DEBUG CONTROLLER VERIFICATION OFA GRAPHICS CHIP

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE AWARD OF THE DEGREE

OF

MASTERS IN TECHNOLOGY

IN

VLSI DESIGN AND EMBEDDED SYSTEMS

Submitted by

SHREYA GUPTA (2K20-VLS-18)

Under the Supervision of

MR. VARUN SANGWAN

Assistant Professor Department of Electronics and Communication

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Delhi-110042

MAY, 2022

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, Shreya Gupta Roll No. 2K20-VLS-18 student of MTech VLSI & Embedded Systems, hereby declare that the project Dissertation titled " DEBUG CONTROLLER VERIFICATION OF GRAPHICS CHIP " which is submitted by me to Mr. Varun Sangwan, Department of Electronics and Communication, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi **SHREYA GUPTA**

Date: 31-05-2022

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CERTIFICATE

I hereby certify that the Project Dissertation titled " Debug Controller Verification of a Graphics Chip" which is submitted by Shreya Gupta Roll No 2K20-VLS-18, Department of Electronics and Communication, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of the Project work carried out by the student under my supervision. To the best of my knowledge, this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi **MR. VARUN SANGWAN**

Date: 31-05-2022 **SUPERVISOR**

ACKNOWLEDGEMENT

It gives us great pleasure to present the Dissertation of the Major Project undertaken during MTech fourth semester. I owe a special debt of gratitude to Mr Varun Sangwan, Department of Electronics and Communication Engineering, Delhi Technological University (Formerly Delhi College of Engineering), for his constant support and guidance throughout this work. His sincerity, thoroughness and perseverance have been a continuous source of inspiration. It is only his conscious efforts that our endeavours have seen the light of the day.

We also take the opportunity to acknowledge the contribution of Prof. N. S. Raghava Head of the Department of Electronics and Communication Engineering, Delhi Technological University (Formerly Delhi College of Engineering), for his full support and assistance during the development of the work.

SHREYA GUPTA (2K20-VLS-18)

Date: 31-05-2022

ABSTRACT

Verification is considered a critical stage of any chip development cycle. This step ensures that the design meets the system requirements and specifications. At this stage, test cases are developed, and the invention's functionality is checked. But it is supposed that Verification takes almost 60% of the total chip development chip cycle. It makes it a critical stage in any chip flow since any bug found post routing is a bit tough to remove, and also, post-fabrication, it is challenging to correct the design.

The project focus on improving the efficiency of verification cycle. It can be achieved by introducing a Debug Checker in the Testbench itself, which reduces verification time many folds.

The project targets segregating the potential bug region once any mismatch or error is found in Testbench.

CONTENTS

Candidate's Declaration i

Certificate ii

Acknowledgement iii

Abstract iv

Contents v

List of Figures vii

CHAPTER 1 INTRODUCTION 1

1.1 Literature Survey 1

1.2 Thesis Organisation 3

CHAPTER 2 GRAPHICS OUTLINE 4

2.1 Graphics - Introduction 4

2.2 Need for Parallelism 7

2.3 Why GPUs 8

2.4 GPU Organisation – How does it works 8

CHAPTER 3 VERIFICAION OF A GRAPHICS CHIP 10

3.1 What is Verification 10

3.2 Verification Problems 10

3.3 Verification Complexity 11

3.4 When is Verification Completed? 11

3.5 Verification Strategies 12

3.6 Verification Environment 12

3.7 Testbench Design and Verification Tests 13

3.8 Verification Plan and Benefits 14

3.9 Tools for Verification 14

3.10 Power Aware Verification 15

3.11 Verification Scope 15

CHAPTER 4 IMPORTANCE OF DEBUG 16

4.1 Need for Debug 16

4.2 How Debug is performed? 17

CHAPTER 5 DEBUG CHECKER 19

5.1 Checker - Explanation 19

5.2 Pseudo Code 24

CHAPTER 6 OUTPUTS 28

CHAPTER 7 CONCLUSION 34

REFERENCES 35

LIST OF FIGURES

Fig. 2.1	Gimples of a 3D Graphics Model	4
Fig. 2.2	Dilution of 3D Model – Different stages shown	4
Fig. 2.3	Graphics Pipeline	5
Fig. 2.4	Arrangement in Shader for 3D Modelling	6
Fig. 2.5	Configurating 3D Models – The smallest element is triangle	6
Fig. 2.6	CPU V/S GPU (Difference in number of cores)	7
Fig. 3.1	Chip Development Flow	10
Fig. 3.2	Maximum time of a Chip Design is taken in Verification	11
Fig. 3.3	Verification Strategies	12
Fig. 3.4	Verification Environment	13
Fig. 3.5	Stages in which Verification Grow	14
Fig. 3.6	Dynamic Simulation	15
Fig. 4.1	Debug takes maximum of Verification timespan	16
Fig. 5.1	DBGC Interface	19
Fig. 5.2	Data coming from HLM (At the start of HLM)	20
Fig. 5.3	Data coming from HLM (At the mid of HLM)	20
Fig. 5.4	Data obtained by analysing every packet and getting MISMATCH correspondingly [Packet 22]	21
Fig. 5.5	Data obtained by analysing every packet and getting MISMATCH correspondingly [Packet 110]	22
Fig. 5.6	Data obtained by running the checker in that code and segregating the bits for MISMATCH terms [Packet 22]	22

Fig. 5.7 Data obtained by running the checker in that code and segregating the bits for MISMATCH terms [Packet 24] 23

Fig. 6.1 Showing data coming from HLM module (At start of HLM module) 28

Fig. 6.2 Showing data coming from HLM module (At mid of HLM module) 28

Fig. 6.3 Showing data of testbench describing the positions of MISMATCH in the design [Packet 22] 29

Fig. 6.4 Showing data of testbench describing the positions of MISMATCH in the design [Packet 110] 29

Fig. 6.5 Showing data of testbench describing the positions of MISMATCH in the design [Packet 22] and with segregations in the packet as well 31

Fig. 6.6 Showing data of testbench describing the positions of MISMATCH in the design [Packet 24] and with segregations in the packet as well 31

Fig. 6.7 Showing data of testbench describing the positions of MISMATCH in the design [Packet 30] and with segregations in the packet as well 32

Fig. 6.8 Showing data of testbench describing the positions of MISMATCH in the design [Packet 32] and with segregations in the packet as well 32

Fig. 6.9 Showing data of testbench describing the positions of MISMATCH in the design [Packet110] and with segregations in the packet as

well 33

Fig. 6.10 Showing data of testbench describing the positions of MISMATCH in the design [Packet 134] and with segregations in the packet as well 33

CHAPTER 1

INTRODUCTION

1.1 LITERATURE SURVEY

The ever-increasing design complexity of Integrated Circuits (ICs) resulted in challenging aspects of functional/logic Verification, in terms of verification platform complexity, achieving verification goals like code/functional coverage and unbounded verification time/efforts for any given digital design.[1]

The scale and complexity of integrated circuit designs are ever-expanding, making the verification process increasingly difficult and progressively time-consuming. [2]

Debug Checker is a project which aims to reduce the load and time consumed in verification step. Traditional simulation-based bus protocol monitors can check whether bus signals obey bus protocol, but they often lack efficient debugging mechanisms. [11]. It could be understood as a function which is called whenever MISMATCH is observed. The debug controller gives visibility to the internal state of the GPU. The debug controller interacts with most GPU building blocks, gathers internal state information, and presents it to the outside world.

It provides visibility to internal microarchitecture. It has the ability to trace states through the AMBA Trace Bus. In modern-day blocks, mostly communication protocols (from AMBA) families are used, such as advanced peripheral bus (APB), advanced high-performance bus (AHB) and advanced extensible inter-face (AXI) to make synchronised communication.[7] It basically provides the ability to detect a certain pattern matches in the internal state. It also shares the ability to trace out all debug signals from all the blocks. Debug is performed for two GPU blocks synchronously without any additional latency. It gives the freedom to trace down the pipeline flow events passing through the GPU Blocks. The debug bus data is used for multiple debug operations such a s debug, trace, counting etc. The debug bus data path is segmented in to four regions. The debug data bus is designed to match the pre-defined pattern. The debug bus data provides general-purpose input and output ports. It saturates the Debug Bus on Backpressure from ATB. The basics functionality also provides the ability to be trace out all debug signals of only one selected block in the queue. The skill to debug two GPU blocks synchronously without additional latency for that particular configuration. It has many

other functionalities like debugging when context switching is happening—managing the trace data generation. So, debugging on its own is a task. A task that needs so much functionality and works to be taken care of. It is significant for verification flow and needs to be done to result in the least possible potential bug.

Debug controller or DBGC is a block specially provided in GPU for having a database of bugs or fatal errors observed at the post-silicon stage of the chip. If this needs to be explained in the manner, Verification is itself an essential part of any chip flow. Still, in case we got an error or bug at post-silicon, it could lead to a significant loss to person-hours and capital invested in that project. Hardware Design verification detects design errors affecting functional and extra–operating. [3]

So, to keep that project's credibility intact, it is imperative to have a parallel code. The code that runs along with the RTL contains the list of all the sequences and potential failures. In the case of a particular scenario, if a bug is found even in the post-silicon stage. It can be debugged.

So, Debug Controller has a very critical role in the chip flow. Now, the question is if Debug Controller itself is free from bugs? The database we are considering GOLDEN must be free from all bugs and fatal errors. So this results in the need for Debugging of Debug Controller itself. So, specific sequences run on that block with different functionalities for debugging. Every sequence has its own defined set of intentions, and they are run accordingly. The failing sequence results in the failing of that particular functionality that needs to debug.
So, this project is about reducing the man-hours in debugging and making it more and more time efficient and powerful.

So, the logger or the checker aims to compare the data from the Perf Read Event block, which is coming from the Cluster Block and The HLM Data, which is the golden response. It checks the data from both blocks. It generates a MISMATCH whenever different output or data is found. Now, the checker is also designed in a way that it helps in debugging. The debugging process includes, whenever there is a MISMATCH for that particular output, the checker or logger segregates the data. In that way, the potential bug is found.

So, the project will be about explaining the GPU Block. The pipeline, the stages, and sub-blocks. Then, the Verification, the need and requirement because of entering into sub deep

micron technology. In the past few decades, there has been a significant improvement in electronics. The size of transistors reduces results in reducing the size of the product and introducing new technology in VLSI.

1.2 THESIS ORGANISATION

This thesis is divided into seven chapters. A Debug Controller is defined in Chapter 1 as an introduction. The functionality and complete literature review are included in Chapter 1. Chapter 2 focuses on Basic of Graphics. Basic introduction, the need of parallelism and why GPU comes in the scene. The later part of this chapter describes its organisation as well. Chapter 3 explains the significance of verification. Verification is a very crucial part of any chip development flow. It presents the complete flow, including, problems, complexity, strategies, environment, tests, tools, benefits scope etc. Chapter 4 explains the need for debugging. As already mentioned, verification takes the maximum time of chip development, Debug is a part of verification which consumes maximum human hours. Most of the time verification is consumed in debugging. So, it presents the need for debugging and how debugging is done. Chapter 5 shows the Debug Checker. This chapter gives an overview of the working of the checker and the pseudo-code. Chapter 6 shows the outputs and explains the completes the whole flow. The conclusion and primary application where this checker is used are presented in Chapter 7.

CHAPTER 2

GRAPHICS OUTLINE

2.1 GRAPHICS INTRODUCTION

Computer Graphics – The Spectrum of such applications is vast. Challenging to list all applications as everything we see around us involving computers contains some computer graphics applications.

It is seen that at the present stage, the integrated discipline "Computer graphics" can absorb the theoretical material of the discipline "Descriptive Geometry" and the functional area of the domain "Engineering Graphics".[5]

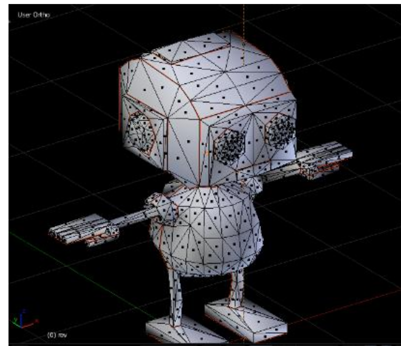

Fig. 2.1 Gimples of a 3D Graphics Model

Apart from desktop/Laptop applications, we traditionally refer to them as computers. Computer Graphics is used in mobile phones, information kiosks at popular spots such as airports, ATMs, large displays at open-air music concerts, air traffic control panels, the latest movies to hit the halls, etc.Computer graphics have become synonymous with the computer for a non-specialist in this digital age.

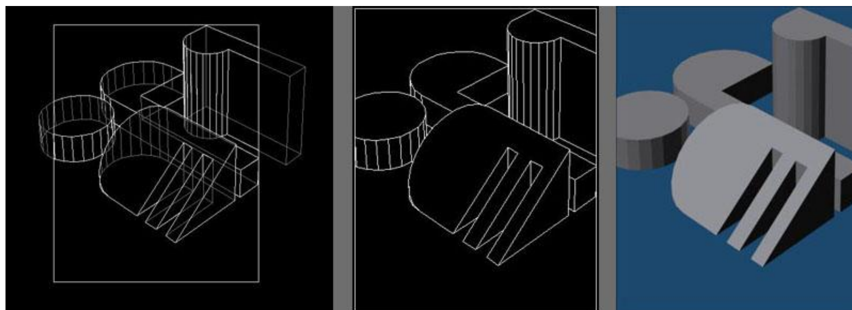

Fig. 2.2 Dilution of 3D Model – Different stages shown

What is expected in all these applications?

- Instances of images displayed on the computer screen (note that the text characters are also images).
- Images are constructed with objects, which are geometric shapes (characters and icons) with colours assigned to them.

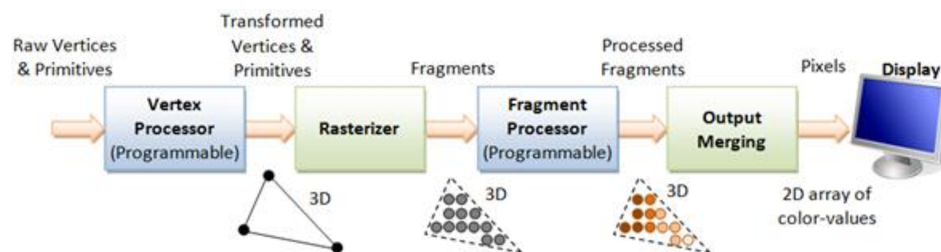

Fig. 2.3 Graphics Pipeline

- Instances of images displayed on the computer screen (note that the text characters are also images).
- Images are constructed with objects, which are geometric shapes (characters and icons) with colours assigned to them.

- When we create/edit/view a document, we deal with letters, numbers, punctuations, and symbols.
- Each object is rendered on the screen with a different style and size.
- In the case of drawing (e.g., using MS Paint or MS Word), we deal with basic shapes such as circles, rectangles, curves, etc.
- In the case of animation videos or computer games, we deal with virtual characters – The characters may or may not be human-like.
- The images or parts can be manipulated (interacted with) by a user – User input devices such as a mouse, keyboard, joystick, etc.
- We know computers understand only binary language – the language of 0s and 1s. Letters, numbers, symbols, or characters are not strings of 0s and 1s.

By this, we face two central questions:

- How can we represent such objects in a language understood by computers so that the computer can process those?

- How can we map from the computer's language to something we perceive (with physical properties such as shape, size, and colour)?

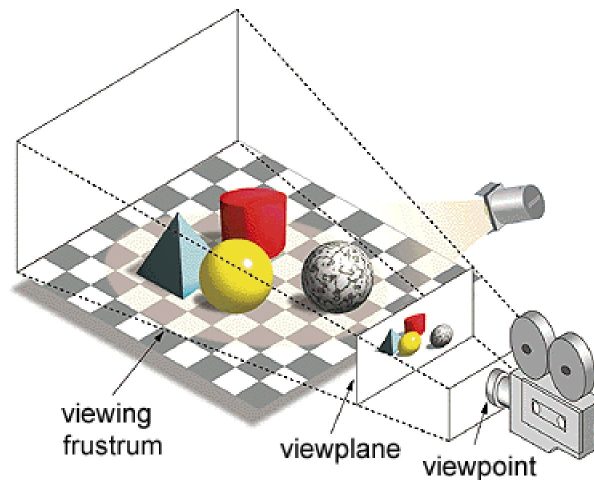

Fig. 2.4 Arrangement in Shader for 3D Modelling

FUNDAMENTAL QUESTIONS – How can we create, represent, synthesise, and render imagery on the computer display?

This is the fundamental question that has been studied in the field of computer graphics.

We can frame FOUR basic questions based on the fundamental questions.

- Imagery has been constructed from its constituent parts – How do represent those parts?
- How to synthesise the constituent parts to form complete realistic imagery?
- How to allow the user to manipulate the imaginary constituents on–screen?
- How to create the impression of motion? (For animation)

Graphics seeks answers to these questions.

Fig. 2.5 Configuration 3D Models – The smallest element is the triangle (Shown on the left side of the figure)

The triangle is considered the smallest element of Graphics Flow

NOTE –

We are using the term computer screen in an inclusive sense (includes small displays such as smartphones, tablets, etc., as well as large displays such as display walls and everything in between)

These variations indicate corresponding variations in the underlying computing platforms.

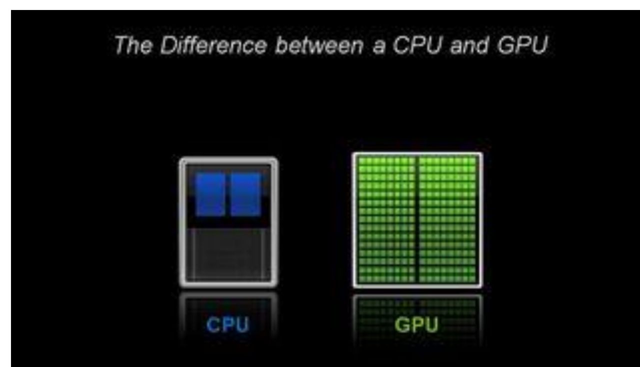

Fig. 2.6 CPU V/S GPU (Difference in number of cores)

Computer graphics seek efficient solutions to the four basic questions:

- Ways that make or try to optimise resource usage for a given platform.
- E.g., - displaying something on a mobile phone requires techniques different from saying on a desktop – due to differences in CPU speed, memory capacity, and power consumption issues in the two platforms.

In summary, "computer graphics is the process of rendering static images or animation (sequence of image) on computer screens efficiently".

2.2 NEED FOR PARALLELISM

- The pipeline is a vital programming pattern, while GPU, designed chiefly for data-level parallel executions, lacks an efficient mechanism to support pipeline programming and implementations.[12]
- Graphics operations – Highly parallel
- Ex – Consider modelling transformation stage - We apply transformation (e.g., rotation) to vertices
- Transformation = multiplication of transformation matrix with vertex vector - Same vectors – matrix multiplication is done for all vertices we want to transform.Instead of serial multiplication of one matrix-vector pair at a time, a significant gain in the

performance if we perform on all vectors simultaneously - Important in real-time rendering – millions of vertices are processed per second.

2.3 WHY GPUs

- CPUs cannot take advantage of this inherent parallelism in graphics operations – they are not designed to do that.
  - Nowadays, almost all graphics systems come with a separate graphics card containing its own processing unit and memory elements – known as a graphics processing unit or GPU.
- MULTICORE - GPU is a multicore system that contains many cores or unit processing elements.
  - Each core has a stream processor – that works on data streams
- SM - Cores capable of performing simple integer and floating-point arithmetic operations only.
  - Multiple cores are grouped to form streaming multiprocessors (SM)
- SIMD Idea - Consider the geometric transformation of vertices –
  - Instruction (Multiplication) same; data (vertex vectors) varies.
  - An instance of single instruction multiple data (SIMD)

2.4 GPU ORGANISATION – HOW DOES IT WORK

- Each SM is designed to perform SIMD operations.
- Most real-time graphics systems assume scenes made of triangles.
  - A surface such as quadrilaterals or curved surfaces is converted to meshes.
- Through APIs supported I graphics library (OpenGL/ Direct3D), triangles are transmitted to GPU one vertex at a time.
  - GPU assembles vertex into the triangle
- Vertices are expressed with homogenous coordinates.
- Objects they define are represented in local/ modelling coordinates.
- GPU performs modelling transformation on vertices.
- Transformation (single/ composite) achieved with a single matrix (transformation) – vector (point) multiplication.
- Multicore GPU performs multiple such operations simultaneously – multiple vertices simultaneously transformed.

- Output – stream of the triangle.
- In the world coordinate system - The viewer is located at the origin ad view direction aligned with the axis.
- Next – GPU computes vertex colour asked on light defied for the screen.
    - Recall colour ca e computed y vector dot product and a series of add ad multiply operations.
    - GPU performs simultaneously for multiple vertices.
- Next stage – each coloured 3D vertex is projected onto the view plane.
    - GPU does this using matrix-vector multiplication.
    - Output – stream of the triangles screen/device coordinates ready to be converted to pixels.
- Each device's space triangle overlaps some pixels on the screen.
- In the rasterization stage, these pixels are determined.
- GPU designers over the years incorporated many rasterization algorithms.
    - All exploit one crucial observation: each pixel can be treated independently from the other pixels.
- This leads to the possibility of handling all pixels in parallel.
    - Thus, given the device space triangle, we can simultaneously determine the pixel colour for all pixels.
- During the pixel processing stage, two more activities take place.
    - Surface texturing
    - Hidden surface removal
- The most straightforward surface texturing method – texture images draped over geometry to give an illusion of detail.
    - Pixel colour replaced by texture colour
- GPUs store textures in high-speed memory.
    - Each pixel calculation must access it.
    - Access very regular (nearly pixels tend to access nearby texture image locations) – specialised memory caches to reduce access time.

CHAPTER 3
VERIFICATION FOR A GRAPHICS CHIP

3.1 WHAT IS VERIFICATION?

The technique is used to analyse the functional correctness of the design under test (DUT). It is crucial to understand the verification need for the design and implementation requirement. To verify that the result of the provided sequence or task is as expected. Chip level verification of processor-based IC designs predominantly uses directed test cases implemented in high-level programming languages. [4]

The proposed chip-level verification methodology aims at developing an enhanced IC verification environment.[4]

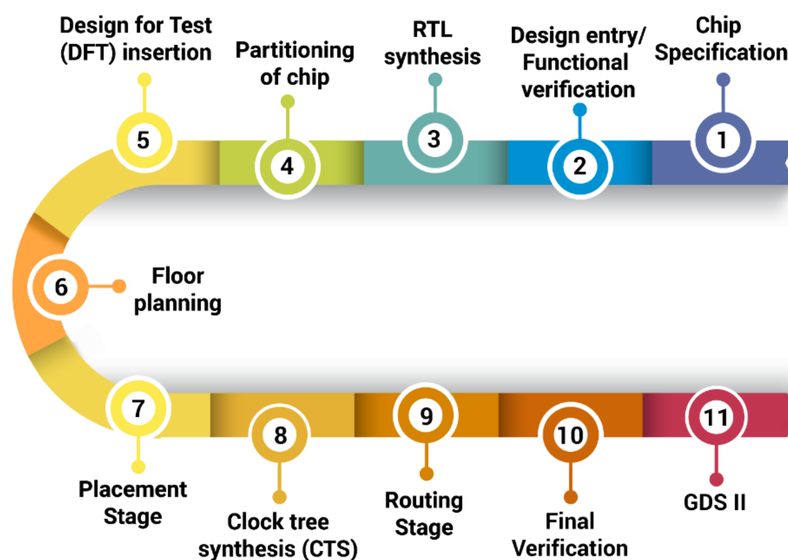

Fig. 3.1 Chip Development Flow

3.2 VERIFICATION PROBLEMS

- Is the specification right – The specification means the requirement of the design. The expectation or the intention of the sequence?
- Does the design (DUT) team acknowledge the spec correctly?
- Are the blocks accomplished fine?
- Are the interfaces between the different blocks have been implemented fine?

- Did we able to get the desired functionality?
- …………..

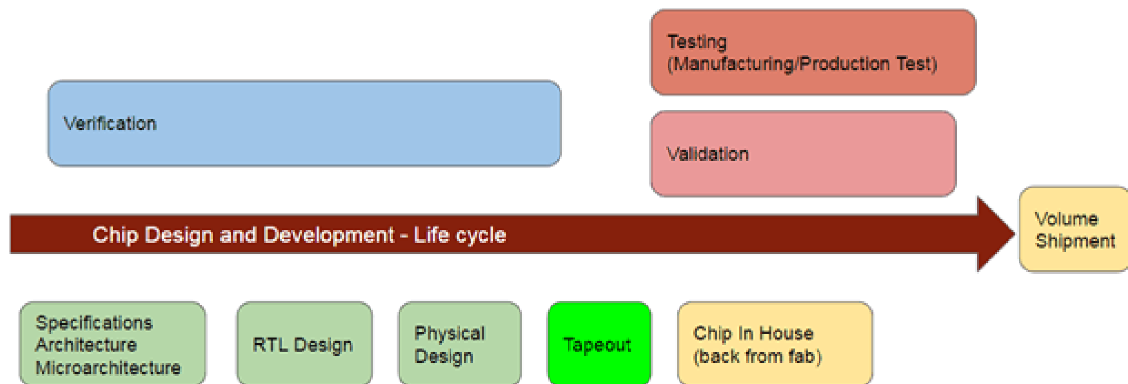

Fig. 3.2 Maximum time of a Chip Design and Development is taken in Verification.

3.3 VERIFICATION COMPLEXITY

For a single flip–flop:

- Number of states = 2
- Number of test patterns required = 4

For a Z80 microprocessor (~5K gates)

- Has 208 registers bits and 13 primary inputs
- Possible state transitions = 2 bits + inputs = 2221
- At 1M, IPS would take 1053 years to simulate all transitions

For a chip with 20M

- GIVE IT A THOUGHT??? (It is a manner of thought that with the increasing no. of chip elements / flops, the need for verifying the functionality of transistors is unimaginable)

3.4 WHEN IS VERIFICATION COMPLETE?

Some answers from real designers:

- When money and time is a constraining.
- When there is a need to deliver the product and the deadline is near.

- When verification of every line of the VHDL code.
- When sufficient testing has been done, we are unable to discover any new bug.
- Finally, we are left with no new thing …... No idea!!
- Even for the design team, it becomes too complex and they are unable to complete full coverage.
- When the potential vectors exceed significantly according to the time available for test.

3.5 VERIFICATION STRATEGIES

- Top-down verification approach – Simply means from the highest hierarchy to the lower one, from system to individual components.
- Bottom-up verification approach – Simply means from lowest hierarchy to higher one, which means individual components to the system.
- Platform-based verification approach – Verifying the already developed IPs for a readily available platform.
- System interface–based verification approach – Model each block at the interface level. It is generally used for final integration verification.

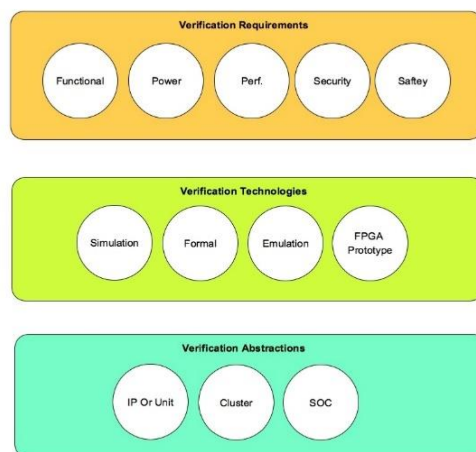

Fig. 3.3 Verification Strategies

3.6 VERIFICATION ENVIRONMENT

- Verification environment – Commonly referred to as testbench (environment)
- Definition of a testbench – When we are talking about Verification environment, is simply contains a set of components (such as bus functional models (BFMs), bus

monitors, memory modules) and the interconnect of such features with the design under – Verification (DUV)

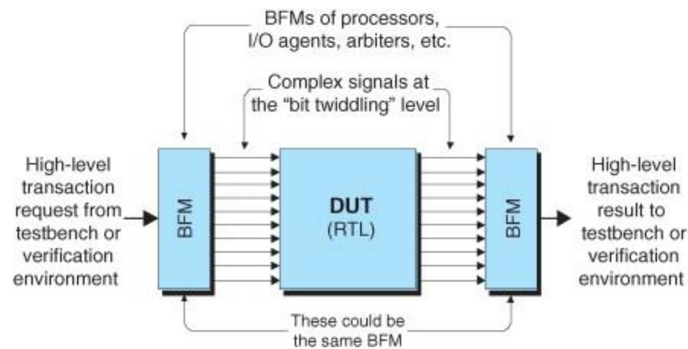

Fig. 3.4 Verification Environment

3.7 TESTBENCH DESIGN AND VERIFICATION TESTS

- Auto or semi-auto stimulus generator is preferred
    - Automatic reaction checking is extremely proposed.
    - There is also the need of the following design techniques:
        - Testbench in VHDL – Testbench is preferably required in VHDL
        - Testbenches in programming language interface (PLI)
        - Waveform
        - Specification
- Random testing
    - It is the way or a methodology in which engineers are provided with seniors which they do not anticipate.
    - Functional testing – User-provided functional patterns and sequences to verify.
    - Corners case testing is done.
    - Real code testing – Avoid misunderstanding the specification.
- Regression testing
    - It highly needs to ensure that fixing a bug will not introduce another bug(s)
    - The regression test system should be self-automated
    - Add new directed tests.

3.8 VERIFICATION PLAN AND BENEFITS

Verifications have high benefits, which can be listed as follows:

- Verification plan is a part of the design reports
    - Test policy/ test plan for both blocks level and top-level modules is performed.
    - Testbench components.
    - Necessary to have a set of verification tools and flows.
    - Simulation environment, including block diagram and test flow.
    - Key features needed to be verified in both chip-level – Block and module.
    - The regression test environment and procedure is be done correctly to justify the output
    - Clear criteria to determine whether the Verification is successfully completed.

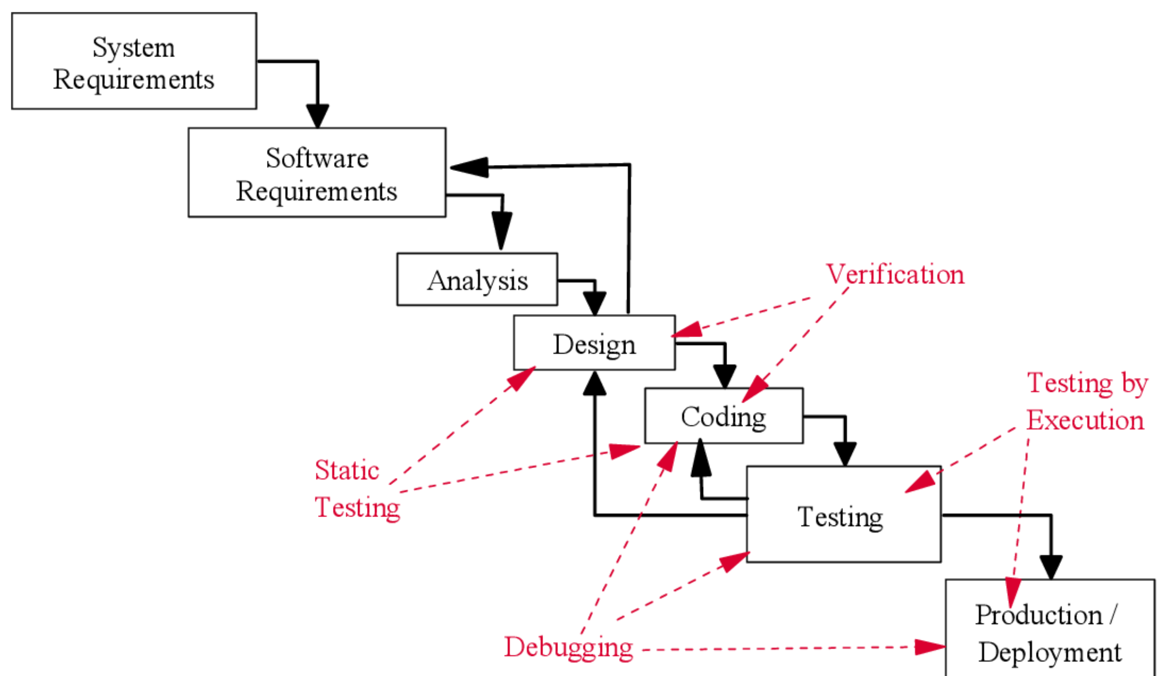

Fig. 3.5 Stages in which Verification Flow

3.9 TOOLS FOR VERIFICATION

- Dynamic Simulation
    - RTL and GLS
    - Event-driven, cycle – based, Transaction – based
    - Coverage driven Verification
    - SV, C, UVM …… The UVM can realise such testbench architectures with coverage has driven verification environments useful for constrained random testing. [2]

    - VCS, per spec….
- Formal Verification
    - Property verification
    - Logic Equivalence check
- Power-aware verification
    - UPF

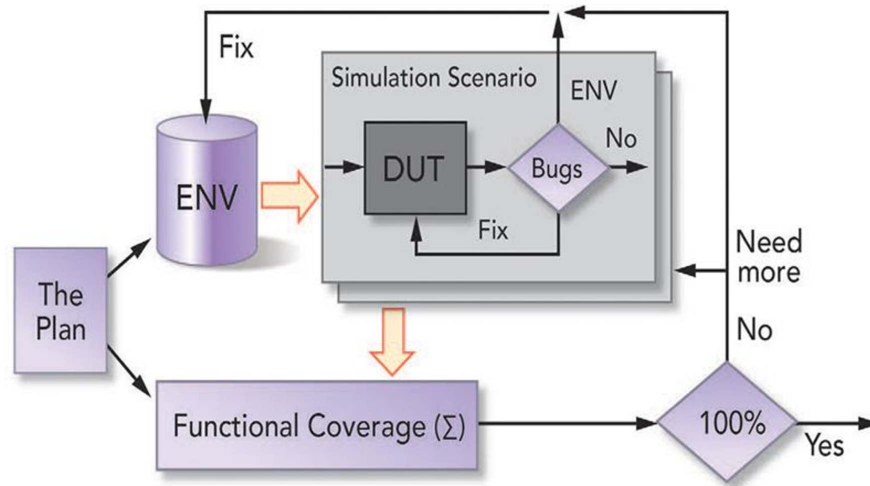

Fig. 3.6 Dynamic Simulation

3.10 POWER-AWARE VERIFICATION

- UPF
- RTL and GLS
- Dynamic simulations crossing low power
- Power estimation

3.11 VERIFICATION SCOPE

- 75% Chip development cycle dominated by Verification
- Number of IP's and complexity are increasing day – by – day as we are heading towards the sub deep micron technology, and it is highly needed to justify this process
- A lot of scope for the invasion to find the design issues and methodologies
- We can acquire knowledge from architecture to silicon debugging.

CHAPTER 4

IMPORTANCE OF DEBUG

4.1 NEED FOR DEBUG

Debugging hardware is very difficult, especially if it is not your hardware design or set-up problems during production.[14] The debug controller is the component in the graphics core to support the post-silicon and software debug requirements. Detecting and isolating bugs that arise only at high processor counts is a challenging task. Over several years, we have implemented a particular debugging method, called "relative debugging," that supports debugging applications as they evolve or are ported to larger machines.[15]

The debug controller gives visibility to the internal state of the GPU. The debug controller interacts with most GPU building blocks, gathers internal state information, and presents it to the outside world.

It provides visibility to internal microarchitecture. It can trace states through the AMBA Trace Bus. It provides it with the ability to detect a specific pattern that matches the internal condition. It also shares the ability to trace out all debug signals from all the blocks. Debug is performed for two GPU blocks synchronously without any additional latency. It gives the freedom to trace down the pipeline flow events passing through the GPU Blocks. The debug bus data is used for multiple debug operations such as debug, trace, counting etc. The debug bus data path is segmented into four regions. The debug data bus is designed to match the pre-defined pattern. The debug bus data provides general-purpose input and output ports.

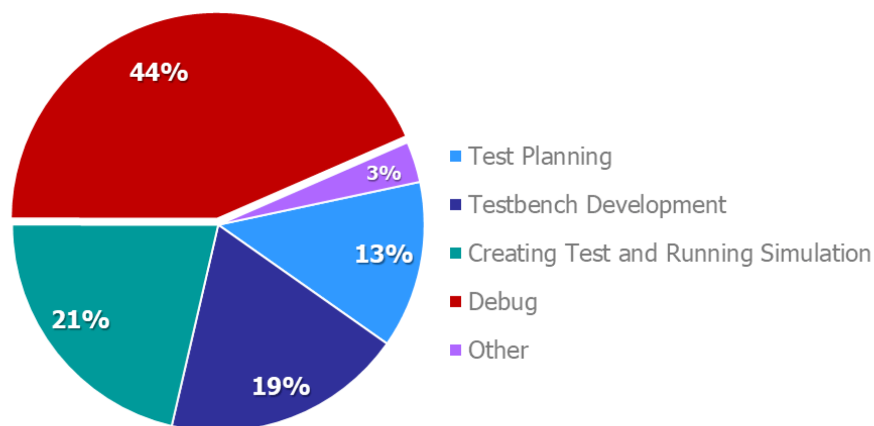

Fig. 4.1 Debug takes a maximum of Verification timespan

The debug bus controller receives the register configuration from the host through the interface. The register configuration from the host is decoded and sent to all the GPU blocks. They have different blocks for configuring additional functionality. They have other modules defined in separate blocks and finally they are connected to the bus or interface.

The GPU blocks decode the configuration and send the debug data through the data path. The event control module controls the event sent by the block through the debug data bus. The data is then fed to the data processing unit, where the data is used for analysis. The data processing unit also generates the trace data for transportation.

The GPU blocks define what architecture and microarchitecture signals are added in the debug bus. The debug bus controller provides an interface with the debug bus configuration. The blocks team verifies a mapping table connectivity.

4.2 HOW DEBUG IS PERFORMED

The process of debugging passes through a set of failing sequences. The need is to find the bug and debug it according to the specification of the design. Every sequence has its pre-defined intention. The aim is to understand the purpose which is provided by the specification and need to know the cause of that particular failure. Now, the case can be two. Since Verification is considered one of the relatively early stages of chip development, debugging is a bit easier and crucial. Since we are debugging at the Verification stage, the cause of failure could be two:

- Design failure
- Testbench failure

Indeed, it is essential to understand whether it's a design failure or testbench failure.

Once we conclude the type of failure, the debugging path is designed in that particular case.

If it's a test bench failure, the whole path of the signal is traced backed to define the actual cause of error. They could be done in multiple ways as:

- Analysing the testbench
- We are using any EDA tools to generate the waveform and debugging through waveforms and signals.
- Checking the sequence speciation or functionality
- If the provided specification is feasible for that case or not.

And if it's a design failure. Then the test case is sent back to the design team by stating it's the fault on their end. The design team decides whether the error that occurred at that place is a fault of design (DUT) or the fault is happening because of the specification Mismatch. Specification mismatch means that the functionality we are trying to obtain in that scenario is not feasible. We should change the functionality and design the DUT again with the new spec.

So, the aim of the verification stage is to remove all the potential bugs at this stage. And if it is not covered, this bug will be carried forward to a later stage. Even at that stage, a pre-check is performed, and if the bug is found at that stage the later team will return that bug to Verification team and hence the debugging process is again stated.

CHAPTER 5

DEBUG CHECKER

5.1 CHECKER – EXPLANATION

The functionality of the logger is defined in a certain way that: - The output flow is from two different paths. The following block diagram will help in getting the actual flow of Testbench.

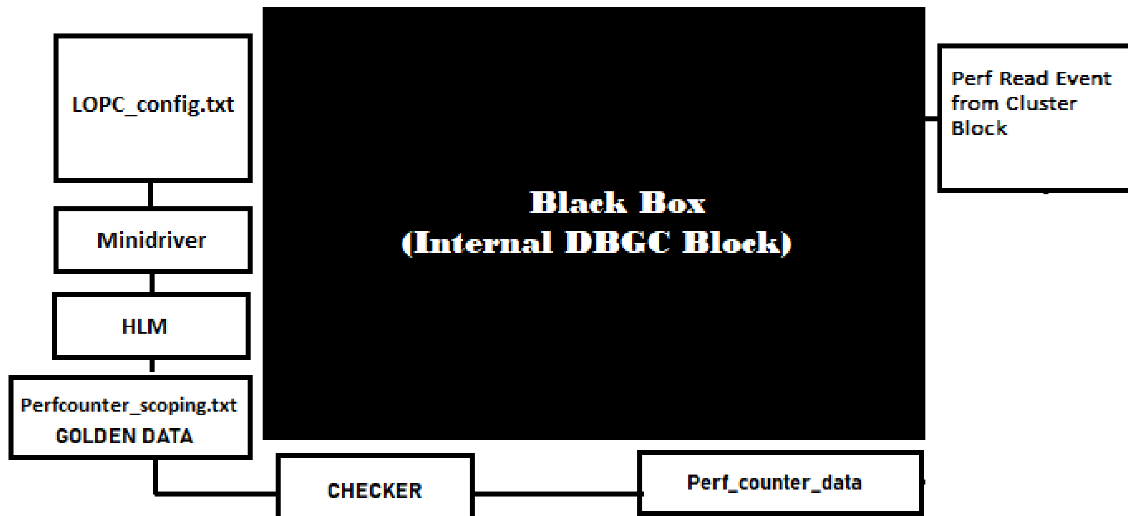

Fig. 5.1 DBGC Interface

So, the logger or the checker aims to compare the data from the Perf Read Event block, which is coming from the Cluster Block and The HLM Data, which is the golden response.

It checks the data from both blocks. It generates a MISMATCH whenever different output or data is found.

Now, the checker is also designed in a way that it helps in debugging.

The debugging process includes, whenever there is a MISMATCH for that particular output, the checker or logger segregates the data. In that way, the potential bug is found. These debugging enhancements offer increased traceability and observability within assertion checkers and improved metrics relating to the coverage of assertion checkers.[13]

The glimpses of output are shown below. That helps in understanding the issue clearly.

```
10002789165440895432778900000000
10005432278976544089778900000000
16544089789100054357788000000000
11004089892c90043277890000000000
1000543278940897f892c90000000000
1000243289eef9d24588210000000000
1000123289456798536e780000000000
100054328912453698a4278000000000
10005432e975329512ba780000000000
10004089092c90543277890000000000
100054325239874ab5e7789000000000
1000543278978563214 12ef000000000
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c5432778900000458763594410000
1200a43240897f89c90778900000c000
40005c327789000012453698ab750000
700054327789cc78a456b23600000000
900954340897f8940897f89900000000
1000543240897f892c907ac540000000
1000543277812458abc0245900000000
100054240897f89c2408978900000000
2a1105432240897f8240877890000000
2000521105432240897f778900000000
10004089092c90543277890000000000
100054325239874ab5e7789000000000
10005432789785632141 2ef000000000
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c5432778900000458763594410000
1200a43240897f89c90778900000c000
40005c327789000012453698ab750000
700054327789cc78a456b23600000000
900954340897f8940897f89900000000
1000543240897f892c907ac540000000
100054325239874ab5e7789000000000
10005432789785632141 2ef000000000
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
```

Fig. 5.2 Data coming from HLM (At the start of HLM)

The HLM data is the golden data by which the test bench cases are compared. So, they are considered to be the reference point for the verification test cases.

```
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c5432778900000458763594410000
1200a43240897f89c90778900000c000
40005c327789000012453698ab750000
700054327789cc78a456b23600000000
900954340897f8940897f89900000000
1000543240897f892c907ac540000000
100054325239874ab5e7789000000000
10005432789785632141 2ef000000000
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c5432778900000458763594410000
1200a43240897f89c90778900000c000
40005c327789000012453698ab750000
700054327789cc78a456b23600000000
900954340897f8940897f80000000000
900954340897f8940897f89900000000
1000543240897f892c907ac540000000
1000543277812458abc0245900000000
100054240897f89c2408978900000000
2a1105432240897f8240877890000000
2000521105432240897f778900000000
10004089092c90543277890000000000
100054325239874ab5e7789000000000
10005432789785632141 2ef000000000
1a000543212344521987477890126700
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c5432778900000458763594410000
1200a43240897f89c90778900000c000
40005c327789000012453698ab750000
700054327789cc78a456b23600000000
900954340897f8940897f89900000000
1000543240897f892c907ac540000000
100054325239874ab5e7789000000000
10005432789785632141 2ef000000000
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
```

Fig. 5.3 Data coming from HLM (At the mid of HLM)

As seen in the above fig., the checker generates a MISMATCH whenever it is accounting for a difference. Now, if the debugging has to be done independently, it is difficult to figure out the potential bug.

The testbench contains multiple failures. And these failures in not just in 10s or 100s, but it could be 1000s or even more.

The accounting is also that if the debug it done manually, it could also result in that if the debug is detected in bit 4, it could be counted as in bit 5, as a man error.

This could make a testbench verification even worse, and even after spending more than a day in a single debug, we conclude nothing. So, testbench debugging can be very tedious work which requires a lot of attention and patience. The aim is to reduce manual labour. This could be achieved by designing a script. The purpose of script is to run in a testbench. Hence whenever a mismatch is accounted is segregates the data to reach as close as to the potential bug.

The testbench can have multiple failures, such as an ERROR or a FATAL. The ERROR has less effect as if an ERROR is obtained, at least the compile is completed, but when a FATAL is obtained, the compile even stops at the point of FATAL.

```
act_perf_counter_pkt_br[10] = 10002789165440895432778900000000
act_perf_counter_pkt_br[11] = 10005432278976544089778900000000
act_perf_counter_pkt_br[12] = 16544089789100054357788000000000
act_perf_counter_pkt_br[13] = 11004089892c90043277890000000000
act_perf_counter_pkt_br[14] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[15] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[16] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[17] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[18] = 10005432e975329512ba780000000000
act_perf_counter_pkt_br[19] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[20] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[21] = 100054327897856321412ef000000000
act_perf_counter_pkt_br[22] = 10005432123445219874778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[23] = 1000543210ea96bc3657778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[24] = 100054327789ab75312c012450000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c327789000012453698ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[29] = 90095432778900000000
act_perf_counter_pkt_br[30] = 1000543240897f892c907ac540000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 100054240897f89c2408978900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[35] = 20005432a68990095432778700000000
act_perf_counter_pkt_br[36] = 40005432778900000000
act_perf_counter_pkt_br[37] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[38] = 10001000543289125432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[39] = 10005432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[40] = 10900954327780000100054328912000
act_perf_counter_pkt_br[41] = 10005439009543277892778900000000
act_perf_counter_pkt_br[42] = 10090095432778905432778900000000
act_perf_counter_pkt_br[43] = 10432778905430054327789000000000
```

Fig. 5.4 Data obtained by analysing every packet and getting MISMATCH correspondingly [Packet 22]

```
act_perf_counter_pkt_br[110] = 1000278916544089543277890000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[111] = 1000543227897654408977890000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[112] = 1654408978910005435778800000000
act_perf_counter_pkt_br[113] = 11004089892c90043277890000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[114] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[115] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[116] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[117] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[118] = 10005432e975329512ba780000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[119] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[120] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[121] = 100054327897856321412ef000000000
act_perf_counter_pkt_br[122] = 10005432123445219874778900000000
act_perf_counter_pkt_br[123] = 1000543210ea96bc3657778900000000
act_perf_counter_pkt_br[124] = 100054327789ab75312c012450000000
act_perf_counter_pkt_br[125] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[126] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[127] = 40005c327789000012453698ab750000
act_perf_counter_pkt_br[128] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[129] = 90095432778054327789090500000000
act_perf_counter_pkt_br[130] = 1000543240897f892c907ac540000000
act_perf_counter_pkt_br[131] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[132] = 100054240897f89c2408978900000000
act_perf_counter_pkt_br[133] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[134] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[135] = 20005432a68990095432778700000000
act_perf_counter_pkt_br[136] = 40005432778900543205778900000000
act_perf_counter_pkt_br[137] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[138] = 10001000543289125432778900000000
act_perf_counter_pkt_br[139] = 10005405432778900543278900000000
act_perf_counter_pkt_br[140] = 10900954327780000100054328912000
act_perf_counter_pkt_br[141] = 10005439009543277892778900000000
act_perf_counter_pkt_br[142] = 10090095432778905432778900000000
act_perf_counter_pkt_br[143] = 10005432778054327789005900000000
act_perf_counter_pkt_br[144] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[145] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[146] = 700054327789cc78a456b23600000000
```

Fig. 5.5 Data obtained by analysing every packet and getting MISMATCH correspondingly [Packet 110]

```
act_perf_counter_pkt_br[10] = 10002789165440895432778900000000
act_perf_counter_pkt_br[11] = 10005432278976544089778900000000
act_perf_counter_pkt_br[12] = 16544089789100054357788000000000
act_perf_counter_pkt_br[13] = 11004089892c90043277890000000000
act_perf_counter_pkt_br[14] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[15] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[16] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[17] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[18] = 10005432e975329512ba780000000000
act_perf_counter_pkt_br[19] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[20] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[21] = 100054327897856321412ef000000000
act_perf_counter_pkt_br[22] = 10005411001028180000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_10
[Data] = 0
act_perf_counter_pkt_br[23] = 1000541200103c39e000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5412
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_1c
[Data] = 0
act_perf_counter_pkt_br[24] = 10005413011028180000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5413
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_17
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c327789000012453698ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
```

Fig. 5.6 Data obtained by running the checker in that code and segregating the bits for MISMATCH terms [Packet 22]

```
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_1c
[Data] = 0
act_perf_counter_pkt_br[24] = 1000541301102818000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5413
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_17
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c3277890000124536 98ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[29] = 1000541e041090095432778900000000
act_perf_counter_pkt_br[30] = 1000541e04103d470000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 541e
[perf_counter_batch_id] = 4
[Pipe_id] = BR
[Perf_Read_Address] = 3d
[Block Sequence] = PERF_CLUSTER_PC_19
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 1000544e05103c390000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 544e
[perf_counter_batch_id] = 5
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_22
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_26
```

Fig. 5.7 Data obtained by running the checker in that code and segregating the bits for MISMATCH terms [Packet 24]

The explanation of these data MISMATCH will be found in next chapter.

5.2 PSEUDO CODE

```
//Performance _checker Design
module logger(input [127:0] perf_data);
bit [1:0] unused_1
bit [1:0] Packet_id;
bit [27:0] Timestamp;
bit [7:0] perf_counter_batch_id;
bit [1:0] unused_2;
bit [1:0] pipe_id;
bit [3:0] umused_3;
bit [5:0] Perf_Read_Address;
bit unused;
 bit [6:0] Cluster_Block_ID;
bit [6:0] Data;
string Packet_Sequence [bit[1:0]];
string Pipe_Sequence [bit[1:0]];
string Block_Sequence [bit[7:0]];
Pipe_Sequence = '{
2'h00 : "A1",
2'h01 : "A2",
2'h10 : "A3",
2'h11 : "A4",
};
Packet_Sequence = '{
2'h00 : "B1",
2'h01 : "B2",
2'h10 : "B3",
2'h11 : "B4",
};
Block_Sequence = '{
```

8’hx00 : “Block_Sequence_1”,

8’hx01 : “Block_Sequence_2”,

8’hx02 : “Block_Sequence_3”,

8’hx03 : “Block_Sequence_4”,

8’hx04 : “Block_Sequence_5”,

8’hx05 : “Block_Sequence_6”,

8’hx06 : “Block_Sequence_7”,

8’hx07 : “Block_Sequence_8”,

8’hx08 : “Block_Sequence_9”,

8’hx09 : “Block_Sequence_10”,

8’hx0a : “Block_Sequence_11”,

8’hx0b : “Block_Sequence_12”,

8’hx0c : “Block_Sequence_13”,

8’hx0d : “Block_Sequence_14”,

…………

……………….

…………………….

………………...

…………...

8’hx4a : “Block_Sequence_74”,

8’hx4b : “Block_Sequence_75”,

8’hx4c : “Block_Sequence_76”,

8’hx4d : “Block_Sequence_77”,

8’hx4e : “Block_Sequence_78”,

```
8’hx4f : “Block_Sequence_79”,

8’hx50 : “Block_Sequence_80”,

8’hx51 : “Block_Sequence_81”,

8’hx52 : “Block_Sequence_82”,

8’hx53 : “Block_Sequence_83”,

8’hx54 : “Block_Sequence_84”,

8’h0x55 : “Block_Sequence_85”,


};


Unused_1 = perf_data [127:126];

Packet_id = perf_data [125:124];

Timestamp = perf_data [123:96];

Perf_counter_batch_id = perf_data [95:88];

Unused_2 = perf_data [87:86];

Pipe_id = perf_data [85:84];

Unused_3 = perf_data [83:80];

Perf_Read_Address = perf_data [79:72];

Unused_4 = perf_data [71];

Cluster_Block_ID = perf_data [70:64];

Data = perf_data [63:0];


$display(“[Packet_id] = %s”, Packet_Sequence [Packet_id]);
```

```
$display("[Timestamp] = %0h", Timestamp);

$display("[perf_counter_batch_id] = %0h", perf_counter_batch_id);

$display("[Pipe_id] = %s", Pipe_sequence[Pipe_id]);

$display("[Perf_Read_Address] = %0h", Perf_Read-Address);

$display("[Block_Sequence] = %s", Block_Sequence[Cluster_Block_ID]);

$display("[Data] = %0h", Data);


endfunction

endmodule
```

CHAPTER 6

OUTPUTS

```
1000278916544089543277890000000
1000543227897654408977890000000
1654408978910005435778800000000
11004089892c9004327789000000000
1000543278940897f892c9000000000
1000243289eef9d2458821000000000
10001232894567985 36e780000000000
```

Fig. 6.1 Showing data coming from the HLM module (Considered the golden data)

```
1a000543212344521987477890000000
1000543210ea96bc3657778900000000
100054327789ab75312c012450000000
203c543277890000045876359441000
```

Fig. 6.2 Showing data coming from the HLM module (Considered the golden data)

The above-shown figures are the output from HLM. HLM is data coming from High-level descriptive languages as a base of the production. So, hence HLM data is considered the data that needs to be used as reference data that is the GOLDEN DATA. It is complicated to find the bug and configure it. It is challenging to understand the point of error.

```
act_perf_counter_pkt_br[10] = 10002789165440895432778900000000
act_perf_counter_pkt_br[11] = 10005432278976544089778900000000
act_perf_counter_pkt_br[12] = 16544089789100054357788000000000
act_perf_counter_pkt_br[13] = 11004089892c90043277890000000000
act_perf_counter_pkt_br[14] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[15] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[16] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[17] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[18] = 10005432e975329512ba780000000000
act_perf_counter_pkt_br[19] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[20] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[21] = 10005432789785632141 2ef000000000
act_perf_counter_pkt_br[22] = 10005432123445219874778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[23] = 1000543210ea96bc3657778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[24] = 100054327789ab75312c012450000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c3277890000012453698ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[29] = 900954327789000000000
act_perf_counter_pkt_br[30] = 1000543240897f892c907ac540000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 100054240897f89c2408978900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[35] = 20005432a6899009543277870000000
act_perf_counter_pkt_br[36] = 400054327789000000000
act_perf_counter_pkt_br[37] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[38] = 10001000543289125432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[39] = 100054327789000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[40] = 10900954327780000100054328912000
act_perf_counter_pkt_br[41] = 10005439009543277892778900000000
act_perf_counter_pkt_br[42] = 10090095432778905432778900000000
act_perf_counter_pkt_br[43] = 10432778905430054327789000000000
```

Fig. 6.3 shows data of testbench describing the positions of MISMATCH in the design [Packet 22]

```
act_perf_counter_pkt_br[110] = 10002789165440895432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[111] = 10005432278976544089778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[112] = 16544089789100054357788000000000
act_perf_counter_pkt_br[113] = 11004089892c90043277890000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[114] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[115] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[116] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[117] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[118] = 10005432e975329512ba780000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[119] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[120] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[121] = 100054327897856321412ef000000000
act_perf_counter_pkt_br[122] = 10005432123445219874778900000000
act_perf_counter_pkt_br[123] = 1000543210ea96bc3657778900000000
act_perf_counter_pkt_br[124] = 100054327789ab75312c012450000000
act_perf_counter_pkt_br[125] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[126] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[127] = 40005c3277890000012453698ab750000
act_perf_counter_pkt_br[128] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[129] = 90095432778054327789090500000000
act_perf_counter_pkt_br[130] = 1000543240897f892c907ac540000000
act_perf_counter_pkt_br[131] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[132] = 100054240897f89c2408978900000000
act_perf_counter_pkt_br[133] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[134] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
act_perf_counter_pkt_br[135] = 20005432a6899009543277870000000
act_perf_counter_pkt_br[136] = 40005432778900543205778900000000
act_perf_counter_pkt_br[137] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[138] = 10001000543289125432778900000000
act_perf_counter_pkt_br[139] = 10005405432778900543278900000000
act_perf_counter_pkt_br[140] = 10900954327780000100054328912000
act_perf_counter_pkt_br[141] = 10005439009543277892778900000000
act_perf_counter_pkt_br[142] = 10090095432778905432778900000000
act_perf_counter_pkt_br[143] = 10005432778054327789005900000000
act_perf_counter_pkt_br[144] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[145] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[146] = 700054327789cc78a456b23600000000
```

Fig. 6.4 shows data of testbench describing the positions of MISMATCH in the design [Packet 110]

Now the mismatch is found in the sequence. They are the point of error and need to be debug. But still, it is difficult to get the bit which results in error. The bits signify the sub-block. Every bit has been mapped to a particular block of Debug Controller. So, every bit reflects the behaviour of the block.

So, if any possible check is implemented in the code, which can segregate the bits as per the functionality of the block helps in the process of debugging and makes it more efficient.

The following outputs can very well explain the functionality of the checker.

```
act_perf_counter_pkt_br[10] = 100027891654408954327789000000000
act_perf_counter_pkt_br[11] = 100054322789765440897789000000000
act_perf_counter_pkt_br[12] = 165440897891000543577880000000000
act_perf_counter_pkt_br[13] = 11004089892c90043277890000000000
act_perf_counter_pkt_br[14] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[15] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[16] = 1000123289456798536e780000000000
act_perf_counter_pkt_br[17] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[18] = 10005432e975329512ba780000000000
act_perf_counter_pkt_br[19] = 10004089092c90543277890000000000
act_perf_counter_pkt_br[20] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[21] = 100054327897856321412ef000000000
act_perf_counter_pkt_br[22] = 100054110010281800000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_10
[Data] = 0
act_perf_counter_pkt_br[23] = 1000541200103c39e000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5412
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_1c
[Data] = 0
act_perf_counter_pkt_br[24] = 100054130110281800000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5413
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_17
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c327789000012453698ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
```

Fig. 6.5 shows data of testbench describing the positions of MISMATCH in the design [Packet 22] and with segregations in the packet as well.

```
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_1c
[Data] = 0
act_perf_counter_pkt_br[24] = 100054130110281800000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5413
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_17
act_perf_counter_pkt_br[25] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[26] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[27] = 40005c327789000012453698ab750000
act_perf_counter_pkt_br[28] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[29] = 1000541e041090095432778900000000
act_perf_counter_pkt_br[30] = 1000541e04103d470000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 541e
[perf_counter_batch_id] = 4
[Pipe_id] = BR
[Perf_Read_Address] = 3d
[Block Sequence] = PERF_CLUSTER_PC_19
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 1000544e05103c390000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 544e
[perf_counter_batch_id] = 5
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_22
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_26
```

Fig. 6.6 shows data of testbench describing the positions of MISMATCH in the design [Packet 24] and with segregations in the packet as well.

```
act_perf_counter_pkt_br[30] = 1000541e04103d470000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 541e
[perf_counter_batch_id] = 4
[Pipe_id] = BR
[Perf_Read_Address] = 3d
[Block Sequence] = PERF_CLUSTER_PC_19
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 1000544e05103c390000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 544e
[perf_counter_batch_id] = 5
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_22
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 20005211054322408 97f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_26
act_perf_counter_pkt_br[35] = 20005432a68990095432778700000000
act_perf_counter_pkt_br[36] = 40005432778900000000
act_perf_counter_pkt_br[37] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[38] = 10001000543289125432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC
act_perf_counter_pkt_br[39] = 10005432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
```

Fig. 6.7 shows data of testbench describing the positions of MISMATCH in the design [Packet 30] and with segregations in the packet as well.

```
[Pipe_id] = BR
[Perf_Read_Address] = 3d
[Block Sequence] = PERF_CLUSTER_PC_19
act_perf_counter_pkt_br[31] = 1000543277812458abc0245900000000
act_perf_counter_pkt_br[32] = 1000544e05103c390000000000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 544e
[perf_counter_batch_id] = 5
[Pipe_id] = BR
[Perf_Read_Address] = 3c
[Block Sequence] = PERF_CLUSTER_PC_22
act_perf_counter_pkt_br[33] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[34] = 2000521105432240897f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC_26
act_perf_counter_pkt_br[35] = 20005432a68990095432778700000000
act_perf_counter_pkt_br[36] = 40005432720005432a6899c900000000
act_perf_counter_pkt_br[37] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[38] = 10001000543289125432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC
act_perf_counter_pkt_br[39] = 10005432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5411
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 28
[Block Sequence] = PERF_CLUSTER_PC
act_perf_counter_pkt_br[40] = 10900954327780000100054328912000
act_perf_counter_pkt_br[41] = 10005439009543277892778900000000
```

Fig. 6.8 shows data of testbench describing the positions of MISMATCH in the design [Packet 32] and with segregations in the packet as well.

```
act_perf_counter_pkt_br[110] = 10002789165440895432778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 2789
[perf_counter_batch_id] = 3
[Pipe_id] = BR
[Perf_Read_Address] = 54
[Block Sequence] = PERF_CLUSTER_PC_67
[Data] = 0
act_perf_counter_pkt_br[111] = 10005432278976544089778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5432
[perf_counter_batch_id] = 8
[Pipe_id] = BR
[Perf_Read_Address] = 43
[Block Sequence] = PERF_CLUSTER_PC_4e
[Data] = 0
act_perf_counter_pkt_br[112] = 16544089789100054357788000000000
act_perf_counter_pkt_br[113] = 11004089892c900432778900000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 4089
[perf_counter_batch_id] = 10
[Pipe_id] = BR
[Perf_Read_Address] = 78
[Block Sequence] = PERF_CLUSTER_PC_3d
[Data] = 0
act_perf_counter_pkt_br[114] = 1000543278940897f892c90000000000
act_perf_counter_pkt_br[115] = 1000243289eef9d24588210000000000
act_perf_counter_pkt_br[116] = 100012328945679853 6e780000000000
act_perf_counter_pkt_br[117] = 100054328912453698a4278000000000
act_perf_counter_pkt_br[118] = 10005432e975329512ba780000000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5432
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 5a
[Block Sequence] = PERF_CLUSTER_PC_55
[Data] = 0
act_perf_counter_pkt_br[119] = 10004089092c905432778900000000000
```

Fig. 6.9 shows data of testbench describing the positions of MISMATCH in the design [Packet110] and with segregations in the packet as well.

```
[Timestamp] = 5432
[perf_counter_batch_id] = 0
[Pipe_id] = BR
[Perf_Read_Address] = 5a
[Block Sequence] = PERF_CLUSTER_PC_55
[Data] = 0
act_perf_counter_pkt_br[119] = 10004089092c905432778900000000000
act_perf_counter_pkt_br[120] = 100054325239874ab5e7789000000000
act_perf_counter_pkt_br[121] = 10005432789785632141 2ef000000000
act_perf_counter_pkt_br[122] = 10005432123445219874778900000000
act_perf_counter_pkt_br[123] = 1000543210ea96bc3657778900000000
act_perf_counter_pkt_br[124] = 100054327789ab75312c012450000000
act_perf_counter_pkt_br[125] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[126] = 1200a43240897f89c90778900000c000
act_perf_counter_pkt_br[127] = 40005c3277890000124536 98ab750000
act_perf_counter_pkt_br[128] = 700054327789cc78a456b23600000000
act_perf_counter_pkt_br[129] = 90095432778054327789090500000000
act_perf_counter_pkt_br[130] = 1000543240897f892c907ac540000000
act_perf_counter_pkt_br[131] = 100054327781245 8abc0245900000000
act_perf_counter_pkt_br[132] = 100054240897f89c2408978900000000
act_perf_counter_pkt_br[133] = 21105432240897f82408778900000000
act_perf_counter_pkt_br[134] = 20005211054322408 97f778900000000
ERROR BR pipe :Perfcounter PKT MISMATCH
[Packet_id] = Performance Counter Packet
[Timestamp] = 5211
[perf_counter_batch_id] = 7
[Pipe_id] = BR
[Perf_Read_Address] = 22
[Block Sequence] = PERF_CLUSTER_PC_31
[Data] = 0
act_perf_counter_pkt_br[135] = 20005432a6899009543277870000000 0
act_perf_counter_pkt_br[136] = 40005432778900543205778900000000
act_perf_counter_pkt_br[137] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[138] = 10001000543289125432778900000000
act_perf_counter_pkt_br[139] = 10005405432778900543278900000000
act_perf_counter_pkt_br[140] = 10900954327780000100054328912000
act_perf_counter_pkt_br[141] = 10005439009543277892778900000000
act_perf_counter_pkt_br[142] = 10090095432778905432778900000000
act_perf_counter_pkt_br[143] = 10005432778054327789005900000000
act_perf_counter_pkt_br[144] = 300040897f892c905432778900000000
act_perf_counter_pkt_br[145] = 203c5432778900000458763594410000
act_perf_counter_pkt_br[146] = 700054327789cc78a456b23600000000
```

Fig. 6.10 shows data of testbench describing the positions of MISMATCH in the design [Packet 134] and with segregations in the packet as well.

CHAPTER 7

CONCLUSION

Since Verification is considered one of the most critical and time-consuming processes of chip design and development cycle, moreover, considering debug it is again a very critical and important part of Verification so this concludes the fact that Debug is a crucial part of complete chip development cycle.

Any manual error or man-made failure can have a significant repercussion on the complete flow.

So, the project aims to increase the efficiency of debugging process. The debug is the process which consumes maximum time in the Verification. The logger or the checker aims to compare the data from the Perf Read Event block, which is coming from the Cluster Block and The HLM Data, which is the golden responses.

It checks the data from both blocks. It generates a MISMATCH whenever different output or data is found.

Now, the checker is also designed in a way that it helps in debugging.

The debugging process includes, whenever there is a MISMATCH for that particular output the checker or logger segregates the data. In that way, the potential bug is found.

This has helped in reducing debugging time in many folds. The debug checker has multiple applications. Verification is a muti state progress which leads to its application in numerous steps. The checker can be introduced in the main code to check the fault in the listed blocks, can be introduced in the individual blocks to check the fault in sub-blocks and can be implemented at SOC level to check which module has MISMATCH.

REFERENCES

[1] A. Thalaimalai Vanaraj, M. Raj and L. Gopalakrishnan, "Functional Verification closure using Optimal Test scenarios for Digital designs," 2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT), 2020, pp. 535-538, doi: 10.1109/ICSSIT48917.2020.9214097.

[2] B. Vineeth and B. B. Tripura Sundari, "UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol," 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018, pp. 307-310, doi: 10.1109/ICACCI.2018.8554919.

[3] M. Jenihhin, X. Lai, T. Ghasempouri and J. Raik, "Towards Multidimensional Verification: Where Functional Meets Non-Functional," 2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), 2018, pp. 1-7, doi: 10.1109/NORCHIP.2018.8573495.

[4] S. Karlapalem and S. Venugopal, "Scalable, Constrained Random Software Driven Verification," 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2016, pp. 71-76, doi: 10.1109/MTV.2016.19.

[5] V. N. Guznenkov and P. A. Zhurbenko, "The Academic Discipline “Computer Graphics” for the Open Education System," 2018 IV International Conference on Information Technologies in Engineering Education (Inforino), 2018, pp. 1-4, doi: 10.1109/INFORINO.2018.8581738.

[6] I. Faraji, S. H. Mirsadeghi and A. Afsahi, "Topology-Aware GPU Selection on Multi-GPU Nodes," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 712-720, doi: 10.1109/IPDPSW.2016.44.

[7] N. Gaikwad and V. N. Patil, "Verification of AMBA AXI On-Chip Communication Protocol," 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), 2018, pp. 1-5, doi: 10.1109/ICCUBEA.2018.8697587.

[8] P. Giridhar and P. Choudhury, "Design and Verification of AMBA AHB," 2019 1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE), 2019, pp. 310-315, doi: 10.1109/ICATIECE45860.2019.9063856.

[9] S. Divekar and A. Tiwari, "Interconnect matrix for multichannel AMBA AHB with multiple arbitration technique," 2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE), 2014, pp. 1-5, doi: 10.1109/ICGCCEE.2014.6922230.

[10] A. Roychoudhury, T. Mitra and S. R. Karri, "Using formal techniques to debug the AMBA system-on-chip bus protocol," 2003 Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 828-833, doi: 10.1109/DATE.2003.1253709.

[11] Yi-Ting Lin, Chien-Chou Wang and Ing-Jer Huang, "AMBA AHB bus potocol checker with efficient debugging mechanism," 2008 IEEE International Symposium on Circuits and Systems (ISCAS), 2008, pp. 928-931, doi: 10.1109/ISCAS.2008.4541571.

[12] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi and W. Chen, "VersaPipe: A Versatile Programming Framework for Pipelined Computing on GPU," 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017, pp. 587-599.

[13] M. Boule, J. Chenard and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug," 2006 International Conference on Computer Design, 2006, pp. 294-299, doi: 10.1109/ICCD.2006.4380831.

[14] M. O. Pahanel, "Hardware checker module," 2012 35th IEEE/CPMT International Electronics Manufacturing Technology Conference (IEMT), 2012, pp. 1-3, doi: 10.1109/IEMT.2012.6521753.

[15] M. N. Dinh, D. Abramson and C. Jin, "Scalable Relative Debugging," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 740-749, March 2014, doi: 10.1109/TPDS.2013.86.

[16] A. M. Gharehbaghi and M. Fujita, "Transaction-based post-silicon debug of many-core System-on-Chips," Thirteenth International Symposium on Quality Electronic Design (ISQED), 2012, pp. 702-708, doi: 10.1109/ISQED.2012.6187568.

[17] M. Gao and K. Cheng, "A case study of Time-Multiplexed Assertion Checking for post-silicon debugging," 2010 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 90-96, doi: 10.1109/HLDVT.2010.5496657.

[18] M. H. Neishaburi and Z. Zilic, "On a New Mechanism of Trigger Generation for Post-Silicon Debugging," in IEEE Transactions on Computers, vol. 63, no. 9, pp. 2330-2342, Sept. 2014, doi: 10.1109/TC.2013.107.