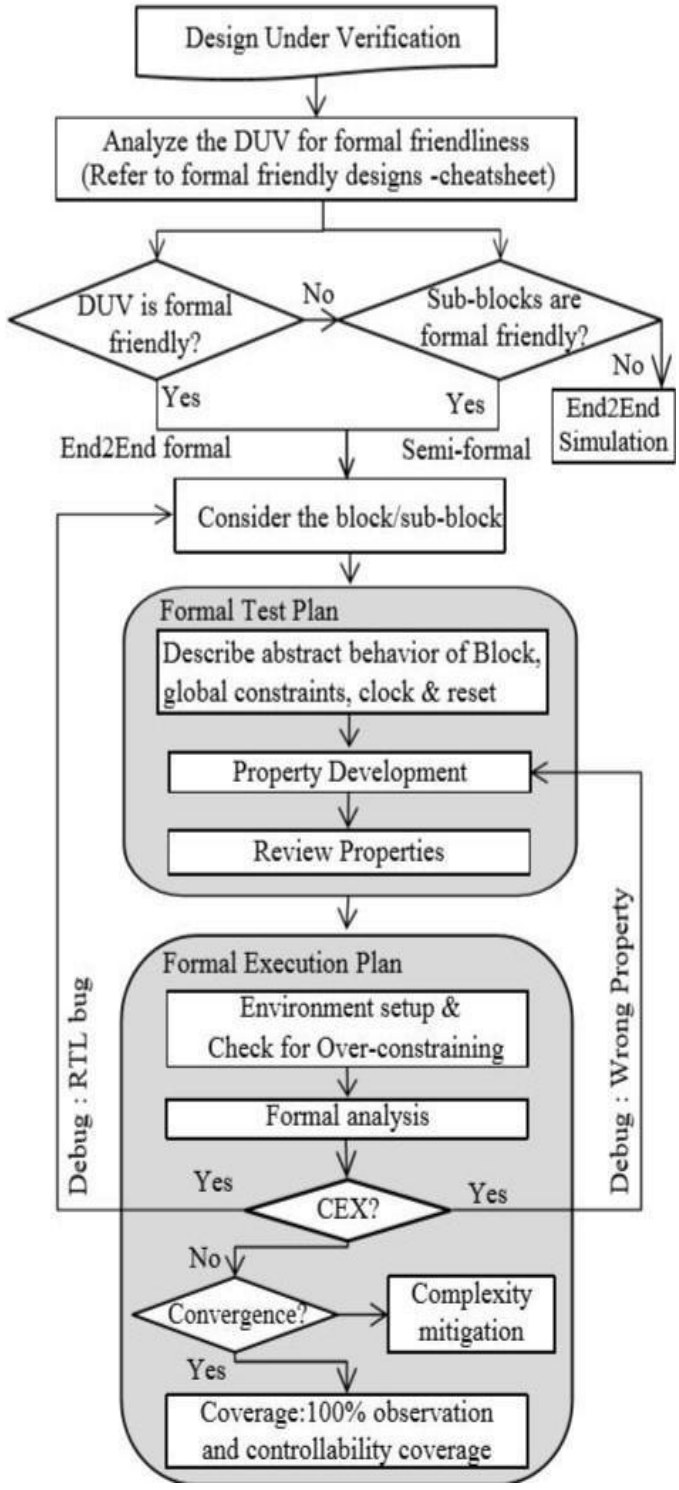
Formal methods analyze the RTL behaviors in a mathematical space. This approach outperforms traditional simulation-based methods in many ways. Some of the advantages of formal methods are:

- **Solving the Right Problem:** The ideal way of verifying a design is by mathematically proving that they meet the specifications. The traditional methods,

simulation and emulation methods are due to technological limitations. But these limitations are now lessening, and thus there should be a transition from these old methods to new ones.

- **Complete Coverage:** Coverage is the proportion of design behaviors analyzed. FV methods inherently provide complete coverage on the design. It may not always possible to get 100% coverage in formal especially when the design is complex [11]. But even if it is the subset of all available states, it is equivalent to running exponential number of simulations.
- **Minimal Examples** Formal methods are quite useful in analyzing corner case scenarios, but in simulation it will be very difficult to hit those cases. Also, the simulation might take many numbers of cycles to reach to that condition. But usually, the formal methods are able to get the required behavior in minimum number of cycles. Also, it is possible to see the condition with minimum signal transitions and this will significantly reduce the debug time.
- **Corner Cases:** In simulation the user drives the input, but irrespective of the effort put in deciding the input vectors some of the corner case scenarios can be missed. In formal, by default all the possible combinations and sequences of inputs are applied and hence the possibility of missing a scenario is less.
- **State-Driven and Output-Driven Analysis:** In formal methods, there is no need to drive anything to reach a particular scenario. It is possible to specify any interested scenario at output or at an internal signal value and the tool will find a way to reach that scenario. This is in contrast with the simulation where the user needs to drive proper inputs to reach the condition they want or in other words in formal, the user can concentrate more on the destination rather than the journey.
- **Understanding Infinite Behaviors:** Since the formal methods are using mathematical models to analyze the design, it can find out the model behaviors which are time unbounded in nature. Formal can analyze scenarios like whether a design can get stuck forever without reaching a particular state or it can analyze some behavior which may not be having a finite time limit.

3.2 Formal Flow



.

Figure 3.1: Formal Flow [19]

First step in formal verification identifies the suitable block/sub-block for formal. In formal method, there is no need of any kind of test bench, design functionality is verified using properties. The exclusion of unwanted values of input or fixing required values too is done by writing the assumptions. This is usually done in System Verilog Assertions (SVA) language. In addition to assumptions, assertions are coded/written to check the functionality of the design. In other words, the user must provide the rules which is to be obeyed by the design and the tool will exhaustively check those for all possible scenarios. These assertions are mostly written in System Verilog Assertions language. It is even possible to write covers to see whether a particular scenario or condition has met or not. The assumptions, assertions and cover points are generally known as formal properties and these play an important role in the successful completion of the formal verification for a design unit[9]. Formal flow starts parallel with design, once both are completed the design is to bound with formal properties with 'bind' SV construct. Alternatively, a top wrapper for design can be designed in which properties can be coded.

The Design with its bind file of properties is loaded to tool with help of TCL script. Firstly, the most important thing is writing more covers for each signal to verify the sanity check of the design. If any of the property fails in formal tool, then its gives a counter example (CEX) for which the property is failing. It's easy to debug with help of CEX. It may not be RTL issue always when we see a CEX, many times it is due to error in the property coded. The properties should be reviewed and modified. Also, the tool provides information of over-constrained design and the assumptions which over-constrain it. Sometimes, the properties are not proven i.e., neither pass nor fail, they remain undetermined. This is referred as convergence issue which may arise due to large design size or under-constrained design. Formal has different ways to overcome non convergence. The assertion which is not converged can be concluded as pass depending on the bound it reaches. It is coined as bounded proof.

CHAPTER 4

Digital Memory Decoder

4.1 Introduction

Formal verification is an umbrella word for a group of methods that use static analysis based on mathematical changes to establish the accuracy of hardware or software behavior. This is in comparison to dynamic proof methods such as simulation. As design sizes and modeling durations grew, verification teams sought methods to decrease the number of vectors required to train the system to an appropriate level of coverage. Because it is not necessary to assess every conceivable state in order to show that a given piece of logic satisfies a given set of characteristics under all circumstances, formal verification has the potential to be extremely quick. However, the sort of logic it is used with and how it is implemented have a significant impact on how well it performs. As the formal methods are advanced, different languages are used to write the properties. As a standard language System Verilog Assertions (SVA) [3] is matured in parallel, which enable the easy and standard way of writing properties. Physical digital memory decoders are a crucial component of many digital integrated circuits. They are responsible for translating address inputs into corresponding memory locations. It is important to ensure that the decoders are reliable and accurate, as incorrect operation can lead to undesired system behavior. Traditional simulation techniques have been used to verify the correctness of such components, although they are not always ensuring that the system is correct. Formal verification techniques can be used to ascertain the correctness of the system with respect to a given specification. Physical digital memory decoders are essential components of computer systems, as they are responsible for accessing various memories like RAM, ROM, Flash, EPROM etc. and are utilized by various resources like micro-controller, direct, indirect etc. Memory decoder, which is basically an arbiter, is used by different resources to access various kinds of memories for read/write operation. Different memories that can be accessed by memory decoder are DRAM, IRAM and FRAM. Apart from accessing memories it can also access near registers like Control and status registers, port registers etc. which can be accessed in a single clock cycle and can access far registers

like analog registers, which take four clock cycles. Memory decoder uses priority multiplexers to provide access of memories and registers to resources. In this module we have several resources that can request for access to memories. Some of the resources are direct, Indirect, µc Dram, Lane resource, etc. Figure2 represents block diagram of digital memory decoder along with memories and resources. In this module each resource is assigned with priority. So, memory and registers located on right hand side in the block diagram, have their corresponding priority multiplexers. Each storing element has its own specific address and based on this address, resource can access it.

To test whether the design so made meeting all the specifications as required, in general, two techniques have been followed viz. traditional design flow and formal verification. In traditional design, emphasis is laid on accomplishment through simulation and testing. However, at times, exhaustive testing for non-trivial devices is not feasible and thereby only probabilistic assurance can be ascertained. Whereas Formal verification provides a holistic view pertaining whether design meets all or parts of specifications as it uses complex and rigorous mathematical reasoning.

To apply formal verification to memory decoders, a formal model of the decoder is created. The model specifies the behavior of the decoder under various input conditions, as well as the desired output. The model is then analyzed using model checking to verify its correctness. This involves checking that the model satisfies a set of specified requirements, such as the absence of glitches or the correct output for specific input patterns. One challenge of using formal verification for memory decoders is the complexity of the model. The number of states in the model can be enormous, making it computationally expensive to analyze. To address this issue, a novel abstraction technique is proposed to reduce the number of states in the model while preserving its behavior. This abstraction technique enables more efficient verification of the model, reducing the time and computational resources needed to verify its correctness.

Digital Memory Decoders are formally verified using the Jasper Gold FPV technique. Here assertion-based approach is used to verify the functionality of the decoder. Different types of resources and memories are used whose behavior is described in the section 4.2.1.

4.2 Internal Structure of Digital Memory Decoder

The internal architecture of Digital Memory Decoder is as in Figure 4.1. Memory decoder, which is basically an arbiter, is used by different resources to access various kinds of memories for read/write operation. Different memories that can be accessed by memory decoder are DRAM, IRAM and FRAM. Apart from accessing memories it can also access near registers like Control and status registers, port registers etc. which can be accessed in a single clock cycle and can access far registers like analog registers, which take four clock cycles. Memory decoder uses priority muxes to provide access of memories and registers to resources. In this module we have several resources that can request for access to memories. Some of the resources are direct, Indirect, µc Dram, Lane resource, etc. Fig 4.1 represents block diagram of digital memory decoder along with memories and resources. In this module each resource is assigned with priority. So, memory and registers located on right hand side in the block diagram, have their corresponding priority muxes. Each storing element has its own specific address and based on this address, resource can access it.

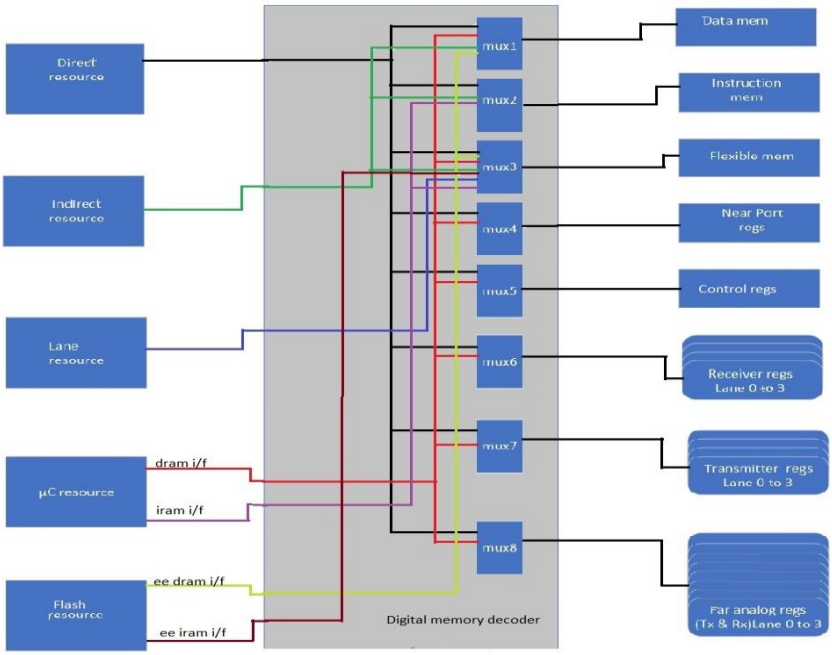


Fig 4.1: Block Diagram of Digital Memory Decoder

4.2.1 Different Types of Memories and Registers Used

Digital Memory Decoder is used by various resources to access different types of memories like DRAM, IRAM and FRAM and different types of registers like status registers, port registers, far registers etc. Detailed discussion about various memories is given below:

A. DRAM

Data RAM (Random Access Memory) is a type of memory that can be accessed randomly, meaning that any location in memory can be read or written in any order. It is a volatile form of memory, meaning that its contents are lost when power is removed. Data RAM can be used for both primary storage and for short-term storage of data

B. IRAM

IRAM (Instruction RAM) is a type of RAM that is used to store instructions that are being executed by the CPU. It is used to store instructions that are frequently accessed, such as those used in loops or branches. IRAM is typically faster than other forms of RAM and is used to speed up program execution. The instructions stored in IRAM are typically accessed and executed more quickly than those stored in main memory, which allows the CPU to process them more quickly. This can lead to a higher performance of the system overall.

C. FRAM

Flexible RAM (FRAM) is a type of non-volatile memory that is designed to be flexible, allowing it to be used in a variety of applications. It combines the features of RAM and ROM, allowing it to retain data while also providing the ability to read and write data at high speed. FRAM has the advantage of being fast, low-power, and reliable, and can store data for up to 10 years without power. It is also resistant to changes caused by magnetic fields, making it ideal for use in environments with high levels of electromagnetic interference.

D. Far registers

Far registers are basically the analog registers which can be accessed by the various resources in four clock cycles. In one clock cycle only 8-bit data can be accessed by resource. So, to access total 32 bits, it requires 4 clock cycles.

4.2.2 Different types of resources Used

Digital memory Decoder is used by various resources to access different types of memories and registers. This section describes the behavior of various resources that are used in this module.

A. Direct resource:

- Pulse (req and gnt).
- It can wait (property to starve).
- Perform both read (after 1 cycle of gnt) and write (at same cycle).

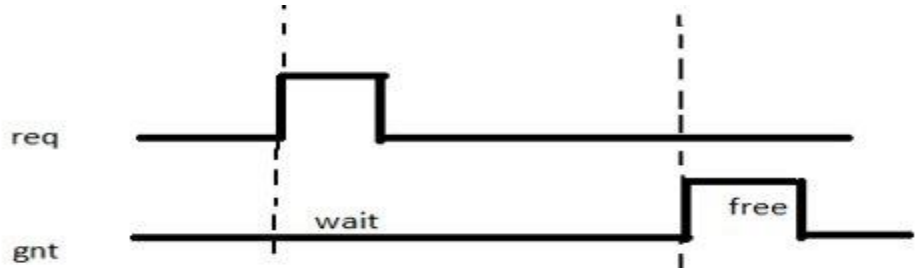


Fig 4.2 Request-grant handshake for direct resource

B. Indirect resource:

- Not a pulse.
- Starvation does not exist.
- Perform both read (after 2 cycles of gnt) and write (at same cycle).

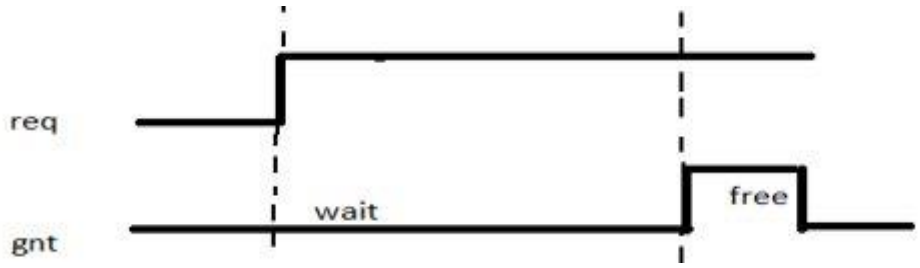


Fig 4.3 Request-grant handshake for indirect resource

C. µC DRAM resource:

- It does not have a property to starve.
- It doesn't have grant signal.
- Performs both read and write operation.
- It has a busy signal whose behavior is defined below (**for far regs**):
    - If it req and doesn't win, busy signal will be asserted in next cycle.
    - Whenever it is doing read transaction, depending upon the byte enable, busy signal is asserted (from next cycle onwards).
    - When it does write transaction, busy gets asserted after two cycles and duration of busy depends upon byte enable.

- **For near regs:**
    - If it req and doesn't win, busy signal will be asserted in next cycle.

D. Flash DRAM resource:

- Toggle signal
- Whenever it toggles, after 2 cycles, it generates a pulse from which tx starts.
- It can only perform write operation.
- It doesn't have any grant signal.

E. Lane resource:

- Working of Lane resource is same as the direct resource.
- It has also the property to starve, i.e., it will not initiate another request until it get response for the previous request.

4.2.3 Priority order of resources available in Digital Memory Decoder

Digital Memory Decoder is like an arbiter, and it decides for which resource gets the control in order to access the memories and registers present in the design. It must be noted that at a time only one resource can access the memory for read and write operation. If more than one resources demand for memory access, then control is given according to the priority of the resources. This section describes the priority order of the resources which is specified by the designer.

For DRAM 1 MUX:

Flash IRAM > Indirect resource > Direct resource > µC DRAM resource.

For DRAM 2 MUX:

Flash IRAM > Flash DRAM > Indirect > Direct > µC IRAM > µC DRAM > Lane resource.

For NEAR register MUX:

Target resource > µC DRAM resource.

For FAR register MUX (Analog registers):

Target resource > µC DRAM resource.

CHAPTER 5

Formal Verification on Digital Memory Decoders

In a typical functional simulation, every piece of the test bench is written manually. Below given are the points for testbench architectures in conventional simulations:

1. Test case - Stimulus is generated and it is decided when it should be injected the RTL.
2. Driver - The drivers or BFMs are written which do the injection of the stimulus based on a certain protocol.
3. Monitor - The monitors are written which receive the output from the RTL.
4. Model- Detailed reference model is a coded, a zero-time equivalent of the RTL, which produces a predicted result.
5. Checkers- The checkers and scoreboard care written which compare the output from the RTL versus that from the reference model and declare Pass or Fail.

The test bench and tests orchestrate everything, and the tool merely simulates on a clock-by-clock basis what's happening in the RTL as the desired stimulus is injected. Also, a lot of code needs to be written before the first test can be written. It may not always possible to get 100% coverage in simulation especially when the design is complex [5]. In Formal Verification, the tool does a lot of the heavy lifting. There is no concept of driver, monitor or test cases. Here,

1. The inputs or internal variables of the DUT are constrained according to the design specification using SVA assume directive.
2. Checkers are written on the desired outputs, or internal variables of the DUT, using SVA assert directive.
3. System Verilog and assertions cover property is used to collect functional coverage.
4. Small pieces of modeling code can also be written which are just sufficient for a particular checker (aux-codes). This is different from Functional Simulation where a detailed reference model of the block being verified is written.

5.1 Steps for verification of Digital Memory Decoder

In order to verify this design using Formal techniques, following are the steps that need to be followed.

- Understand the specification and create a test plan for the features outlined from the specifications.
- Review the test plan and modify it accordingly.
- Create a tcl file (a script file) for jasper tool setup.
- For memory decoder each register, and memory has its specified address. So, we do a memory mapping which is shown in Table 1.
- Write assumptions to avoid illegal inputs and to restrict the verification of entire module to a sub- module.
- Write auxiliary code to create critical scenarios required for verification and bind this code with the assumptions.
- After writing assumptions and auxiliary code, write some covers to verify whether environment is working correctly.
- Finally write the assertions to verify the behavior of the design.
- Proof assertions and covers in JG and check for vacuous PASS and covers.
- Debug the failed assertions that are also known as counterexamples (CEX) and find the root cause.

Memories/registers	Address range (in hexadecimal)
Data memory 1	2000 – 27FF
Data memory 2	1000 – 1BFF, 3000 – 3BFF
IRAM port A, B, C, D	0000 – 4FFF
IRAM port E	0000 – 37FF
Flexible memory	3800 – 3FFF
Port registers	0900 – 097F
CAR registers	0A00 – 0A7F
Rx registers Lane 0 to 3	0000 – 007F, 0200 – 027F, 0400 – 047F, 0600 – 067F
Tx registers Lane 0 to 3	0100 – 017F, 0300 – 037F, 0500 – 057F, 0700 – 077F
Analog Rx Lane 0 to 3	0080 – 00BF, 0280 – 02BF, 0480 – 04BF, 0680 – 06BF
Analog Tx Lane 0 to 3	0180 – 01BF, 0380 – 03BF, 0580 – 05BF, 0780 – 07BF

Table 4.1. Memory mapping for different registers/memories

5.2 Formal Techniques to verify Digital Memory Decoder

5.2.1 Formal Property Verification (FPV)

Formal property Verification is a technique which is used to verify the functionality of a design by using assertions only. In Digital memory decoder various safety assertions are written to verify the behavior of the decoder. FPV tool automatically generate all the possible scenarios for the inputs and try to fail the assertion even for the illegal combination of inputs. So, we have to constraints these illegal combinations using assumptions. Assertions to check read/write operation for resources, to check priority of various components, to check data integrity, to access the near and far registers, to check starvation case, to perform handshake protocols, etc., are used in this module [14]. Apart from assertions some other techniques are also used for optimization of formal techniques and are described below:

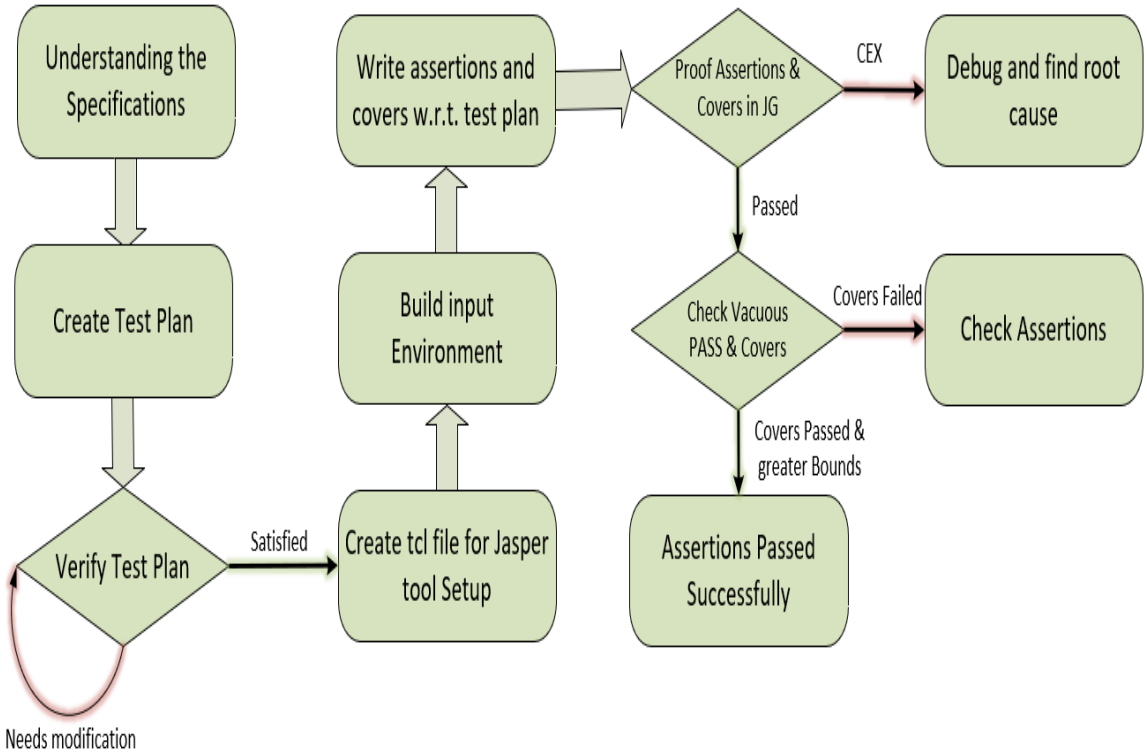


Figure 5.1: FPV Flow for this Project

5.2.2 Auxiliary Code

Auxiliary codes are the additional code (apart from RTL) that we use to build the environment for verification. In memory decoder, we write FSM code to convert liveness assertions to safety assertions. Liveness assertions take more time to run and bounds for these assertions are also high. So, to optimize are verification we use these codes. Let us consider an example, if direct resource asserts a request, then next request cannot be asserted before the acknowledgement (grant). If req is asserted and grant is not there, then state of FSM is WAIT else state is IDLE. In this case two scenarios are possible which are described as: First one is when request is asserted then grant is provided within same cycle and second one is that when request is asserted, it may be possible that master is busy so it may take four cycles to provide the grant to that request. So, for the FSM we have two states i.e., WAIT and IDLE and switching between those states depends upon the value of request and grant.

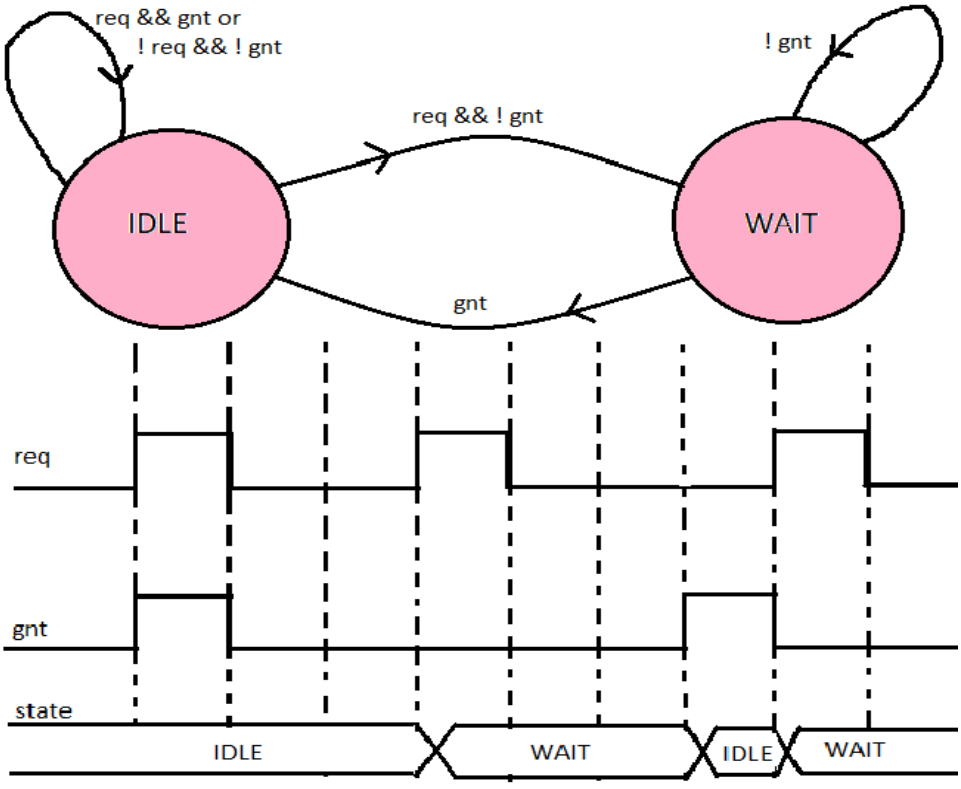


Figure 5.2: FSM for req/ack handshake

5.2.3 Counter Abstraction

Counter abstraction is another technique used to optimize the verification process. Using this, we can directly jump to the states that is of point of interest for us and all the skipped states are considered as one state. In memory decoder, we use counter abstraction to check value of write enable signal at the last address of instruction memory. If counter abstraction is not used, then first counter will increment the address from starting point to the last address of IRAM and then check for the enable signal. In this case bound is very high and assertions take more time to prove. Using counter abstraction, it will reduce to three states (initial address state, last address state and the address between initial and last address state) as shown in Figure 5.3. Counter abstraction technique is basically used in this design to reduce the time of verification and hence we can optimize our results in a faster way. Counter abstraction reduces number of states and focuses on the states which are point of interest for the verification of the Digital Memory Decoder. In figure 5.2, it is clearly visible that without using counter abstraction total number of states are (4'hFFFF-4'h0000) +1 whereas with counter abstraction technique these states reduced to three states which is far lesser than the states in former one.

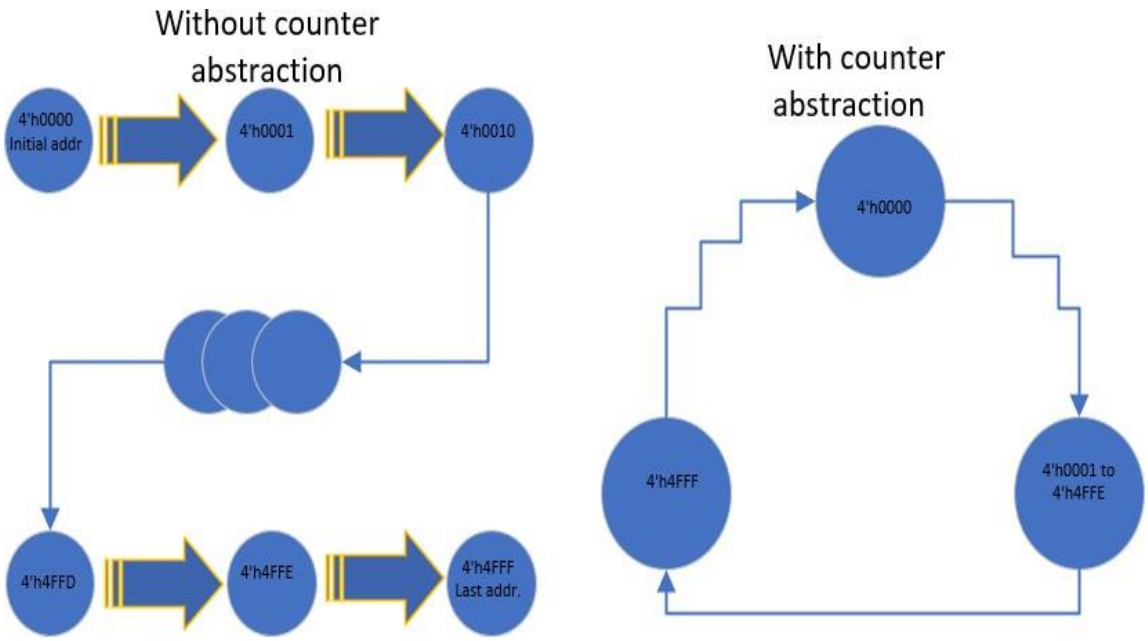


Figure 5.3. Counter Abstraction

5.3 Bugs found in the Design

Below is the list of bugs found in the digital memory decoder:

- ***Data Integrity is not maintained***

When request is asserted by the resource, it must perform read operation one cycle after the grant is received. But in this module the data obtained in the next cycle is not same as the data present on the previous cycle. Waveform for the above bug is shown in figure 5.4.

Assertion used to find this bug is:

assert property: req && rd_en |→ gnt ##1 (out_data = =$past(in_data))

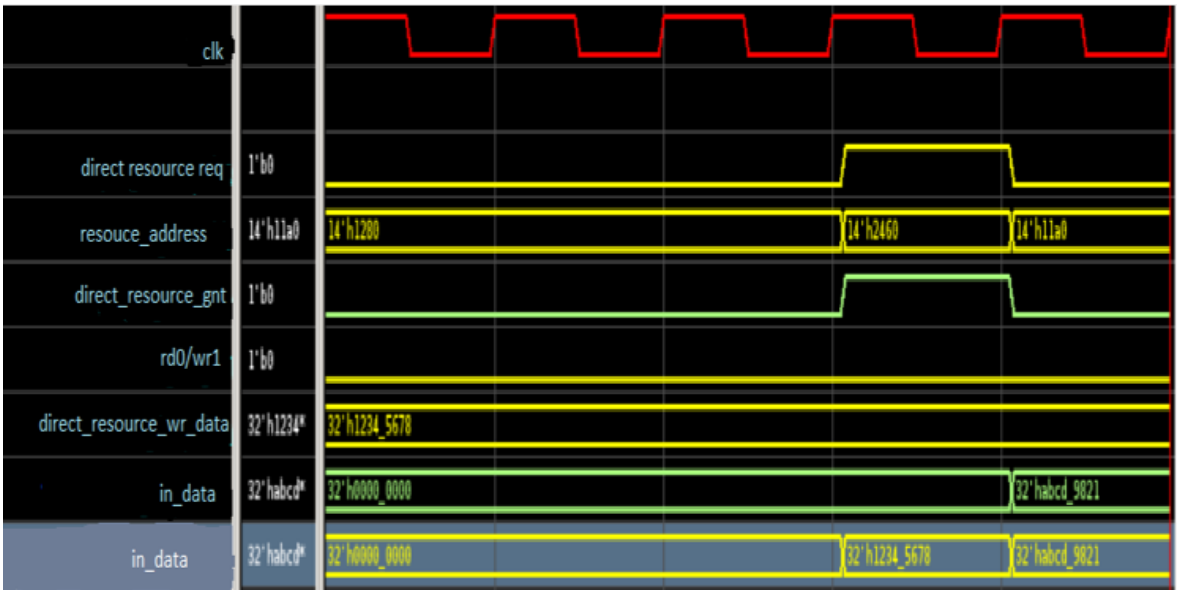


Figure 5.4. Waveform for failed assertion to check data integrity

- ***When indirect resource reads from dram2 memory then wrong data reads by source.***

When indirect resource reads from dram2 memory then wrong data reads by source. Instead of reading data from next cycle it takes 2 cycles to read data, where data integrity is lost.

Assertion used to find this bug:

assert property: (Indirect_req &&! indirect_dram0_iram1) && fv_dram2 (indirect_addr) &&! indirect_wr1_rd0 |=> (indirect_rd_data == dram2_data)

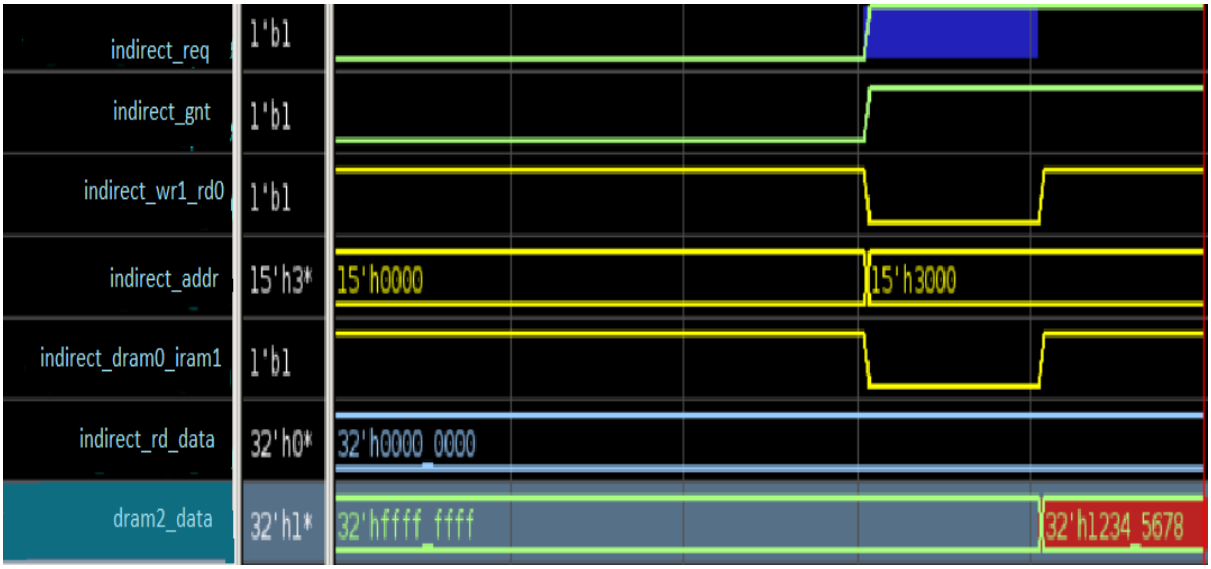


Figure 5.5. Waveform for failed assertion to check data integrity for indirect

- ***Indirect gets wrong data while accessing to 8kB fRAM space in Port E (part of DRAM 2).***

While indirect resource trying to write in DRAM2, it is unable to write in the same address of DRAM2 provided by it because in RTL designers fix the MSB bit to hard wired zero. So data is wrongly written in the given address.

Assertion used to find this bug:

assert property: indirect_req && indirect_wr1_rd0 && ((indirect_addr >= 'h3800) && (indirect_addr<='h3FFF)) && indirect_dram0_iram1 && !$past ($changed(ee_dvalid_toggle_tbt_phy_iram), 2)
|-> indirect_gnt && (dram2_ram_di == indirect_wdata) && (dram2_byte_wen == 4'b1111) && (dram2_wr_rd_addr == indirect_addr[13:2]) && dram2_wr_en && !dram2_rd_en
);

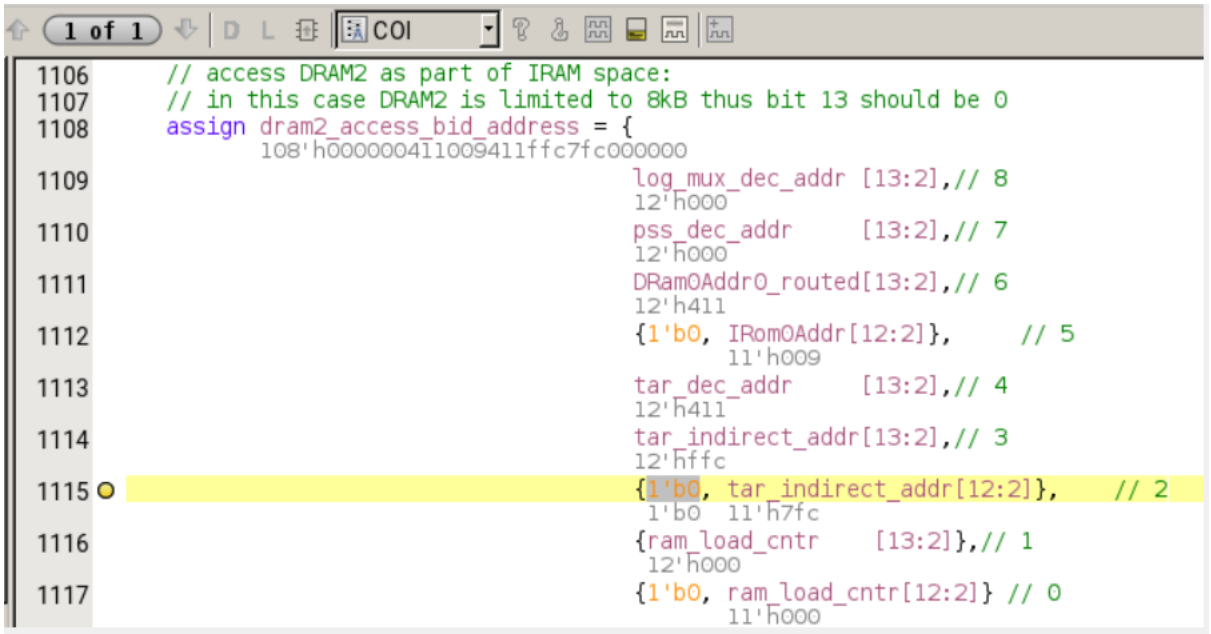


Figure 5.5: Snippet of wrong address allocation due to hard wired zero for indirect address

- ***Data lost due to interceding of previous transaction.***

When µc request is asserted for read transaction for far analog registers (takes 4 cycles, since µc integrate 8-bit far register data into 32 bits read data), data was not sampling in the desired clock cycle due to read operation performed in the previous transaction by the µc. In this case formal tool generates a scenario for the input combinations where uc_req is high for three cycles and then req is disable and after one cycle req is enabled. During this whole duration uc is reading the data. Functionality said that data should be received four cycles after when req is enabled high Continuously for four cycles. But in this case due to previous read transaction for three cycles, it interferes with the read data of next four cycles and hence read operation is not done correctly.

Assertion used to find this bug is:

assert property: µc_req && rd_en |→ ##4 (µc_out_data[31:24] = = $past(far_in_data) && µc_out_data[23:16] = = $past(far_in_data,2) && µc_out_data[15:8] = = $past(far_in_data,3) && µc_out_data[7:0] = = $past(far_in_data,3))

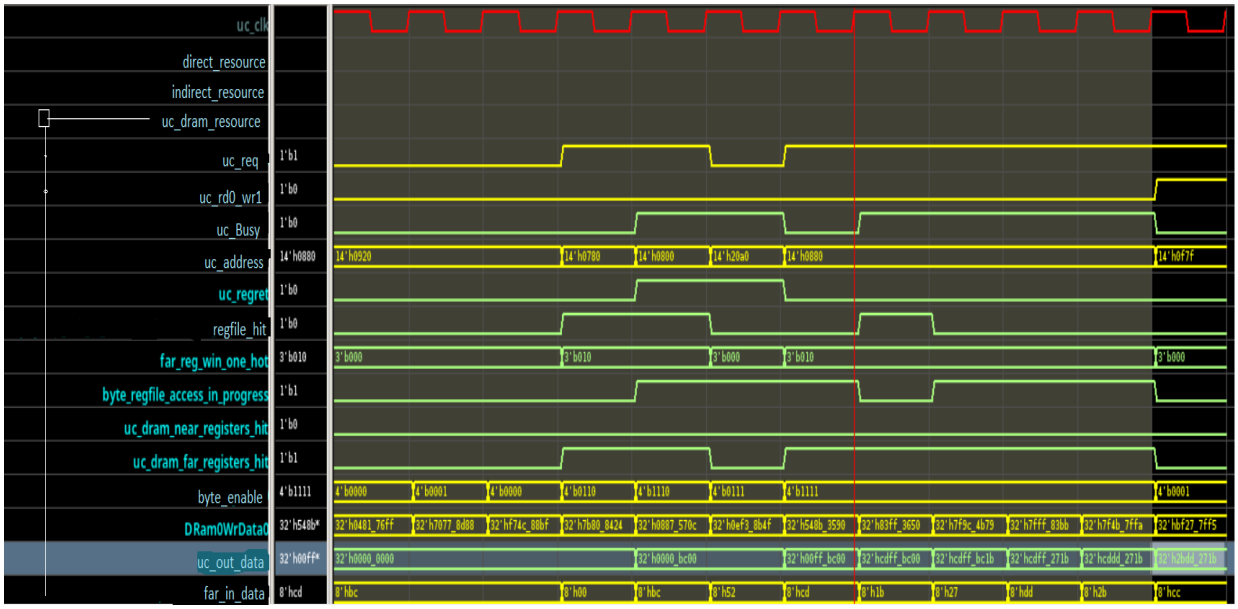


Figure 5.7. Waveform for failed assertion to check data lost due to interceding of previous transaction

- ***Priority checks***

When three resources try to access a particular memory in same clock cycle, then based on priority particular resource grant should be asserted which means neither any of three grants should occur at the same cycle.

Assertion used to find this bug:

assert property: $onehot0({tar_gnt, ind_gnt, lane_gnt}).

This check is also known as mutual exclusiveness check.

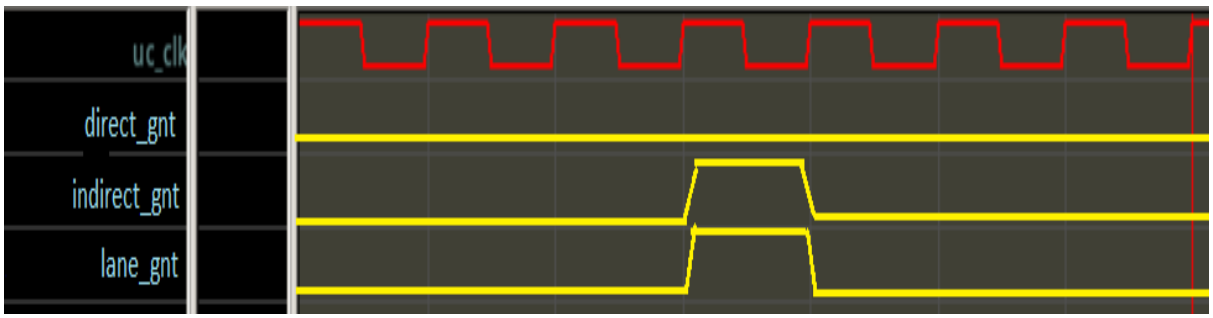


Figure 5.8. Waveform for failed assertion to check priority

- ***Overflow in read/ write transaction.***

Memory for iRAM port A/B/C/D allocated only up to 80 kB, but even after exceeding this space, source tar_indirect managed to do read/write transactions in this unused space. This bug is also known as overflow of data. Since indirect resource can write the data out of the range and this is because of the overlapping of the IRAM port space with the unmapped memory space. So extra data is written in the unmapped memory space.

Assertion used to find this bug:

assert property:(indirect_req && indirect_dram0_iram1) && indirect_gnt && indirect_wr1_rd0 |-> fv_iRAM_port_ABCD (tar_indirect_addr).

```
44 ### ============================================================
45 ### CLOCKS and RESETS
46 ### ============================================================
47
48 # clocks :
49 set clocks { uc_clk }
50 # resets :
51 set resets { !uc_rst_n}
52
53 ### How to define clocks with different factor and phase
54 ## clocks sclk {clk1 3 2} {clk2 4 1}
55
56 ### How to define FV internal clocks - require full path to top module
```

Figure 5.9: Snippet of inserting clock and reset signal in the tcl file.

5.4 Advantages of Formal Verification

Formal verification has many advantages over conventional simulation:

- One of the biggest advantages of formal verification is that it does not require any testbench. So, efforts to build testbench architecture in dynamic simulation is saved. All the input combinations are automatically generated by the tool in formal verification.

- Dynamic simulation takes more time to find corner case scenarios and FSM deadlock cases whereas in formal, tool automatically find the corner cases in fraction of time. Data lost due to interceding of previous transaction is an example of corner case bug and by writing suitable assertion in formal, it can be detected in lesser time.

- For small modules (having flip flop up to 20 K), formal gives much faster results and is preferred to find the bugs in the design at initial level. Formal verification helps to find the potential bugs in the design even before the simulation test environment is created.

- Cost-Effective: Formal verification can be more cost-effective than traditional testing methods. Traditional testing methods involve testing the system under different input conditions, which can be time-consuming and expensive. Formal verification, on the other hand, involves creating a mathematical model of the system, which can be used to verify the system's correctness under all possible inputs.

- Rigorous Verification: Formal verification techniques provide a rigorous way to verify the correctness of a digital system. Formal verification involves creating a mathematical model of the system and proving that it satisfies a set of specified requirements. This approach guarantees that the system is correct for all possible inputs.

- Early Detection of Errors: Formal verification can detect errors early in the development process. By verifying the system's correctness at an early stage, formal verification can detect errors before they become more difficult and expensive to fix

CHAPTER 6

RESULTS

6.1 Results

Formal verification found to be useful to verify the functionality of the Digital Memory Decoder features during bring up itself.

- Formal verification along with simulation added confidence in verification of the features/block.
- Formal Property Verification helped in verifying functionality of decoder based on Safety and Liveness assertions.
- Through Counter abstraction technique, time to prove the assertions reduces and efficiency of the verification increased.
- Formal Verification helped to find out the corner cases bug in very efficient way.
- Using Auxiliary Code, Liveness assertions were converted to Safety assertions and hence time to verify the module reduced.
- Large number of scenarios were possible to cover for verifying all the requests in grey window.

6.2 Challenges

The first and foremost challenge faced in doing formal verification is constraining inputs of the Design in module level. Also, identifying the vacuous passes of the assertions which may have occurred by over-constraining the Design.

6.3 Limitations of FPV

In the current industrial scenario, even though formal seems to be a promising option it has some limitations as well. Some of the prominent limitations are discussed in this section. One of the major limitations is that the tool only works with synthesizable RTL. As all the standard functional simulation methodologies are object oriented, it is not possible to directly use them in the formal environment. The convergence is another area of concern. As the design size increases the number of states will increase exponentially and this will affect the performance. The formal tools work best with designs which are having around 20,000 to 40,000 flip flops. If the size is more than that the design may take too long to converge and sometimes it will not converge at all. Due to these reasons, it is never possible to run formal at full chip level. But as the software technologies are advancing and better algorithms are developing these issues are expected to solve soon.

6.4 Good Design Candidates for Formal

Concurrency and multiple data streams, which are difficult to completely verify using simulation.

- Arbiters, On-chip bus bridge and Power management unit
- DMA controller
- Host bus interface unit
- Scheduler, implementing multiple threads
- Virtual channels for QoS
- Interrupt controller, Memory controller and Token generator
- Cache coherency, Credit manager block and Standard interface (AMBA®, PCI Ex- press. . .)
- Proprietary interfaces and Clock disable unit

CHAPTER 7

CONCLUSION

The intention of this project was to do a Proof of Concept (PoC) on formal methods in the context of end-to-end IP verification. The work is carried out successfully and identified that the formal methods are very effective in finding bugs at initial stages of the design. Formal methods can catch corner case bugs which may not be easily identifiable in simulation. Additionally, the time and effort required for the formal verification is very less compared to simulation. However, the drawback of formal methods is the inherent in ability to handle the complexity especially when the design is large. If there are properties to be converged, it will be difficult to sign off the IP as there can be still bugs present. As the verification being an important phase in the SoC design cycle, the correct strategy would be the combined use of formal and simulation methods together. Formal Verification is very effective in initial phases and finding deep corner case bugs, but simulation-based methods do a better job when the design complexity is huge. Probably in near future, a dedicated team might form for exclusive formal verification activities.

REFERENCES

[1] V. M. A. Kiran Kumar, A. Gupta, and R. Ghughal, “Symbolic trajectory evaluation: The primary validation vehicle for next generation intel processor graphics fpu,” in *2012 Formal Methods in Computer-Aided Design (FMCAD)*,Oct 2012, pp. 149–156.

[2] C. Rodrigues, "A case study for Formal Verification of a timing co-processor,"*2009 10th Latin American Test Workshop*, Buzios, Rio de Janeiro, 2009, pp. 1-6.

[3] V. Ashok and B. Mehta, “system Verilog assertions and functional coverage, springer,” July 2013.

[4] Wen-Shiu Liao and Pao-Ann Hsiung, “FVP: a formal verification platform for soc,” in *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.* March 2003, pp. 21–24.

[5] P. Basu and S. Das, and A. Banerjee, “Design-Intent Coverage-A New Paradigm for Formal Property Verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*,vol. 25, no. 10, pp. 1922-1934, Nov. 2006

[6] J. Hassen and S. Tahar, “Formal verification of an SoC platform protocol converter,” *IEEE Int. Symp. Circuits Syst.,* Sept. 2004.

[7] M. B. Slimane, I. B. Hafaiedh, and R. Robban, “Formal-Based Design and Verification of SoC Arbitration Protocols,” *IEEE Des. Test. Comput.*, vol. 34, no. 5, pp. 54-62, 2017.

[8] D. Bustan, D. Korchemny, and E. Seligman, “System Verilog Assertions: Past, Present, and Future SVA Standardization Experience,” *IEEE Des. Test. Comput.*, vol. 29, no. 2, Nov. 2012.

[9] E. Cerny, S. Dudani, D. Korchemmy, and L. Piper, “Verification case studies: Evolu--tion from SVA 2005 to SVA 2009,” *in Proc. Design and Verification Conf. (DVCon),* 2009.

[10] S. Mitra, A. Banerjee, P. Dasgupta, P. Ghosh, and H. Kumar, "Formal Guarantees for Localized Bug Fixes," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 8, Aug. 2013.

[11] R. Jones, J. Seger, and M. Agaard, "Practical formal verification in microprocessor de- sign," *IEEEDes. Test. Comput.*, vol. 29, no. 2, 2012.

[12] S. Erik Seligman and A. K. Kumar, Formal verification: *An essential Toolkit for modern VLSI design.* United States: Morgan Kaufmann Publishers, 2015.

[13] G. Swamy and R. Brayton, "Incremental formal design verification," *in Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 458-465, 1994.

[14] O. Sokolsky and S. A. Smolka, "Incremental model checking in the modal mu-calculus," *in Proc. Computer-Aided Verification (CAV)*, pp. 351-363, 1994.

[15] A. R. Bradley, F. Somenzi, Z. Hassan, and Y. Zhang, "An incremental approach to model checking progress properties," *in Proc. Formal Methods in Computer-Aided Design (FMCAD)*, pp. 144-153, 2011.

[16] T. Shiple, K. Harer, J. Kukula, R. Damian, and V. Bertaccor, "Smart simulation us--ing collaborative formal and simulation engines," *in Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 120-126, 2000.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lua, and H. Veith, "Counterexample - guided abstraction refinement," *in Proc. Computer-Aided Verification (CAV)*, pp. 154-169, 2000.

[18] P. Basu et.al, "Formal verification coverage: Computing the coverage gap between temporal specifications," *in Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 198-203, 2004.

[19] P. Basu, S. Das, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, L. Fix, and R. Armoni, "Design-intent coverage new paradigm for formal property verifi--cation," *IEEE Trans. on Comput. -Aided Design of Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1922–1934, Nov 2016.