# Exploration of formal verification in GPU hardware IP

NISHANT GUPTA

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

# Exploration of formal verification in GPU hardware IP

Nishant Gupta
ni4086gu-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors:
Erik Larsson, LTH
Sadat Rahman, ARM

Examiner: Christian Nyberg

October 10, 2019

© 2019
Printed in Sweden
Tryckeriet i E-huset, Lund

# List Of Acronyms

**AXI** Advanced Extensible Interface.

**BDD** Binary Decision Diagram.

**CH** Cache Hit.

**CM** Cache Miss.

**CTL** Computation Tree Logic.

**DUT** Design Under Test.

**FSDC** Fragment Shader Descriptor Cache.

**FSM** Finite State Machine.

**GPU** Graphics Processing Unit.

**IC** Integrated Circuit.

**IO** Input Output.

**IP** Intellectual Property.

**LRU** Least Recent Used.

**LTL** Linear Temporal Logic.

**RTL** Register Transfer Level.

**SAT** Satisfiability.

**SCM** Shader Context Manager.

**SVA** System Verilog Assertions.

**TCL** Tool Control Language.

**UVM** Universal Verification Methodology.

**VIP** Verification Intellectual Property.

# Glossary

**cache hit** When new pointer address value to FSDC is already present inside the cache.

**cache miss** When new pointer address value to FSDC is not present inside the cache.

**least recent used** It is a cache replacement policy algorithm which discards the least recent used item first.

**state space** A set of states a system can occupy.

# Abstract

Today, digital circuits are part of every ones daily life in form of mobile phones, computers, television, smart cards etc. The advent of new technologies such as internet of things, 5G etc. are continuously making the digital circuits more and more complex in design. With this increase in complexity comes the possibility of more bugs in the system. Finding and fixing these bugs is of paramount importance as it can lead to huge financial losses or can even be fatal especially in safety critical applications. Over the last few decades, traditional simulation based verification has been used as the default methodology to verify digital hardware designs. Although it has certain drawbacks such as:

- It is not exhaustive in nature meaning that it is very tough to cover all the possible input test vector as part of input stimulus being applied to the test bench.
- And due to its non-exhaustive nature, there will always be a possibility of missing bugs in the design especially the corner cases that can lower the design quality.

Recently formal verification has been evolved as an attractive and more comprehensive alternative to simulation based verification. In this master thesis work, model checking (alternatively property based verification) has been applied as a preferred formal verification technique on a module inside the Arm Mali GPU hardware IP in order to try and hit the corner case design bugs, if any. The module or DUT chosen for investigation in this thesis work is Fragment Shader Descriptor Cache (FSDC). This is a new module and mainly consists of control path logic which suits formal verification. The DUT has already been extensively verified by simulation based verification. The result shown later for this thesis work highlights that:

- Formal verification can be applied to a complete module as an alternative verification methodology.
- After the application of formal verification on DUT, 3 design bugs were discovered which were missed by the simulation verification.

Finally, based on above results and various challenges faced during this thesis work such as state space exploration problem or applying formal verification on complex parts of the design, it can be concluded that formal verification complements and at least provides a partial solution to the limitations of simulation based verification.

# Acknowledgements

I would like to take this opportunity to thank LTH, Lund University and Arm for giving me this opportunity to pursue this master thesis. Moreover, I would like to thank my supervisor at Arm, Sadat Rahman for his continuous support and guidance. I would also like to thank Yunus Ökmen and Henrique Martins for their great insight and patience to answer all my questions.

Last but not the least, I would like to thank my supervisor at the university, Erik Larsson and my examiner, Christian Nyberg for their great support and continuous feedback throughout this thesis work.

Nishant Gupta, Lund 2019.

# Popular Science Summary

The advent of transistors in 1947 revolutionized the field of electronics and paved the way for development of first IC in 1949 by the German engineer Werner Jacobi. In 1965, Gordon Moore predicted that number of transistor per IC will double every year while their cost will get halved, which famously later come to known as Moore's law. For later part of most of 20th century, there has been a tremendous growth in advancement of IC's in terms of speed, size and capacity. Almost following the Moore's law, the IC's of today are million times more faster and bigger in size then the ones in 1970's. Although IC's continue to grow in design and complexity, the engineers at the same time start to get haunted by the question of reliability of these designs.

Today advancement in machine learning, big data analytic, Internet of Things (IoT), autonomous vehicles are further fueling the innovation in hardware, thus, further worsening the challenges of test and verification of these designs. Verification almost account for nearly 70% of the overall design cost. Dynamic simulation has been the most popular choice of verification methodology for testing and verification of hardware designs. It has evolved over years by consolidating various diverse sets of methodologies into one now known as UVM. Although overall framework looks promising but it still has some way to go.

While almost in parallel a new verification methodology was evolving in background called formal verification. During this thesis work, application of formal verification on DUT will be explored mostly from the point of view that it can be applied as an alternative verification methodology on a complete module inside a system and also that it can help to try and hit the corner case bugs which might be missed by the simulation verification.

# Table of Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Today electronic systems are getting more complex in nature. The increasing design complexity increases the possibility of bugs in the design. As shown in figure 1.1, cost of fixing bugs increases a lot later in the development cycle, it is better to find bugs in the design as early as possible. Considering the increasing cost of fixing bugs with shorter development cycle and time to market pressure, a lot of effort and resources are put into the hardware verification these days. Almost 70% of the design time and engineering resources are put into design verification and that's why it is becoming such a bottle-neck for time to market in integrated circuit design [1].

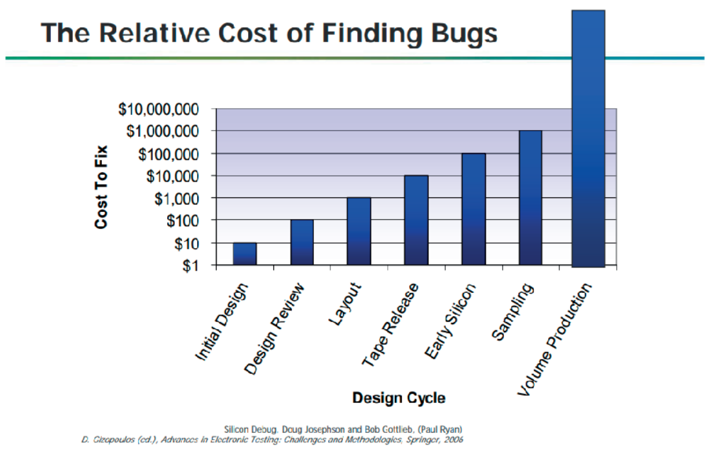


**Figure 1.1:** Bug cost [2]

Hardware verification is the process of proving the correctness of the system based on its specification and implementation. Basically most of the hardware verification done today can be classified into two broad categories - simulation and formal verification.

## 1.1 Simulation vs formal verification

Simulation verification is a technique where a test bench or test wrapper is created around the design. Then both direct written test or randomly generated test are used to stimulate the design. After running the test, the failures are debugged and fixed and then re-run again until the design is clean [3]. The advantages of simulation based verification is [4]:

- It is a proven methodology as it has been successfully used by verification engineers for so many years.
- It is scalable with bigger and complex designs.
- It helps to find most of the bugs, simple ones immediately and complex ones after some time.

Some disadvantages of simulation based verification are [4]:

- Creation of test bench is a lengthy process, generally takes months for the complex part or area of the design.
- It is very tough to create an exhaustive test bench covering all the possible input test cases.
- Some bugs are found very late in the development cycle and some are not found at all.

As shown in the figure 1.2, there are two tests (Test1 and Test2) which are applied on the design. After running them, they hit some states in the state space and depending on the presence of bugs on those path, they will either be able to find them or not. But at the same time a lot of states are still not covered with the generated input tests and because of that there is a possibility of missing some bugs in the design as shown in red.

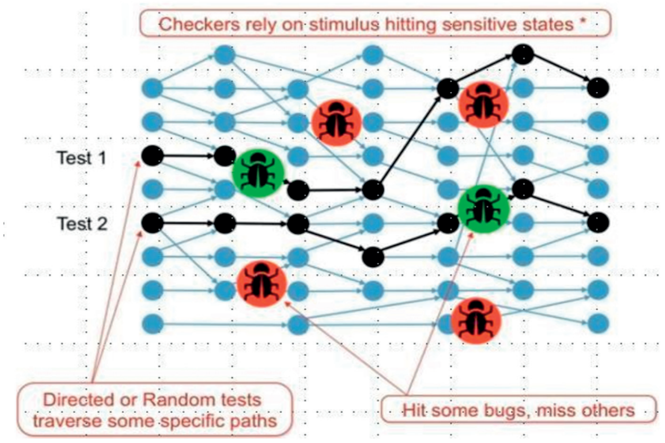

Checkers rely on stimulus hitting sensitive states *
Test 1
Test 2
Directed or Random tests traverse some specific paths
Hit some bugs, miss others

**Figure 1.2:** Simulation verification[5]

In Formal verification, the intended behavior of DUT is described with formal test properties written as SVA and the formal tool checks that model of DUT obeys this behavior in every possible way. This can be considered as doing all possible simulations and filter out traces which do not satisfy the proof. A misbehavior results in counter-example. Formal verification provides the following advantages over simulation verification:

- It is faster to implement.
- No input simulation test or test vectors are required.
- It is exhaustive meaning that a proven property covers the whole design space, thus, eliminating the chance to miss bugs.

But formal also has some limitations too as mentioned below :

- It is not scalable as simulation verification and more suited to smaller size designs.
- Formal verification is also not suitable to apply on all the designs such as arithmetic heavy designs or designs with high sequential depth.
- It requires a lot of effort to get the constraints right to eliminate spurious scenarios and focus on real ones.

As shown in figure 1.3, the properties are written on the output pins and the assumptions or design constraints are put onto the input pins of the design. The tool will drive the input pins values and in process cover more state space thus hitting more design bugs in comparison to simulation verification. Still there can be some missing corner cases due to over constraining of the design or missing properties or unreachable states.

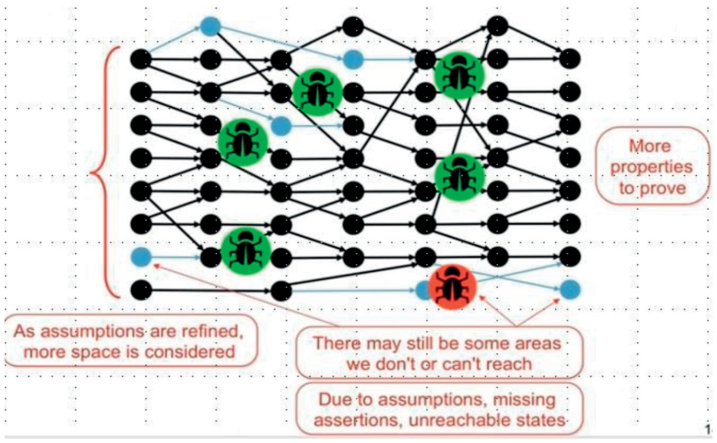

More properties to prove
As assumptions are refined, more space is considered
There may still be some areas we don't or can't reach
Due to assumptions, missing assertions, unreachable states
1

**Figure 1.3:** Formal verification[5]

In comparison, formal verification provides an advantage over simulation verification by covering more state space and try to reach and hit the corner case bugs that might or might not get missed by the simulation verification. Also, it is faster to implement compared to simulation environment set-up time and can be very useful to find the bugs early in a development cycle.

In this thesis work, the purpose would be to take the advantage of formal verification and apply it on the DUT. The DUT for this project is FSDC which is a module inside the Arm Mali GPU hardware IP. The DUT has already been extensively verified using simulation verification at Arm. The DUT consists mainly of control logic which suits formal verification. So the goals for this thesis work are :

- Application of formal verification on DUT.
- Try to find design bugs which might have been missed by the simulation verification already done on the DUT.

## 1.2 Structure of the thesis

The rest of the thesis is structured as following. In chapter 2, in detailed theory is explained on *Formal verification and Fragment Shader Descriptor Cache (FSDC)*. In chapter 3, the approach or method to apply the formal verification on DUT is being mentioned. In chapter 4, the results for this thesis works are discussed. And finally in chapter 5, the conclusion with possible future work is mentioned.

Chapter 2

# Theory

In this chapter, the basic theory related to theoretical concept of formal verification and FSDC is explained in detail.

## 2.1 Formal verification

By definition, formal verification is the use of tools that mathematically analyze the space of possible behaviors of a design, rather than computing results for particular values [6]. By saying that it means that formal verification will consider the entire space of possible input stimulus and will use clever mathematical techniques to cover all the possible behaviors. There are mainly two major formal verification techniques - first is *automated model checking* which uses mathematical models to verify that a specification meets it's implementation by proving a property over a vast state space. It is mostly automated without much human intervention and return one of the 3 results [7]:

- Property is proved or passed.
- Property is not proved and will give back a counter example for it.
- Property is undetermined which means that the state space is too big for the tool to give back a result with in reasonable amount of time.

The second one is *theorem provers*, which requires human expertise in addition to the automated techniques to prove the design correctness. These are more powerful than the automated model checking but require lot of expertise to apply it [7]. For this thesis work, automated model checking is used as formal verification technique and described below in more detail.

### 2.1.1 Automated model checking

Model checking problem is a problem for verifying that a formula f holds in a model M:

```
M |= f
```

where M is the design implementation while f are the properties based on the design specifications expressed in temporal logic like Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) [8]. As shown below in figure 2.1, the design implementation (RTL model) and its specifications are described in terms of intended design properties that is then being fed to the model checking tool which either prove its correctness or give a counter example if the property is not satisfied.

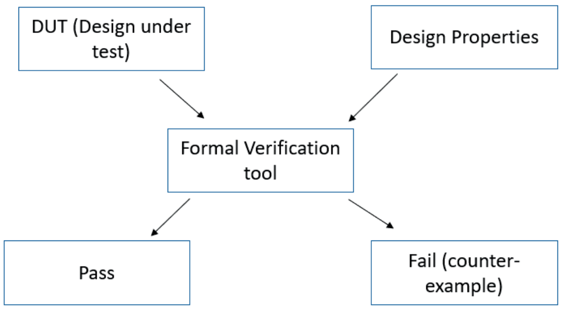


**Figure 2.1:** Model checking

### 2.1.2 Design assertions

Formal verification conforms that the design meets its specifications. In order to achieve this the first step needed is to find a way to express what it means for a design to be correct. This can be achieved by writing assertions using the System Verilog Assertions (SVA) language. The SVA can be thought of as several layers of increasing complexity as shown in figure below 2.2 [9].

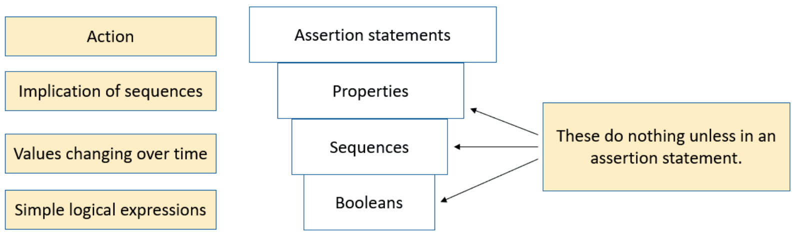

Action
Implication of sequences
Values changing over time
Simple logical expressions
Assertion statements
Properties
Sequences
Booleans
These do nothing unless in an assertion statement.

**Figure 2.2:** Layers of SVA assertion language [10]

#### Booleans

It can be a simple logical standard boolean expressions. It might be a single logic variable or a boolean formula such as *a & b*. The boolean expressions can be used in a sequence or property as shown in figure 2.2 and shall be evaluated over the sampled values of all the variables [11]. Whereas, the sampled values are the values of the variable at the end of each previous simulation time step.

## Sequences

These are the statements with boolean expressions in it that are happening over time. The simplest of sequences are linear in order meaning that they are just a list of finite boolean expressions that occur over linear order to increasing time. The increasing passage of time is defined with a clocking event. Sequences are composed by concatenation which specifies the time delay, using ## operator, from the end of first sequence to beginning of second sequence as shown below [9] [11].

```
a ##N b
It means that signal b shall be true on the Nth clock tick after
signal a was true.
```

## Properties

A property combines sequences with additional operators in order to capture the design behavior that is expected to be verified based on design specification. A property can be used as an assumption, assertion or coverage specification but does not produce result by itself. Shown below is an example of a named property reqgnt:

```
property reqgnt;
      req |-> s_eventually gnt;
endproperty
```

It states that if there is a request *req* which is an antecedent then eventually a grant must come for it *gnt* which is the consequent. The antecedent and consequent in the property are connected via the implication operators |-> or |=>. The first one (|->) is overlapping meaning that, if there is a match for antecedent then the end point for match is start point for the evaluation of consequent expression. While the second one (|=>) is non-overlapping meaning that, the start point for the evaluation of consequent is one tick after the match for antecedent [9] [11].

## Assertion statements

An assertion statement is used to validate the behavior of a system. As stated above, properties do not produce any result by itself. They need to be put into an assertion statement which uses one of the following keywords *assert, assume or cover*:

- Assert: to specify the property as an obligation for the design that is to be checked to verify that the property holds [11].

  ```
  assert property (req |-> gnt);
  ```

- Assume: to specify the property as input constraints on the environment. Formal tool use these input constraints in order to generate the input stimuli [11].

```
assume property (!req |-> !gnt);
```

- Cover: these are used to monitor the property evaluation for coverage [11]. They make sure that the intended behavior is happening at least once by finding a single trace for it.

  ```
  cover property (req |-> gnt);
  ```

The assertions in general can be of two types - immediate and concurrent:

- Immediate assertion statements are simple assertions which follow simulation event semantics and are executed in procedural blocks. There is no clocking or reset and does not support many advanced property operators. Due to this they cannot check conditions which have passage of time. The immediate assertions are defined using only *assert* keyword without using *property* keyword with it [9] [11]:

  ```
  immediate1: assert (!req && !gnt);
  ```

- Concurrent assertion statements follow the clock semantics and can describe the behavior that include passage of time. They also support advanced property statements about logical implementation that include time intervals. Concurrent assertion statements use both *assert and property* keyword in the statement as shown below [9] [11]:

  ```
  conc1: assert property (a ##2 req |=> gnt);
  ```

It is worth noting that immediate assertion statements are mostly used for simulation. In this thesis work, only concurrent assertion statements were written to verify the design behavior.

### 2.1.3 Model checking tool

As stated earlier, automated model checking is the preferred choice of formal verification technique used for this thesis work. At present Cadence Jaspergold is one of the leading tool in this area. It is designed by Jasper design automation which is a software company and provides formal functional verification software. Jasper design automation was founded in 1999 and was later acquired by Cadence in 2014. Cadence Jaspergold encapsulates the knowledge of formal expert in a tool. It combines advanced techniques such as tunneling and abstractions with powerful proof engines based on clever mathematical models which can help to prove a property smartly and efficiently. It has easy to use interface as shown in figure 2.3. Through its interface the formal verification process can be easily iterative (performing a verification, property changes and iterating on the verification). It also provides the support for liveness properties which are missing by some other tools [7] [12].

The Cadence Jaspergold applications consist of multiple Satisfiability (SAT) and Binary Decision Diagram (BDD) based proof engines with variation of these algorithms. Proof engines are basically mathematical models that are run on a

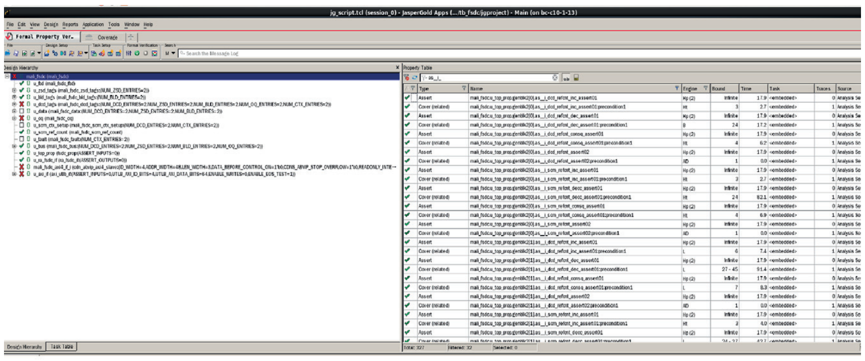


**Figure 2.3:** Interface of Cadence Jaspergold formal tool

property in order to prove or disprove it. Different proof engines can be used for different purposes for e.g. engine H is used for finding the counter-examples while its variant Ht is used for finding proof. Engine J is better suited for finding the traces and often doesn't find any proofs. The Cadence Jaspergold proofgrid application can be used to monitor various proof engines jobs running on cluster of hosts for all the design properties. As shown in figure 2.4, proof grid manager can help to perform below mentioned tasks [13]:

- Observe proof progress
- Mine data
- Control multiple proofs.
- Maximize engine and computing power.

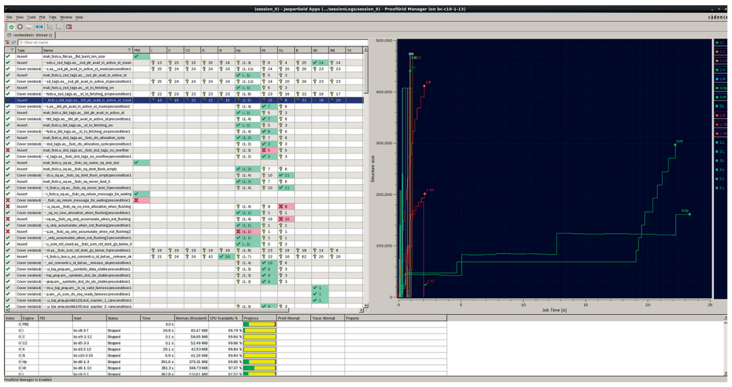


**Figure 2.4:** Proof grid manager

## 2.2 Fragment Shader Descriptor Cache (FSDC)

FSDC stands for Fragment shader descriptor cache. It is a module/block inside the Arm Mali GPU hardware IP from Arm. It is chosen as DUT for this thesis work as it is a new module and has control path logic which suits the formal verification. As shown in figure 2.5 below, as a module FSDC interacts with various other modules around it. Its main functionality is to fetch various programmable instruction set registers like ISR1 , ISR2, ISR3 and also act as the cache for them. These programmable instruction set registers can be programmed by the data received from software to front end which is resource allocator or rasterizer.

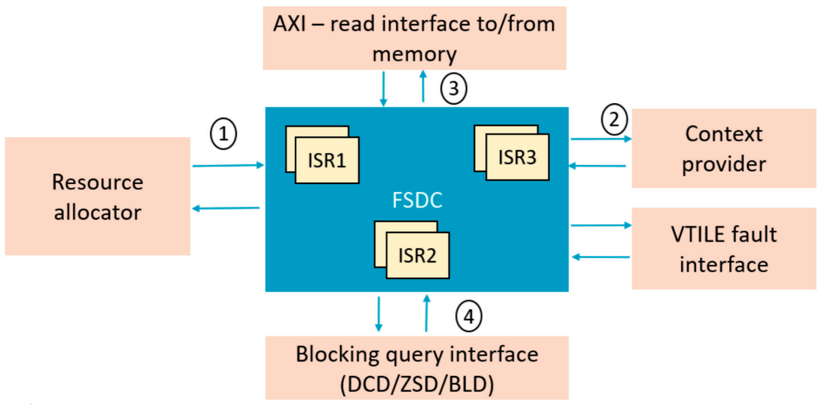


**Figure 2.5:** FSDC

The operation of FSDC can be broadly divided into 3 pipeline stages as shown in figure 2.6.

- Pre-fetch: The first stage will be the pre-fetch where the very first allocation request starts from resource allocator or rasterizer with a valid and data pointer request for ISR1 from resource allocator to FSDC. Then this data pointer which contains the address gets stored inside the cache of the FSDC module. A cache index gets allocated for this new request and then FSDC requests for a new context from Shader Context Manager (SCM). SCM as the name suggests is used to assign an available context id to the cache index. This request is then gets completed with the valid/ready handshake between the resource allocator and FSDC. While requesting for the context id if a fault is reported by the SCM then that fault will be reported by FSDC to vtile fault interface.

- Fetch: After successful completion of first stage operation, the pipeline will move into the second stage now which is fetching stage. Here the information for ISR1 will be fetched from the memory via AXI interface. If an AXI fault occurs here while reading the data then it will be reported again to the vtile fault interface by FSDC. The AXI interface only works in the read mode for the FSDC.

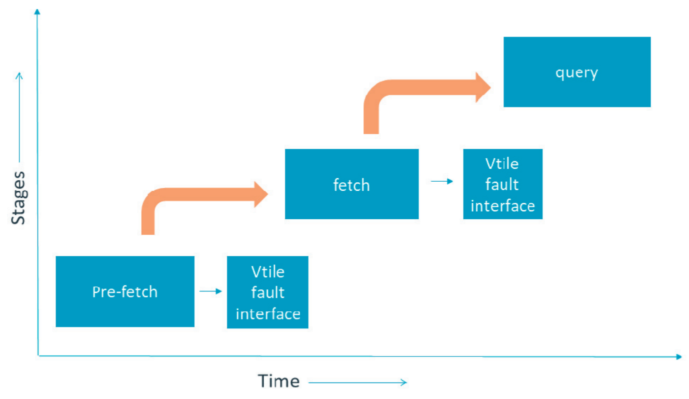


**Figure 2.6:** Time vs FSDC operation stages

- Query: Once the fetching will be done then the pipeline will move into the third and final stage which is blocking query stage. Now any other sub-module can query FSDC asking for the fetched data for ISR1 by raising the blocking query through blocking query interface. In the meantime, FSDC will receive various shader context increment/decrement signals. The FSDC keeps track of the number of dependencies on a context and when it reaches to 0 then a pdw deallocation signal is sent to retire the context. The lifetime of shader context id is bigger then the lifetime of cached ISR1 index attached to it as show in figure 2.7.

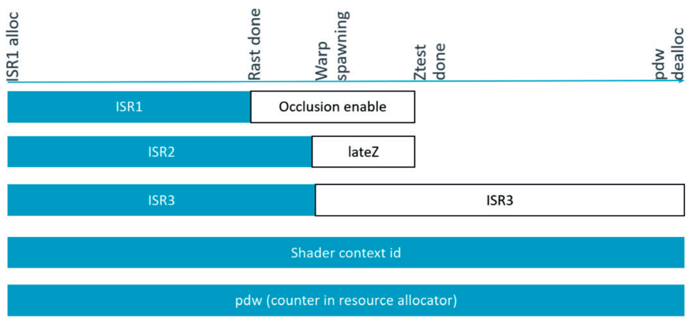


**Figure 2.7:** Lifetime of various registers and their index

FSDC also performs some more functionality that is independent from programmable instruction set register caching like occlusion query. Although it is not in scope of this thesis and thus, have not been discussed here.

Chapter 3

# Method

After some basic literature studying about the formal verification and design itself the actual task of implementing formal verification on DUT was started. Implementation of formal verification environment set-up and formal verification on DUT is explained in detail in this chapter. As stated before, formal verification tries to validate that *"does design conforms to it's specifications"*, the first step to perform is to understand the design specifications for FSDC. Also in parallel the formal verification test environment was been set-up in order to execute the formal verification test properties written for DUT.

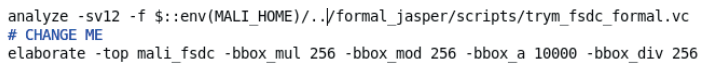
## 3.1 Formal verification test environment

Setting up the formal verification test environment is the first step into the thesis. It is better to make sure that formal verification test environment is up and running as soon as possible in order to run some basic checks or to try out some examples for more hand on practice. For setting up the formal verification test environment the first thing needed to be done is to create a Tool Control Language (TCL) script where various operations (like analyze and elaborate, define clocks and reset) required for a formal verification test bench can be mentioned.

### 3.1.1 Tcl script

TCL is a programming language that can be used as a command language for controlling the behavior of other tools [14]. Tcl scripts can help to automate the following behavior required for a formal verification test bench.

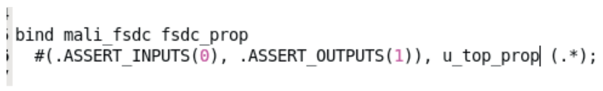
- Analyze and Elaborate the RTL design source code, bind and property file as shown in figure 3.1.



```
analyze -sv12 -f $::env(MALI_HOME)/../formal_jasper/scripts/trym_fsdc_formal.vc
# CHANGE ME
elaborate -top mali_fsdc -bbox_mul 256 -bbox_mod 256 -bbox_a 10000 -bbox_div 256
```

**Figure 3.1:** Analyze and elaborate command in the TCL script

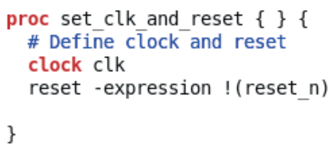
– Bind file is the file which use to bind the SVA property file with the RTL design model as shown in figure 3.2. Also, it was used to bind the Cadence AXI VIP module with design.



```
bind mali_fsdc fsdc_prop
  #(.ASSERT_INPUTS(0), .ASSERT_OUTPUTS(1)), u_top_prop (.*);
```

**Figure 3.2:** Bind file

- Define clock and reset for the design as shown in figure 3.3.

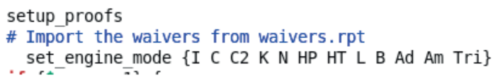


```
proc set_clk_and_reset { } {
  # Define clock and reset
  clock clk
  reset -expression !(reset_n)

}
```

**Figure 3.3:** Define global clock and reset

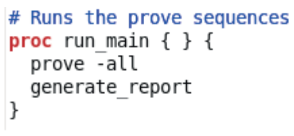
- Set-up the proof grid and proof engine for the tool as shown in figure 3.4.



```
setup_proofs
# Import the waivers from waivers.rpt
  set_engine_mode {I C C2 K N HP HT L B Ad Am Tri}
```

**Figure 3.4:** Define proof engines for the tool

- Add command to prove all the properties as shown in figure 3.5.



```
# Runs the prove sequences
proc run_main { } {
  prove -all
  generate_report
}
```

**Figure 3.5:** Command for proving all the properties

## 3.1.2 Running Cadence Jasper Gold

Once the TCL script is ready, the following command is used to run the tool. It includes the application name, memory for the tool. For this thesis work, the tool was run on the clustered network for better performance.

```
bsub -I -app FG -Jd misc -P PJ02477 -W 240:0 -c 240:0 -M 8192000 jg
fsdc.tcl &
```

## 3.2 Design specifications

Design specifications basically specify the functional behavior and requirements of the system. It is more abstract and really helpful in order to understand the intent of DUT and also helps in describing the functional behavior, timing behavior, performance characteristics, or internal structure of the design[15]. There is a design specification document available for the FSDC. First step to proceed in this thesis was to read the design specifications for the FSDC and try to understand the general behavior of the design. Reading and understanding the design specifications for FSDC helped with the following -

- It helped to identify basic assumptions or boundary constraints on the design.
- It helped to create a clear and concise formal verification strategy.
- It helped to understand the FSDC block functionality.
- It helped to identify interfaces to apply proper constraints and checks. For e.g. blocking query interface or Rasterizer FSDC interface.
- It helped to identify the elements in FSDC which elicit larger run times. For e.g. cache and counters inside the FSDC.

## 3.3 Formal verification plan

After getting some basic understanding on the FSDC block, the basic verification plan was made on how to implement formal verification on the FSDC module. The whole FSDC was first divided into 3 parts. First was to identify and implement the formal verification on ready valid interfaces where the required checks were missing, second was to create Input Output (IO) ports list for the FSDC to identify the number of inputs and outputs port that will be affecting the formal verification and the third was to identify functional checks needed to be verified with formal verification for this thesis work. The verification plan was very abstract in start and was fine tuned over the period of this thesis.

### 3.3.1 FSDC interfaces

The first task was to identify the ready valid interfaces for FSDC and make sure they are verified properly to ensure clean hardware IP with relevant assertions and assumptions. For FSDC as shown in figure 3.6, 4 interfaces were identified which required ready valid checks. Constraints have to be applied for the interface checks based on the design specifications.

- The 3 interfaces identified with missing ready valid checks for the FSDC module were - resource allocator/Rasterizer-FSDC interface, FSDC-SCM interface and FSDC-blocking query interface.

- The 4th interface identified was AXI-FSDC interface but for this interface, the Cadence standard AXI VIP module was already available and implemented. In FSDC, AXI is only used for the read operations. Thus, this VIP module was only implemented for AXI read operation mode and made sure that it is working as expected with already implemented set of assertions and assumptions.

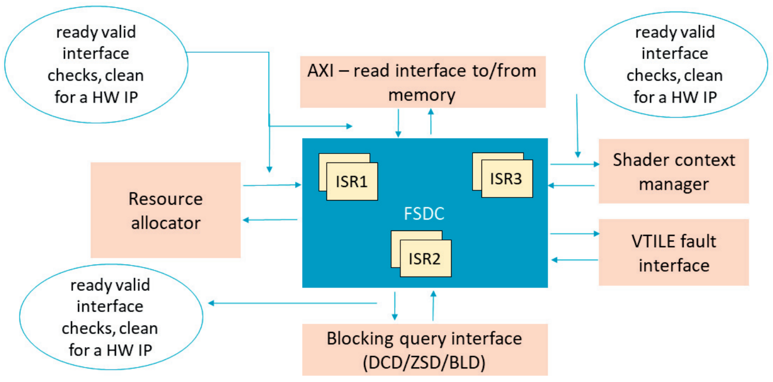

ready valid interface checks, clean for a HW IP
AXI – read interface to/from memory
ready valid interface checks, clean for a HW IP
Resource allocator
ISR1
ISR3
FSDC
ISR2
Shader context manager
VTILE fault interface
ready valid interface checks, clean for a HW IP
Blocking query interface (DCD/ZSD/BLD)

**Figure 3.6:** Ready valid interface for FSDC

### 3.3.2 FSDC IO port list

As part of the second task identified, a list of IO port list was made which will affect the formal verification on FSDC module. In total, there were 64 input ports and 50 output ports that were identified which will be affecting the formal verification in this thesis work.

### 3.3.3 Functional checks

The last task was to identify the FSDC functionality which will be targeted using the formal verification in scope of this thesis work. In the start, following discussions with thesis supervisor and module designer at Arm, various functionality within the FSDC module was planned to target as part of this thesis but over the time with various challenges it was trimmed down to the manageable level. Below mentioned is the list of the functionality which was finally covered through formal verification as part of this thesis.

1. ISR1/descriptor functional checks
    - Internal cache behavior
        - Cache hit scenarios.
        - Cache miss scenarios.

- New load request check (only in case of CM)
  - New SCM context request.
  - New SCM context configuration request.
  - New SCM context ready request.
  - New AXI request.
  - Blocking query response (for both CH and CM)

2. Proving design counter for both cache and context index id.
3. FSDC FSM checks.
4. Reset checks.
5. Fault transmission behavior
   - Fault transmission through AXI
     - for ISR1 caching.

## 3.4 Formal verification implementation

After a basic plan was ready with the areas to target with formal verification for the FSDC module, it's time to start implementing these task by actually writing the assertions and assumptions for the DUT. In the start, the work started with actually writing a very basic property in order to see if it can be run successfully as part of the first trial process. It took a lot of wiggling through out the thesis work to get things right but it will be described later in the report.

As shown in figure 3.7, the first basic check written was pretty straight forward where a design specification is converted into an assertion or property - *"if there is a rasterizer valid request with the data pointer to FSDC then eventually a ready should be asserted back by the FSDC."*.

```
assert property (ra_dcd_valid |-> s_eventually ra_dcd_ready);
```

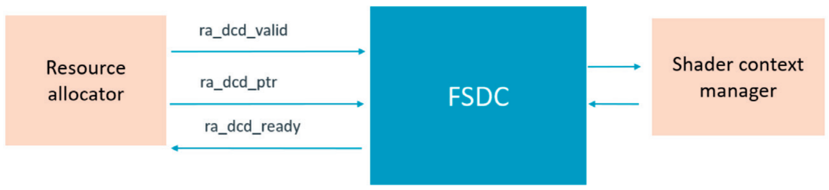


**Figure 3.7:** Example - Rasterizer to FSDC valid and ready

As this property was running with almost no constraints and abstractions applied on the design. The tool was not able to either prove or disprove it and this property got finished as undetermined which can be argued as a result but not in this case. The possible reason that can be concluded for this property to end up as undetermined would be that without any design constraints, a large amount of logic was fed to the tool which makes the tool to traverse through huge state space of the system thus increasing the complexity for the tool in order to either prove or disprove formal verification test property. This challenge of complexity is also know in formal verification as *state space explosion problem*. It can be entirely or partially solved by applying right design constraints and abstraction techniques. It will be covered later in this report. For now, we will try to verify the identified ready valid interface checks to make sure that interfaces are clean. After that we will try to start verifying more complex functional checks.

### 3.4.1 FSDC interface checks

Revisiting the figure 3.6, the first priority now is to formally verify the identified ready valid interfaces in FSDC module. Verifying interfaces helps to model the exact behavior of these interfaces between FSDC and the other modules.

- Rasterizer-FSDC interface - This interface has ready valid handshake between the two modules. According to the specification for the ready valid handshake, the valid signals and payload must be stable until the ready is asserted to complete the handshake. This specification was then converted into the design constraint or assumption as shown below.

  ```
  assume property (ra_dcd_valid && !ra_dcd_ready |=> $stable
  ({ra_dcd_valid, ra_dcd_ptr}));
  ```

- FSDC-Blocking query interface - This interface also has a ready valid handshake with the same design specification as mentioned above. The ready valid protocol here is also constrained so the valid and payload stay stable until ready is asserted as shown below.

  ```
  assume property (vl_dcd_valid && !vl_dcd_ready |=> $stable
  ({vl_dcd_valid, vl_dcd_index}));
  ```

  Also for this interface, there is another constraint applied based on design specification - *if the data in ISR1 is not yet allocated to the cache then no blocking query should be raised for it.*

  ```
  assume property (dcd_refcnt[i] == 0 |-> !(vl_dcd_valid &&
  vl_dcd_index == i));
  ```

- FSDC-SCM interface - This interface has a little different ready valid protocol in the way that valid is actually output from FSDC to SCM and while ready signal with the payload is sent back from SCM as an acknowledgement and there is no need to apply the similar kind of constraint on it as done for other two interfaces. Rather than it has generic specification stating that the ready signal should come at some point so the tool won't wait for it to be asserted indefinitely. The applied assumption on this interface is shown below.

```
assume property (!fsdc_scm_ctx_req_ready |-> s_eventually
fsdc_scm_ctx_req_ready);
```

- FSDC-AXI interface - This is the fourth interface identified but there is a standard Cadence AXI VIP module already available which was integrated directly in the formal test environment as mentioned in section 3.1.1. This VIP module was for complete AXI verification although for FSDC, AXI is used only in read mode and not in write, thus, the VIP was configured only for read mode according to the design specification.

With all the constraints for the interfaces now in place according to the design specification, now some properties or assertions can be written in order to formally prove or verify them. It will make sure that the interfaces are bug free and working as expected. Although the assertions for these interfaces were implemented as part of the functional checks and will be explained later in functional checks section.

## 3.5 Functional checks

After setting the constraints for all the interfaces, next task in the verification plan mentioned in section 3.3.3 is to start implementing the formal verification on various identified functional checks. Considering back the example shown in figure 3.7, the tool was not able to either prove or disprove the property and it was finished as undetermined. As stated before, formal verification runs exhaustively which leads to the state space explosion problem. In order to solve the state space problem various techniques such as case splitting, abstractions, proper design constraints can be employed on the design which can reduce the amount of logic being fed to the tool for each property. All these techniques mentioned were wiggled continuously throughout the thesis work. These techniques are described below for this thesis work and are important to mention first before diving into the discussion of assertions.

### 3.5.1 Abstractions

This is a useful technique in order to reduce the complexity of the design, thus, reducing the amount of logic or state space that the tool has to search. For example, as there are lots of corner case issues present in an architecture when a buffer is full, then reducing the size of buffer will lead to explore the design in the architecture when buffer is full [16]. As one of the main intentions of this thesis is to hit some corner cases in the DUT. After reading the design specifications, it was observed that in the design there are 12 cache lines and design counters are of 16 bit. Memory and counters are known to explode the state space exponentially. Initially the checks were running with the original cache and counter sizes but later on to reduce the complexity, the cache size was reduced to 2 and counter to 2 bits.

Both the cache size and counters were parameterized on the top level and was expected to trickle down the hierarchy. After changing the parameters for both cache size and counters, 2 design bugs were identified and fixed which are explained in detail in section 4.1.

### 3.5.2 Assumptions

Assumptions are used to constraint the behavior of the design in order reduce the complexity and state space of the design. As mentioned before, formal verification suffers with challenge of handling the *spurious scenarios* as lot of time can be wasted upon debugging just the spurious cases. Right constraints on the design ensures that the spurious scenarios are filtered out and only the interesting scenarios are focused upon. It also helps with bug hunting as it will help to hit the specific behavior in the design [16]. The assumption applied at the start of the thesis kept on wiggling through out the thesis work to make sure that they conform with design specification. For FSDC, some of the assumptions applied on the design include:

- SCM faults were blocked.
- The cache invalidate signal is set to 0.
- Functionality related to occlusion is blocked as it is independent of the targeted functional checks.
- The AXI faults were blocked although they were relaxed for functional checks related to AXI faults.
- There are lots of increment and decrement signals which affect various counters within the design. For this thesis work, only one increment and one decrement signal is kept while setting others to 0.
- Some assumptions were applied on the design programmable instruction register ISR1 and context reference counters like -
  - Both the design counters don't go negative.
  - Both the design counters won't overflow.
- If a context id is allocated then the same id won't be allocated again.
- if a context id is not allocated then it won't be deallocated.
- On FSDC-SCM interface, if the context configuration happens then the context ready valid signal must be asserted for same index from SCM.

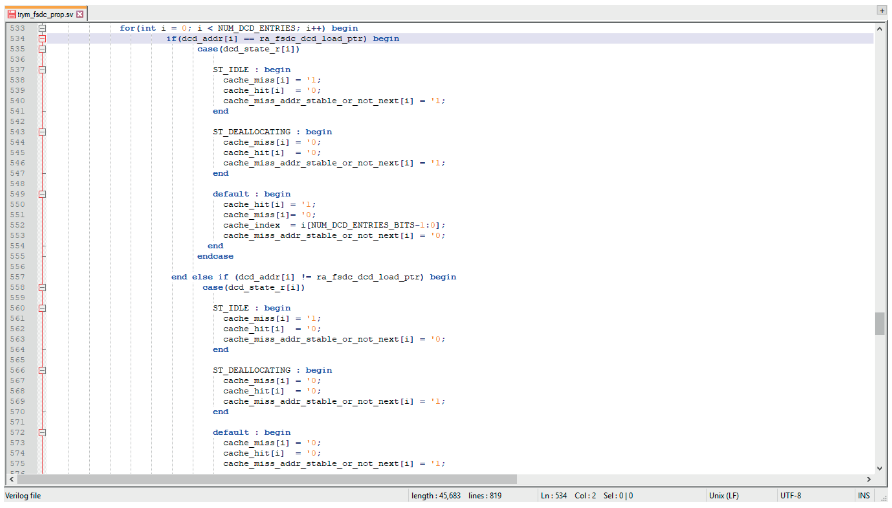
### 3.5.3 Auxiliary code

Auxiliary code is the helper RTL code that can help to reduce the complexity for writing down the assertions and assumptions for some of the functional checks. As mentioned before, the abstractions and assumptions (constraints) already defined before were actually wiggled and identified through out the thesis work. In order to write down a lot of these assumptions and assertions for the design, a reference model or auxiliary code was created so it can aid in writing down these properties in a simple way which can reduce lot of complexity and also help with the debugging in case of a failure. The auxiliary code for this thesis work is written in SystemVerilog. As mentioned in section 3.3.3, part of functional check is to verify the internal behavior of cache by validating the CH and CM scenarios. A specific model was created with the auxiliary code which identifies these when the cache hit and miss is taking place inside the cache and then these cache hit and miss variables can be used in the assumptions and assertions directly to verify the behavior of the cache. In below table 3.1, conditions are specified depending on the FSM that when these cache hit or miss should take place based on the design specifications.

**Table 3.1:** CH and CM scenario

| Address match | | | Address don't match | | |
|---|---|---|---|---|---|
| IDLE | DEALLOC | REST | IDLE | DEALLOC | REST |
| CM = 1<br>CH = 0 | CM = 0<br>CH = 0 | CM = 0<br>CH = 1 | CM = 1<br>CH = 0 | CM = 0<br>CH = 0 | CM = 1<br>CH = 0 |

As shown in the table 3.1, based on incoming data pointer from rasterizer if it matches with the cache data or not and also on the given state of that cache present inside the FSDC, the cache hit or miss variable is set with 1 or 0. Then based on values assigned in these variable the assertions can be used to verify the behavior of the cache. A small code snippet of the auxiliary code to mimic this behavior is shown below in the figure 3.8.


```
for(int i = 0; i < NUM_DCD_ENTRIES; i++) begin
        if(dcd_addr[i] == ra_fsdc_dcd_load_ptr) begin
            case(dcd_state_r[i])

              ST_IDLE : begin
                cache_miss[i] = '1;
                cache_hit[i]  = '0;
                cache_miss_addr_stable_or_not_next[i] = '1;
              end

              ST_DEALLOCATING : begin
                cache_miss[i] = '0;
                cache_hit[i]  = '0;
                cache_miss_addr_stable_or_not_next[i] = '1;
              end

              default : begin
                cache_hit[i] = '1;
                cache_miss[i]= '0;
                cache_index  = i[NUM_DCD_ENTRIES_BITS-1:0];
                cache_miss_addr_stable_or_not_next[i] = '0;
             end
            endcase

         end else if (dcd_addr[i] != ra_fsdc_dcd_load_ptr) begin
             case(dcd_state_r[i])

              ST_IDLE : begin
                cache_miss[i] = '1;
                cache_hit[i]  = '0;
                cache_miss_addr_stable_or_not_next[i] = '0;
              end

              ST_DEALLOCATING : begin
                cache_miss[i] = '0;
                cache_hit[i]  = '0;
                cache_miss_addr_stable_or_not_next[i] = '1;
              end

              default : begin
                cache_miss[i] = '0;
                cache_hit[i]  = '0;
                cache_miss_addr_stable_or_not_next[i] = '1;
```

**Figure 3.8:** Auxiliary code snippet

This is just one example of benefit of auxiliary code that helped to reduce down the complexity of the functional task in order to write down simple and easy to understand assertions. The actual assertions written with the help of these auxiliary code variables is explained more in the assertions section 3.6.

## 3.6 Assertions

After implementing the abstractions, assumptions and auxiliary code for the FSDC module, its time to define the assertions on the functional checks identified before in section 3.3.3. As explained before, assertions are the properties that will actually try to prove if the design conforms to its specifications. And any failure in an assertion will lead to a counter example which is the waveform capturing the failure and can be debug for the root cause analysis of the failure. The first set of assertions were applied to verify the cache behavior and are explained below.

### 3.6.1 Cache behavior

As explained before, FSDC fetches data for various programmable instruction set registers and then also act as a cache for them. The first data allocation request starts when resource allocator or rasterizer sends the *ra_ dcd_ valid and ra_ dcd_ ptr* signal to the FSDC. This input data load request will then get stored inside the cache present in the FSDC module. The subsequent request to FSDC will get stored if the cache is empty and it is a CM or will get skipped if it is a CH. As this is the first task that is done by FSDC, thus, assertions were written in order to verify the cache functional behavior of FSDC module.

For verifying it, as stated before in the section 3.5.1 and 3.5.2 the number of cache lines were reduced from 12 to 2 and the cache invalidate signal was set to 0. Also the auxiliary code mentioned in section 3.5.3 was developed. The checks were mainly divided into cache miss and cache hit.

- Cache Miss (CM) - When a cache miss happens then the new data pointer which contains the address of the first descriptor is stored in a cache line and a request is made for its context id by FSDC to SCM as shown in the figure 3.9.

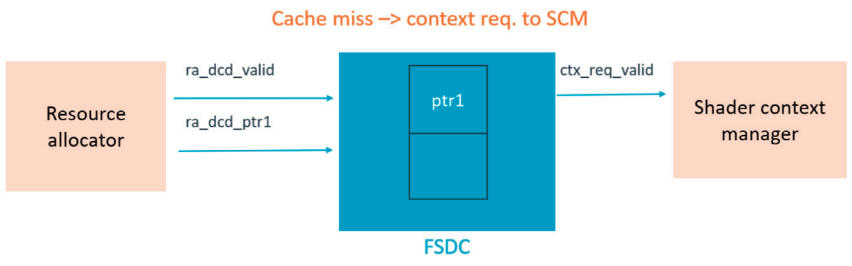


**Figure 3.9:** Cache miss scenario

The example code snippet of the assertions to verify the two functional checks when a cache miss happens is shown below:

The first and second assertions verify that if a cache miss happens then the cache line won't be stable as it will be updated with the new data.

```
assert property(!cache_full_flag && cache_miss_flag_r |=>
!$stable(dcd_addr));
```

```
assert property( cache_full_flag  && cache_miss_flag_r |=>
!$stable(dcd_addr));
```

While in the third and fourth assertions it is verified that if a cache miss happens then FSDC must assert a context request valid signal to SCM requesting the context id for the newly stored data.

```
assert property(!cache_full_flag && cache_miss_flag_r |->
ctx_req_valid );

assert property( cache_full_flag && cache_miss_flag_r |->
s_eventually ctx_req_valid );
```

A control point - *cache_full_flag* was created to further reduce down the complexity of the logic.

- Cache Hit (CH) - When a cache hit happens then the new data pointer is already present in one of the cache lines and no request for the context id will be raised by FSDC to SCM as shown in figure 3.10.

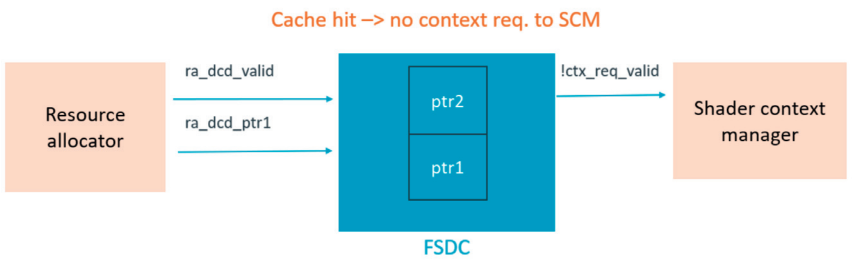


**Figure 3.10:** Cache hit scenario

The example code snippet of the assertions to verify the two functional checks when a cache hit happens are shown below:

In the first and second assertions, if there is a cache hit then after 2 clock cycle delay the cache line must be stable and won't be updated with the incoming data request. The timing delay of 2 clock cycles was expected as stated in the design specifications.

```
assert property(!cache_full_flag && cache_hit |=> ##1
$stable(cache_hit_var_r));

assert property( cache_full_flag && cache_hit |=> ##1
$stable(cache_hit_var_r));
```

In third and fourth assertions, if there is a cache hit then FSDC won't be requesting for the new context id from SCM.

```
assert property(!cache_full_flag && cache_hit ##[0:$]
scm_hsk |=> !ctx_req_valid);

assert property( cache_full_flag && cache_hit |=>
s_eventually !ctx_req_valid);
```

Here again, the control point mentioned above is used to reduce down the complexity of logic. In table 3.2, total number of assertions written for the verifying the cache behavior is mentioned with their status:

**Table 3.2:** Total no. of assertions for verifying the cache behavior with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 10 | 10 | 0 | 0 |

### 3.6.2 SCM new load request check

As shown in figure 3.11, after the data is cached then FSDC will request for the context id. Once the context id is provided to FSDC by SCM, FSDC will start reading the cached data for ISR1 and for sub-registers ISR2, ISR3 from memory. After reading the data it will then configures the context. After the configuration is done then SCM will assert a ready valid signal FSDC for the same context id. The context id stays active until SCM will receive a pdw context de-allocate signal from the FSDC.

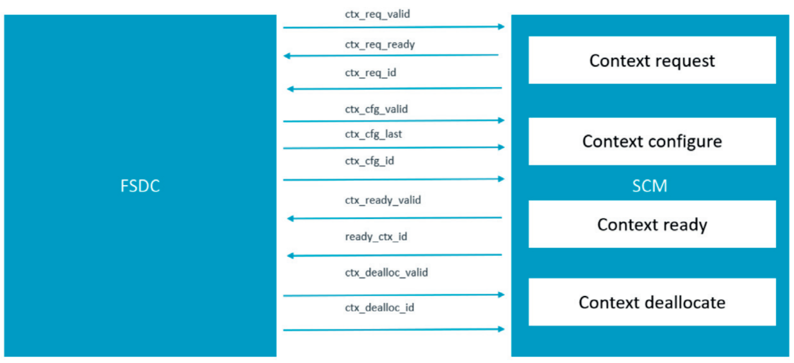


**Figure 3.11:** FSDC-SCM interface verification

- Context request: After the data is cached, FSDC will request the SCM for the context id by raising the *ctx_ req_ valid* signal. Then, the SCM will acknowledge the context request with an available context id and ready signal. Once this SCM handshake happens after then only FSDC will respond with *ra_ dcd_ ready* signal to resource allocator or rasterizer. This design specification is verified by an assertion mentioned below which states that after

SCM handshake, FSDC will eventually assert the ready back to rasterizer. This assertion will also verify the missing rasterizer-FSDC interface check identified in section 3.4.1. Also, 1 design bug was discovered while trying to prove this property due to an incorrect mask signal inside the design which was missed during the simulation verification. This bug is explained in detail in section 4.1.

```
assert property(scm_hsk |=> s_eventually ra_dcd_ready);
```

- Context configure: Once the context id was requested, the FSDC will start reading the data from memory for ISR1 data cached in the cache. After all the data is fetched without any fault then FSDC sends a valid and last signal with the related context id to SCM in order to configure the context id. The below mentioned assertion will verify this design specification for context configuration where *i* is a variable in a *for loop* to cover both the cache index. The code snippet for the assertion is mentioned below:

```
assert property (ctx_req_valid && ctx_req_ready && ctx_req_id
== i |-> s_eventually (ctx_cfg_valid && ctx_cfg_last &&
ctx_cfg_id == i));
```

- Context ready valid: Now the SCM will send back a ready valid signal with the context id to FSDC notifying that the configuration has been completed. As shown in figure 3.11, these are input signals for FSDC, thus, a constraint was set in place to ensure that the respective behavior will work as expected. The code snippet for the assumption is shown below:

```
assume property (ctx_cfg_valid && ctx_cfg_last && ctx_cfg_id
== i |-> s_eventually (ctx_ready_valid && ctx_ready_ctx_id
== i));
```

- Context de-allocation: After the context configuration, context stays active until a pdw de-allocation signal is sent from FSDC to SCM with the context id. Also, as highlighted in section 3.5.2, an assumption was written to verify that if a context id is allocated then it won't be allocated again. A context flag is created through auxiliary code which will be set when the SCM context request handshake is done for a context id while unset to 0 when pdw de-allocation signal occur for the same context id. Thus, when the context flag is 1, then design is constrained as shown in assumption to make sure context request ready won't happen for the same context id. While the assertion was written as shown below to verify that when context flag is 0 (not allocated), then the de-allocation request for it won't happen.

```
assume property ( ctx_flag_r[i] == 1 |-> !(ctx_req_id == i &&
ctx_req_ready));

assert property ( ctx_flag_r[i] == 0 |-> !(ctx_dealloc_id== i
&& ctx_dealloc_valid));
```

The above properties will also complete the verification of FSDC-SCM interface. In below table 3.5, highlights the number of properties that are written in total for verification:

**Table 3.3:** Total no. of assertions for verifying the SCM context load with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 7 | 4 | 0 | 3 |

As highlighted in table 3.5, 3 properties are undetermined and tool was not able to prove or fail it. But saying that, undetermined can be considered as a result in itself. Although it is not a full proof but still it can be concluded that till that clock cycle (depth) tool is not able to find any issue which is a bounded proof instead [17]. In table below 3.4, the undetermined properties are highlighted. First one is for context request for which tool went till the depth of 68 clock cycles and second for context configure for which tool tried to disprove it till the depth of 33 clock cycles. Hence, it can be assumed safely that their are no bugs till this depth while the undetermined can be result of some missing constraints. In this case, It was very hard to convert this bounded proof into a full proof but saying that it can still be considered as similar to running exponential number simulations on the design when compared to simulation verification. These bounded proofs should be analyzed further as part of future work to try to convert it into full proofs.

**Table 3.4:** Undetermined properties for SCM context load with their depth

| Assertion | Undetermined |
|---|---|
| Context request | clock cycle = 68 |
| Context configure for both cache lines | clock cycle = 33 |

For these undetermined properties, the cover property was also written in order to perform the sanity checks to ensure that the functional behavior is working as expected. The code snippet for the cover property is given below with their status result in the table .

```
cover property(!cache_full_flag && scm_hsk |=>  ra_dcd_ready);

cover property (ctx_req_valid && ctx_req_ready && ctx_req_id == i |->
s_eventually (ctx_cfg_valid && ctx_cfg_last && ctx_cfg_ctx_id == i));
```

### 3.6.3 Blocking query checks

As shown in figure 2.6, after the fetching stage comes the query stage where any other sub-module can query the FSDC for cached ISR1 data. While the query request is being performed, FSDC will block the cached ISR1 data for that cache

**Table 3.5:** Total no. of covers for undetermined property with their status

| Number of covers | Proved | Fail | Undetermined |
|---|---|---|---|
| 3 | 3 | 0 | 0 |

line. As shown in figure 3.12, a handshake happens between the FSDC and the blocking query interface for an active cached index. Thus, an assertion was written to verify that if a blocking valid is asserted by the blocking query interface then FSDC will eventually assert a ready for an active cached index.

```
assert property (vl_dcd_valid && dcd_refcnt[vl_dcd_index] > 0 |->
s_eventually (vl_dcd_ready ));
```

Another assertion was written to verify that if the cached index is not active based on its counter value then no ready signal will be asserted for it by FSDC. The code snippet for the assertion is shown below:

```
assert property (dcd_refcnt[vl_dcd_index] == 0 && !vl_dcd_fault |->
!vl_dcd_ready);
```

In table below 3.6, shows the status for blocking query interface assertions:

**Table 3.6:** Total no. of assertions for verifying the blocking query with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 2 | 1 | 0 | 1 |

Out of the two assertions, the first one was undetermined and the tool was never able to prove it. As mentioned before, undetermined assertions or properties can happen due to missing design constraints as state space for the tool might be too big to prove it. Still it can be considered as a bounded proof as the tool was not able to find any counter example till the depth of 36 clock cycle as shown in table 3.7.

**Table 3.7:** Undetermined properties for blocking query with their depth

| Assertion | Undetermined |
|---|---|
| blocking query | clock cycle = 36 |

But in order to see if the intended functional check is happening at least, a cover was written to perform the sanity check within the 50 clock cycles. The cover got completed successfully with in the bound of 35 clock cycles. The code snippet for the cover is highlighted below with its status specified in the table 3.8.

```
cover property (vl_dcd_valid && dcd_refcnt[vl_dcd_index] > 0 |->
##[0:50] (vl_dcd_ready));
```

**Table 3.8:** Total no. of covers for undetermined property with their status

| Number of covers | Proved | Fail | Undetermined |
|---|---|---|---|
| 2 | 2 | 0 | 0 |

### 3.6.4 Design counters

There are two design counters in FSDC module. As stated in section 3.5.1, they were reduced from 16 bit to 2 bit to reduce down the state space for the tool to cover. One of the counters is for the ISR1 cache index while the other one is for its context. As shown in figure 2.7, the lifetime of counter for cache index is smaller than the one for the cache context. This is one of the design specification and an assumption was written for it. Also as stated in section 3.5.2, few more assumptions were applied on these counters to make sure they don't go negative and also they don't overflow. After the correct assumptions were put in place the assertions were written in order to prove both the counters. The code snippet for the first assertion is shown below. It checks that if the SCM reference counter has an increment then in next clock cycle the current value of counter must be equal with the past (one clock cycle in the past) value of the counter with an increment or decrement operation on it. Similar can be said for the decrement operation.

```
inc || dec |=> counter == $past(counter) +inc - dec

assert property ((scm_refcount_inc[i]) |=> (scm_refcount[i] == $past
(scm_refcount[i])+$past(scm_refcount_inc[i])-$past(scm_refcount_dec[i]
)));
```

In the next assertion, it was verified that if the counter is not stable in current cycle than in the past there must be an increment or decrement operation has occurred. The code snippet for the following assertion is highlighted below:

```
!$stable(counter) |-> $past(inc || dec)

assert property (!$stable(scm_refcount[i]) |-> $past
(scm_refcount_inc[i]) || $past(scm_refcount_dec[i]));
```

Third assertion was written to verify the following property for a counter - A == A' |=> A == A'. It means that in every clock cycle the current value of counter is equal to the past value of counter with any increment or decrement event happening on it. The code snippet for the following assertion is highlighted below:

```
assert property (scm_refcount[i] == ($past(scm_refcount[i]) + $past
(scm_refcount_inc[i]) - $past(scm_refcount_dec[i])) |=> scm_refcount[i]
== ($past(scm_refcount[i]) + $past(scm_refcount_inc[i]) - $past
(scm_refcount_dec[i])));
```

The above code snippet only highlights one counter but similar assertion were written for the other counter too in order to prove it. In below table 3.9, total number of assertions written for both counters and cache lines are highlighted with their status:

**Table 3.9:** Total no. of assertions for verifying the design counters with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 12 | 12 | 0 | 0 |

### 3.6.5 Verify FSDC FSM

After verifying the counters, few assertions and covers were written in order to prove the FSDC FSM. There are 7 states in the FSDC FSM which also define the sequential depth of the design. Covers were written as a sanity check in order to verify that for each cache line at least the design will reach every state of the state machine once. As shown in the code snippet below, for each cache line defined by $i$, the cache state must reach all the states of state machine for e.g. ST_IDLE or ST_REQUEST0 and so on. While as shown in code snippet below, assertions were written to prove that for each cache line the state transitions for state machine is happening as expected for e.g. from ST_IDLE to ST_REQUEST0. The example of some covers and assertions code snippet is shown below:

```
cover property ( dcd_state_r[i] == ST_IDLE);
cover property ( dcd_state_r[i] == ST_REQUEST0);

assert property ($past(!reset_n) |-> s_eventually dcd_state_r[i]
== ST_IDLE);

assert property ( dcd_state_r[i] == ST_IDLE && dcd_state_en[i] |=>
dcd_state_r[i] == ST_REQUEST0);
```

The below table 3.10 shows total number of assertions written for the proving the FSDC finite state machine with their status.

### 3.6.6 Reset checks

A few assertions were written in order to verify that the design gets its initial values like counters must start from 0 after the reset has happened. The FSDC has active low reset in the design. As it can be seen in the below assertion code snippet, if a reset have occurred in the past then in the current clock cycle various

**Table 3.10:** Total no. of assertions for verifying the FSM with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 16 | 16 | 0 | 0 |
| Number of covers | Proved | Fail | Undetermined |
| 14 | 14 | 0 | 0 |

flags like cache miss and hit or the design counters like SCM reference counters must start from their initial assigned value.

```
assert property ($past(!reset_n) |-> ((cache_miss_flag_r == ’0) &&
(cache_hit_var_r == ’0) ));

assert property ($past(!reset_n) |-> (dcd_refcnt == ’0) &&
(scm_refcount[0] == ’0) && (scm_refcount[1] == ’0));
```

The below table 3.11 shows total number of assertions written for verifying the reset checks with their status.

**Table 3.11:** Total no. of assertions for verifying the reset checks with their status

| Number of assertions | Proved | Fail | Undetermined |
|---|---|---|---|
| 2 | 2 | 0 | 0 |

### 3.6.7 AXI fault checks

So far the faults have been constrained but in order to write some functional checks around AXI faults the constraints will be relaxed in order to let the AXI faults happen. After the rasterizer-FSDC handshake the FSDC will start fetching the data from memory for the cached data through AXI interface. While fetching this data a fault can occur and then that fault will be reported by FSDC to vtile fault interface as shown in figure below 3.12. Also note that, AXI interface is only used in read mode by FSDC. To write assertions for AXI faults here we will be using the symbolic variables. Symbolic variables are free variables which can hold the data of a logic variable depending on its entry and exit point.

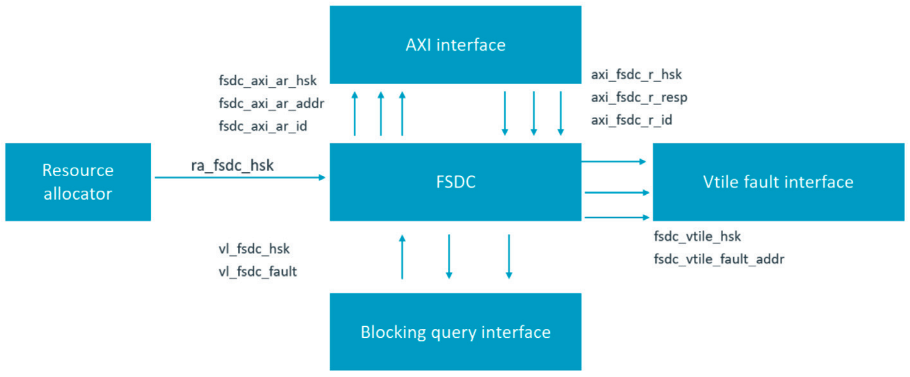


**Figure 3.12:** AXI fault verification

As shown in figure 3.13, the entry point and an exit point are selected which will define the lifetime of a symbolic variable. For this particular case, the entry point is chosen when the rasterizer hand shake happens with the FSDC and at same moment the value of data pointer with cache and context index for the cache data is stored in the symbolic variable. While the exit is chosen when the FSDC-Blocking query handshake happens for the same cache index for which rasterizer FSDC handshake happened. In order to make sure that symbolic variables hold their value through out the lifetime, a below mentioned assumption was written which make sure the symbolic variables stay stable as long as *symb_in* is high.

```
assume property ( symb_in |-> $stable(symb_ra_ptr));
assume property ( symb_in |-> $stable(symb_dcd_idx));
assume property ( symb_in |-> $stable(symb_ctx_idx));
```

After the symbolics, now the assertions will be written with the help of symbolic variables to check if any fault occurs on AXI or not. The AXI-FSDC interface can also be checked when no faults will occur with the help of the control point *axi_faults* as highlighted below in assertion code snippet:

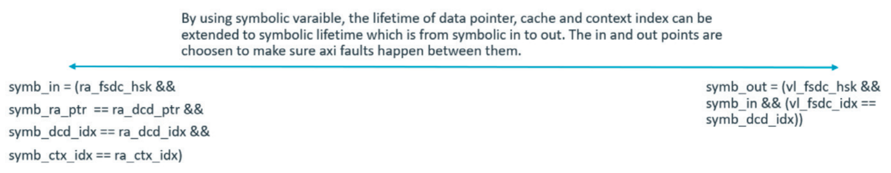

By using symbolic varaible, the lifetime of data pointer, cache and context index can be extended to symbolic lifetime which is from symbolic in to out. The in and out points are choosen to make sure axi faults happen between them.
symb_in = (ra_fsdc_hsk &&
symb_ra_ptr == ra_dcd_ptr &&
symb_dcd_idx == ra_dcd_idx &&
symb_ctx_idx == ra_ctx_idx)
symb_out = (vl_fsdc_hsk &&
symb_in && (vl_fsdc_idx ==
symb_dcd_idx))

**Figure 3.13:** Symbolic variable lifetime

```
assert property
($past(symb_in) && $past(fsdc_axi_ar_hsk) && $past(fsdc_axi_araddr ==
symb_ra_ptr)), fsdc_arid_var = fsdc_axi_ar_id) |-> s_eventually
(axi_fsdc_r_hsk && (axi_fsdc_r_id == fsdc_arid_var) && (axi_faults ||
!axi_faults)));
```

Here, if a *symb_in* has been set with all the other symbolic variables then the FSDC will start the fetching of data from AXI which will raise an FSDC-AXI address read handshake for the same cached data. Once this pre-condition is met as specified on the left hand side of the property, then AXI will start reading the data and once finished it will generate a read last signal with matching transaction id (ar_id == r_id) but the most important is the response signal which tells if the transaction completed is clean or with fault.An AXI control point will monitor the transaction response and thus, can be used to verify the same in the above specified assertion. If a fault occurs than FSDC will propagate this fault to vtile fault interface and blocking query interface. The assertions to verify fault prorogation are shown in below code snippet:

```
assert property (symb_in && axi_faults && axi_fsdc_r_hsk |->
s_eventually (vtile_hsk && (fsdc_vtile_fault_addr == symb_ra_ptr) &&
(fsdc_vtile_fault_data_type == DATA_TYPE_XYZ)));

assert property (symb_in && vtile_fsdc_hsk && (fsdc_vtile_fault_addr
==  symb_ra_ptr) |-> s_eventually  (vl_fsdc_hsk && vl_fsdc_fault &&
(vl_fsdc_index == symb_dcd_idx)));
```

In the first assertion, if a fault occurs at AXI then it will get propagated to vtile fault interface for the same cached data pointer address. While in the second assertion same fault will get propagated to blocking query interface for the same cached index.

Chapter 4

# Results

In this chapter, the final results for this thesis work are presented. As discussed before, the first aim was to successfully try to implement formal verification as an alternate mode of verification on a complete module inside the Arm Mali GPU hardware IP. After that the second aim was to try and hit some corner cases that can lead to functional bugs that might have been missed by the simulation verification already done on the FSDC block.

## 4.1 Formal verification application on DUT

As highlighted in section 3.4, formal verification has been successfully applied on DUT. During the implementation, the formal verification bring up time for formal verification test environment and some functional checks were observed and compared against the simulation verification. The comparison is highlighted in below table 4.1,

**Table 4.1:** Formal Vs. Simulation bring up time comparison result

| Functional Check | Simulation verification | Formal verification |
|---|---|---|
| Environment bring up time | 3-4 weeks (UVM) | 1 week |
| DCD load request & response check | 3 weeks | 2 weeks |
| SCM request & response check | 2 weeks | <1 week |
| Fault checks | 1.5 week | 2 days |

Based on the results shown in table 4.1, the following points were concluded.

- The formal verification test environment takes less time to set-up and get started when compared to the Universal Verification Methodology (UVM) simulation verification environment set-up.

- It takes less time to write functional checks in terms of SVA. Thus, it can help to run the basic functional checks early in design phase and find early design bugs when UVM test environment is not ready.
- Another conclusion that was made was formal verification can be really helpful for the design engineer with deep understanding of the DUT in comparison to a verification engineer.

Note that in the table 4.1, it only captures the time taken to set-up the formal verification test environment and converting the functional specification into functional system verilog assertions. But it does not include the time spent on actually writing the auxiliary code, debugging and proving the properties as verified or not.

Also the total number of system verilog assertions written as part of formal verification implementation for covering all the functional checks are highlighted in below table 4.2,

**Table 4.2:** Total no. of SVA written in this thesis work

| Assertions | Assumptions | Covers |
|---|---|---|
| 51 | 64 | 19 |

Also in section 3.3.2, an IO list was made to capture all the inputs and outputs that were affecting the formal verification for this thesis work. Below table 4.3 highlights the number of IO that were covered by applied assumptions and assertions on the design in this work.

**Table 4.3:** IO Planned vs Covered

| Input | | Output | |
|---|---|---|---|
| Planned | Covered | Planned | Covered |
| 64 | 41 | 50 | 19 |

As it can be seen from the table 4.3, there are some IO that are not covered. It is because a lot of FSDC functional checks were trimmed down considering the time and complexity for this thesis. But as part of future work, formal checks can be applied on them to prove them formally.

Another observation was made related to the abstraction of cache line and counters as discussed in section 3.5.1. As stated earlier, to reduce the complexity the cache lines were reduced from 12 to 2 while counters were reduced from 16 bit to 2 bit. This led the tool to complete the assertions fairly faster as compared to original specifications. A comparison was made for the following Cache Hit (CH) property that *if a cache hit happens then there won't be any scm request happening for it* as described in section 3.6.1. The comparison was made with help of the plot (structure size vs time) created by proof grid manager inside the Cadence jasper gold. Here, structure size is the amount of logic that tool traversed over time in order to either prove or disprove a property. With 12 cache lines the structure size is roughly $4 * 10^6$ which is the amount of logic being fed to the tool. As it is a too big of state space for the tool, roughly after 33,000 seconds this property completed as undetermined.

Now after applying the abstraction techniques it can be observed that the same property got proved by the tool in 55 seconds with structure size 550,000. Thus, with the abstraction technique the amount of logic was almost reduced by 7 times while preserving the functional behavior of the design. This observation was a good measure to conclude that how abstraction techniques can help to formally prove or disprove the properties.

## 4.2 Corner case bugs

The second aim of the thesis work was to try and hit some corner cases in order to find bugs that might be missed by simulation verification. During the duration of this thesis work, 3 bugs were identified that lead to the RTL changes in the design code.

- *Parameterization inconsistency* - As mentioned before in section 3.5.1, in order to solve the state space problem some design reduction was done by reducing the number of cache lines from 12 to 2 and counters from 16 bit to 2 bit through cache size and counter parameters in the design itself. While performing the reduction some synthesis failures for the design were observed. After successful investigation, it was observed that parametrization of these parameters were not consistent and some of them are hard coded instead of being parameterized due to which on changing the main parameters for cache line and counter size synthesis failed. After successfully parameterizing, the design synthesis passed. As design reduction was never a requirement for the simulation verification this inconsistency in parameters couldn't be identified before.
- *Missing if condition* - After the cache lines and counter size reduction as part of abstraction techniques in section 3.5.1, another issues was observed during code synthesis. For addressing the 12 cache lines 4 bit address was required but with 4 bit address a total of 16 lines can be addressed (12 cache lines and 4 extra address space), thus, 4 extra address lines were left and in the code they were padded with 0. But on reducing the cache lines to 2, a missing if condition was observed in the code for handling the 0 bit padding for extra cache lines. Hence, it lead to a RTL code fix and the relevant if condition was added. Again, the cache reduction was never a requirement for the simulation verification, this corner case has never been observed or caught before.
- *Incorrect mask signal* - This bug was found during the debugging of a counter example for a property failure in section 3.6.2 (context request). This counter example was generated for a very specific set of inputs which were not tested in the simulation verification before. In simulation verification every cached index has either 0 to 0 or 1 to 1 mapping with every context id but formal verification tried to also map 0 cache index to 1 context id or 1 cache index to 0 context id. This lead to a corner case where a signal was not behaving properly. While performing the debugging, it was discovered that there is a masked signal in the code which is not working correctly. After further analysis, the masked signal was fixed as part of RTL code fix and then property proved afterwards. This bug was a good example of how formal verification is a powerful tool in finding the corner cases that might get missed by the simulation verification.

Chapter 5

# Conclusion and Future work

In this chapter a summarized version of the whole thesis project is presented as part of the final conclusion. Also, some comments are mentioned that can be considered as possible future work based on the work done in this thesis project.

## 5.1 Conclusions

There were 2 main objectives for this master thesis project titled "*Exploration of formal verification in GPU hardware IP*". The DUT has already been extensively verified by simulation verification:

- Explore the application of formal verification on FSDC as an alternate verification methodology on a complete module inside the Mali GPU hardware IP.
- Try to hit some corner cases in order to find bugs which might be missed by the simulation verification.

The FSDC was chosen as a module for this thesis work because it was a new module and mostly has control logic which suits formal verification. The implementation for this thesis project started by first setting up the formal test verification environment. A basic test property was written covering the following design specification *"if there is a rasterizer valid request with the data pointer to FSDC then eventually a ready should be asserted back by the FSDC."*. But this lead to the state space exploration problem which indeed is a challenge for formal verification. A formal verification test plan was created to handle the state space exploration problem and divide the checks into :

- FSDC interface checks.
- Functional checks.

Also abstractions were used to further reduce down the verification complexity to tackle down the state space explosion problem. The abstractions that were applied on the design are :

- Cache lines were reduced from 12 to 2.
- Counter size was also reduced down from 16 to 2 bits.

The formal test verification properties were written using the SVA. For writing down the formal verification test properties which are easy to understand and implement some auxiliary code was written. Also various design constraints were applied using SVA in order to filter out the spurious scenarios which is another challenge faced during the implementation of formal verification in this thesis project. All the implementation was done in line with both the objectives of applying formal verification on DUT and try to find the bugs in the design.

The results obtained after this thesis work is very interesting given that the thesis worker had no prior knowledge about formal verification before the start of the thesis work. A lot of challenges were faced during the thesis work especially due to state space explosion problem and filtering out the spurious scenarios. After tackling all these challenges to various extent the formal verification was applied on the entire module inside the Arm Mali GPU hardware IP. It was observed that the time taken to setup the formal test verification environment was less compared to set up the UVM for simulation environment. Also during the duration of the thesis work various interesting corner cases were explored. The spurious scenarios were filtered out by putting the right constraints on the design. Finally, 3 bugs in the design were discovered which led to the RTL changes. As the DUT was already verified by the simulation verification the identification of these bugs highlighted the advantages of formal verification.

The results shows that formal verification has some advantages over simulation verification. At the same time, formal verification has its own challenges such as it is more suitable to be applied on designs with more control logic. Formal verification is not very suitable for others designs such as arithmetic heavy designs or designs with big sequential depths where simulation verification is still more preferable compared to formal verification due to state space explosion challenges. Hence, after observing the result obtained in this thesis work and challenges of formal verification , it can be finally concluded that formal verification can complement simulation verification well for designs which are suitable for formal verification.

## 5.2 Future work

The work done in this thesis project was to explore the application of formal verification on the FSDC module. Considering in mind that the thesis started with no prior formal verification knowledge it took some time to understand it and also the FSDC design. Thus, previously the plan was to complete the functional checks for FSDC but after re-evaluation the scope was changed and only one type of instruction set register ISR1 was focused with AXI faults. The work done in this thesis gives a good head start on formal verification but it is still far from the point of saying that complete FSDC is verified formally. Thus, below mentioned areas can be continued to be work on as part of future work but was not included in the scope of this thesis work.

- Right now functional checks are focused only one ISR1 descriptor. In future, more functional checks can be added for other sub instruction set register descriptors such as ISR2 and ISR3 in order to completely verify the FSDC block functional checks through formal verification.

- There is a cache block present inside the FSDC. Checks related to Cache Hit (CH) and Cache Miss (CM) are already verified in this thesis work. But the cache itself gets updated by Least Recent Used (LRU) algorithm which is not included in scope of this thesis. But as a future work, it would be very interesting to verify the algorithm for the cache to formally check the cache integrity.
- Another area for future work can be to optimize the helper or auxiliary code and make it more simpler which can further help in writing more simple properties.
- As mentioned before, not all the properties are still proved and there are some undetermined properties especially for blocking query handshake. This might be because of missing design constraints or too big state space for the tool. Further investigation into all the undetermined properties can lead to the root cause analysis and might help to prove undetermined properties.
- Coverage analysis of the design for all the properties was out of scope for this thesis work. But as future work it would be really interesting to run the various coverage analysis through Jasper gold in-built coverage application. It will give a good measure of how much the design is covered by written properties and also help to identify the holes in the design which are yet not covered or missed because of some constraints. It can possibly further lead to hit some interesting corner cases and maybe unearth some design bugs.

# References

[1] V. Bertacco. Ph.D. Dissertation, Stanford University, August 2003. [Online]. Available: `http://web.eecs.umich.edu/~valeria/research/thesis/thesis2.pdf`

[2] H. Foster.Verification is a Problem, but is Debug the Root Cause?, Mentor Graphics, MTV 2010. [Online]. Available: `http://www.vennsa.com/mtv/Harry_Foster.pdf`

[3] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, section 1.3.1.

[4] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. *Formal Verification: An essential toolkit for modern VLSI design*, section Foreword.

[5] S. Rahman, Verification in hardware (ASIC/FPGA), Ericsson Lund 2016. [Online]. Available: `https://www.eit.lth.se/fileadmin/eit/courses/eitf35/2016/LTH_Course_2016_Verification_in_ASIC_FPGA.pdf`, page 39,40.

[6] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. *Formal Verification: An essential toolkit for modern VLSI design*, page 2.

[7] R.C. Armstrong, R.J. Punnoose, M.H. Wong, J.R. Mayo. *Survey of existing tools for formal verification*, Sandia report, 2014.

[8] S. Berezin. *Model Checking and Theorem Proving: a Unified Framework*, Carnegie Mellon University, 2002.

[9] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. *Formal Verification: An essential toolkit for modern VLSI design*, page 49-54.

[10] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. *Formal Verification: An essential toolkit for modern VLSI design*, fig. 3.2.

[11] *IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language*, IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, 2012.

[12] V. Singhal, H.D. Foster. Jasper design automation(Cadence now), 2005.[Online]. Available: `https://people.eecs.berkeley.edu/~alanmi/courses/2005_290A/lectures/vigyan.pdf`

[13] *JasperGold Engine Selection Guide*, Cadence, September 2017.

[14] What is Tcl ?, Doulos. [Online]. Available: `https://www.doulos.com/knowhow/tcltk/tutorial/`

[15] E.M. Clarke and J.M. Wing. *Formal Methods: State of the Art and Future Directions*, Carnegie Mellon University, 1996.

[16] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. *Formal Verification: An essential toolkit for modern VLSI design*, page 162-173.

[17] J. Bromley, J. Sprott. Verilab, 2016. [Online]. Available: `https://www.verilab.com/files/dvcon_eu_2016_fv_tutorial.pdf`

[18] A. Sanghavi. Jasper design automation, *What is formal verification ?*, EE-TAsia, 2010.

[19] C.E. Cummings. *System Verilog Assertions Design Tricks and SVA Bind Files*, SNUG, San Jose, 2009.

[20] R. Stuber. *Formal Verification: Not Just for Control Paths*, Mentor, June 2017.

Printed by Tryckeriet i E-huset, Lund 2019

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-732
http://www.eit.lth.se