Politecnico di Torino

Politecnico di Torino
Master Degree in Electronic Engineering

Migration of register verification methodologies, with focus on workflow automation and profiling

Candidate:
Gabriele Sanna

Supervisors:
Prof. Guido Masera

Academic Year 2024-2025

Politecnico di Torino
Dipartimento di Elettronica e Telecomunicazioni

Politecnico di Torino

Politecnico di Torino

Master Degree in Electronic Engineering

Migration of register verification methodologies, with focus on workflow automation and profiling

Candidate:
Gabriele Sanna

Supervisors:
Prof. Guido Masera

Academic Year 2024-2025

A 9 ♣, sapevi già tutto

Abstract

This thesis explores the architectural evolution and verification methodologies associated with modern GPUs, with a particular focus on their role in high-performance and data-intensive applications. General purpose registers are integral to every digital design, thus ensuring reliable read and write functions is crucial. Transitioning from deprecated methodologies, like OVM, to contemporary methodologies, like UVM, aims to enhance efficacy, reduce manual effort, and improve scalability for future designs. The objective is to profile both current and previous methodologies to identify flow limitations and improve the register verification process through automation of design data capture and processing. This migration seeks to eliminate technical debt, ensure scalability, and increase overall efficiency.

The main focus is the verification of memory-mapped registers, crucial for hardware-software communication. GPUs serve as the case study due to their complexity and relevance in machine learning and graphics. The methodology explains UVMs modularity and reusable components, contrasts RGM (limited flexibility) with RAL, which provides an abstract, consistent register model supporting frontdoor and backdoor access.

Three strategies are compared: RGM sequencer, RAL frontdoor, and RAL backdoor, using Synopsys VCS for simulation and profiling. Results show RAL backdoor significantly reduces simulation time in frequent polling scenarios, though with slightly higher memory usage. RGM and RAL frontdoor offer better protocol realism but are slower. Introducing delays can improve synchronization.

RAL backdoor access is identified as a strong candidate for performance-sensitive verification tasks, especially when protocol overhead can be minimized. Future improvements may include adaptive polling strategies and dynamic access method selection, with potential extensions into power consumption analysis for a more comprehensive evaluation.

Contents

List of Figures

Acronyms

AI Artificial Intelligence

APB Advanced Peripheral Bus

API Application Programming Interface

AXI Advanced eXtensible Interface

BFM Bus Functional Model

CPU Central Processing Unit

DMA Direct Memory Access

DPI Direct Programming Interface

DUT Device Under Test

GPU Graphics Processing Unit

HBUS High-speed Bus

HDL Hardware Description Language

IP Intellectual Property

LSF Load Sharing Facility

OVM Open Verification Methodology

PLI Programming Language Interface

RAL Register Abstraction Layer

RGM Register and Memory Model

SSAA Super Sampling Anti-Aliasing

SV SystemVerilog

TLM Transaction Level Modeling

UART Universal Asynchronous Receiver-Transmitter

UVC Universal Verification Component

UVM Universal Verification Methodology

VC Verification Component

VCS Verilog Compiler Simulator

XML eXtensible Markup Language

Introduction

In todays rapidly evolving world of digital design, making sure hardware works correctly is very important. One key part of this is checking memory-mapped registers. These registers help software and hardware talk to each other, by controlling and report the status of different parts of a chip or System On Chip (SoC). Its important to make sure these registers can read and write data correctly, start with the right values, and follow access rules like read-only or write-only. This helps keep the system working well and safely.

This thesis looks at moving from older ways of checking registers, like OVM (Open Verification Methodology), to newer and more standard methods like UVM (Universal Verification Methodology). The goal is to make the process easier, faster, and more scalable by reducing manual work and avoiding outdated practices. By using automation to collect and process design data, the verification process becomes smoother and more efficient.

The thesis begins by describing how Graphics Processing Units (GPUs) have evolved over time. Initially designed to handle graphics tasks, GPUs have become powerful and flexible processors capable of performing many operations at once. This makes them suitable for areas such as machine learning, scientific computing, and real-time graphics. Due to their complexity and widespread use, it is essential to apply strong verification methods to ensure they function correctly. This relevance is further supported by the fact that the work presented in this thesis was carried out within the GPU verification team, providing direct exposure to the challenges and requirements of verifying such systems.

Next, the thesis explains how UVM works. UVM is built to be modular, scalable, and reusable. It uses standard parts like drivers, monitors, sequencers, and scoreboards that work together through clear interfaces. This setup makes it easier to build test environments that are flexible and easy to maintain, even for different types of designs.

One of the earlier methods used for register verification was RGM (Register and Memory Model). RGM was developed under the older OVM framework and later adapted for UVM. It introduced a way to model registers and memory using a centralized database and symbolic access. This helped automate consistency checks and made it easier to reuse components. However, RGM had limitations, it is harder to learn, lacked documentation, and didnt support some advanced UVM features like layered sequences or dynamic reconfiguration. As designs became more complex, RGM struggled with performance and integration, especially with third-party IPs. These challenges led to the development of more flexible and standardized solutions like RAL.

RAL gives a consistent way to model and access registers. It hides the technical details of how registers are built, so engineers can write tests that work across different designs. RAL supports both frontdoor access (through the bus) and backdoor access (directly through HDL), giving flexibility in how tests interact with the hardware.

To see how different methods work in practice, the thesis compares three ways to access registers: RGM sequencer, RAL frontdoor, and RAL backdoor. Each method uses a different level of abstraction and interacts with the hardware in its own way. The comparison looks at things like simulation time, memory use, and profiling overhead to help decide which method is best.

The tests use Synopsys VCS, a powerful tool for simulating and verifying hardware. VCS has profiling tools that help analyze how simulations run, including how long operations take and how much memory is used. By using a dedicated plus argument, the simulation collects data that helps find ways to improve performance and spot problems.

A key idea in the thesis is to keep memory profiling and time profiling separate. Mixing them can cause inaccurate results. Memory profiling adds extra work that can slow down timing, while time profiling needs to run smoothly to give correct performance data. Keeping them apart helps get better insights.

The results show that using RAL backdoor access can make simulations faster, especially when there are lots of register operations or polling. Since it skips the usual protocol steps and directly changes register values, it saves time. But it does use a bit more memory because of the extra layers involved.

On the other hand, RGM sequencer and RAL frontdoor methods give more realistic modeling of how the bus works. The sequencer uses less memory but takes longer to run. The frontdoor method balances realism and speed but can be slowed down by protocol rules.

The thesis also looks at how test setup affects results. For example, adding a delay before polling can help match the timing of hardware responses, reducing errors. These small changes show how important it is to design tests that reflect real-world conditions.

In the end, the thesis finds that RAL backdoor access is a good choice for tests that need to run fast and match real hardware behavior. It works best when protocol steps can be skipped and quick results are needed. But its success depends on how complex the test is, how polling is set up, and the environment it runs in.

Future improvements could include smarter polling and choosing access methods based on profiling during the test. These changes could make simulations even faster and more flexible. Also, adding power usage analysis could give a fuller picture of the trade-offs in hardware verification.

Chapter 1

Background

1.1 GPU

1.1.1 Introduction to GPU

In contemporary computer architecture, the Graphics Processing Unit (GPU) has emerged as one of the most critical hardware components. Originally developed to accelerate the rendering of complex 3D scenes in real-time applications such as video games, GPUs were designed to offload image generation tasks from the central processing unit (CPU). Their initial role was limited to converting binary data streams into visual output for display.

Over time, however, GPUs have undergone a significant transformation. They have evolved from fixed-function graphics accelerators into highly programmable, general-purpose parallel computing engines. This evolution has been driven by their architectural strengths, particularly their ability to perform high-throughput floating-point operations and vector arithmetic across large datasets[2].

Today, GPUs are not only indispensable in graphics rendering but are also widely used in domains such as big data analytics, scientific computing, machine learning, and artificial intelligence. Their massively parallel structure makes them exceptionally well-suited for executing algorithms that require simultaneous processing of large volumes of data.

At the core of a GPUs traditional role is the task of synthesizing images from scene descriptions at high frame rates, typically 60 frames per second or more. These scenes are composed of geometric primitives, lighting models, material properties, and camera perspectives. The computational demands of real-time rendering, particularly the need to process millions of vertices and pixels per second, have driven the

development of highly parallel GPU architectures. These architectures are capable of executing thousands of threads.

The graphics rendering pipeline is a structured sequence of stages that transforms 3D models into 2D images. It is typically divided into three key stages[3]:

- Vertex shading
- Rasterization
- Fragment shading

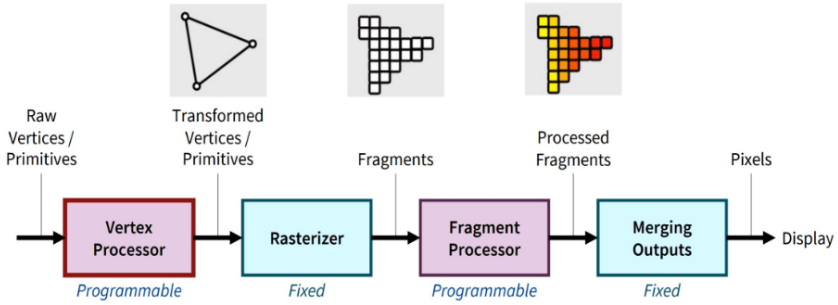

Figure 1.1: GPU stages

1.1.2 Vertex Shading

In the vertex shading stage of the graphics pipeline, the GPU processes the geometric data that defines the structure of a scene. Most real-time rendering systems operate under the assumption that all objects are composed of triangles. Consequently, any complex shape is first decomposed into a mesh of triangles to simplify processing and ensure compatibility with the pipeline[4].

Developers typically use computer graphics libraries to submit these triangles to the GPU one vertex at a time. The GPU then assembles the vertices into triangles as needed. Each triangle is defined by three vertices, and each vertex carries essential spatial information, its position in 3D space represented by coordinates (x, y, z). Additionally, vertices may include supplementary attributes such as color, texture coordinates, and surface normals, which are crucial for subsequent stages like lighting and shading.

To project these 3D vertices onto a 2D screen, the GPU applies a series of geometric transformations. These include translation, rotation, and scaling, all of

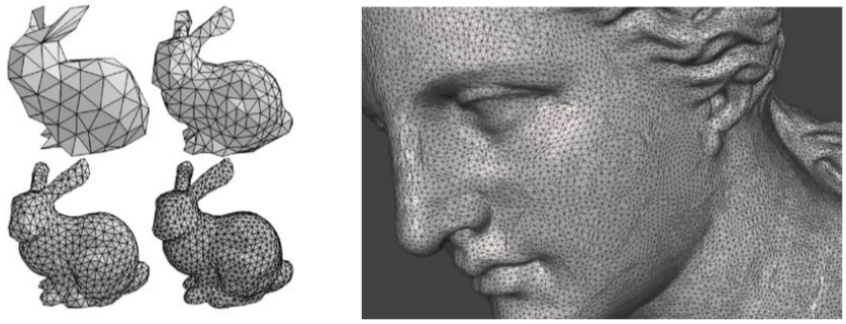

Figure 1.2: Vertex Shading Example

which are performed using transformation matrices. These matrices encapsulate both the object's local coordinate system and the camera's viewpoint. Through matrix-vector multiplication, the GPU efficiently transforms each vertex from object space to world space, then to camera space, and finally to screen space. The use of homogeneous coordinates enables these hierarchical transformations to be executed in a single, unified operation.

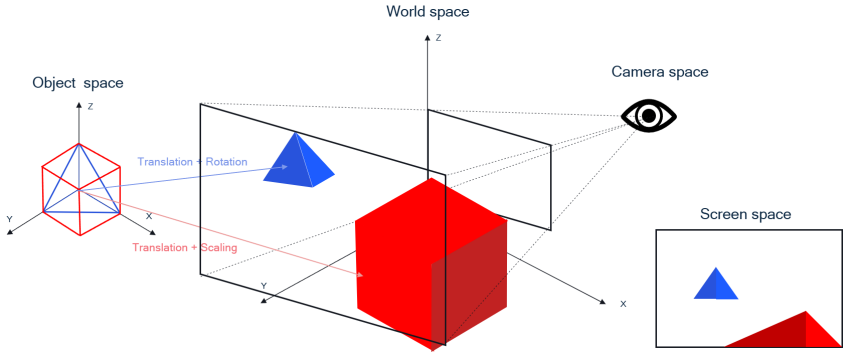

Figure 1.3: Object Transformations

The result of this transformation process is a set of screen-space coordinates and depth values (z-values) for each vertex. These values determine the position and orientation of each triangle on the display, ensuring that the scene is rendered from the correct perspective.

Once all triangles are transformed into a common coordinate system, typically one where the viewer is positioned at the origin and the viewing direction aligns with the z-axis, the GPU proceeds to compute lighting effects. The color of each triangle is determined based on a lighting model, which may involve multiple light sources. The GPU evaluates lighting equations using vector operations, such as dot products, to calculate the contribution of each light source to the final appearance

of the surface.

These lighting calculations are performed repeatedly and efficiently using hardware-accelerated multiply-add operations. This parallelism allows the GPU to handle complex lighting scenarios in real time, contributing to the realism and visual fidelity of modern graphics applications.

1.1.3 Rasterization

The rasterization stage is a fundamental component of the graphics rendering pipeline. Its primary function is to determine which pixels on the screen are covered by each triangle in a 3D scene. After a triangle has been transformed into screen space during the vertex shading stage, the GPU analyzes its vertex coordinates to identify the corresponding region on the display grid. This process converts geometric primitives into fragments, which are groups of pixels originating from the same triangle and sharing common attributes such as color, texture coordinates, and surface normals.

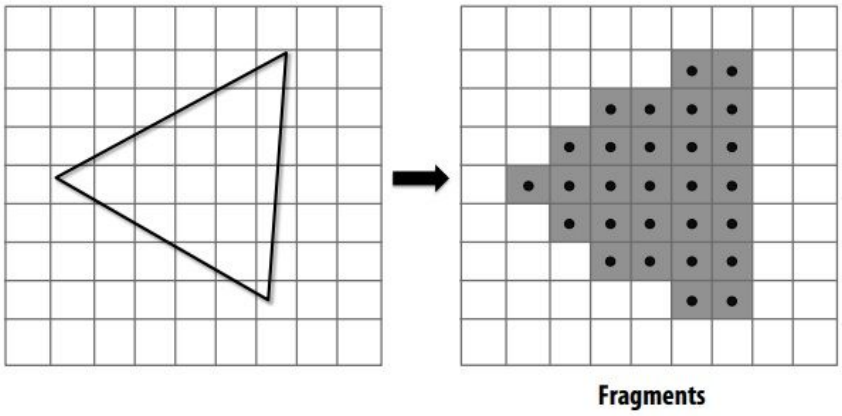

Figure 1.4: Rasterization Example

One of the key advantages of rasterization is the independence of pixel calculations. Each pixel can be processed in isolation from others, enabling highly parallel execution. This characteristic has driven GPU architectures toward increasingly parallel designs. Modern GPUs leverage this parallelism to evaluate millions of pixels simultaneously, significantly accelerating the rendering process and enabling real-time graphics performance [5].

To enhance visual realism, the color of each pixel is not solely derived from lighting calculations. Instead, textures, predefined images mapped onto surfaces,

are frequently applied to simulate fine surface detail. These textures are stored in high-speed memory, and the GPU accesses them during fragment shading to determine the final appearance of each pixel. Because adjacent pixels often access nearby texture locations, GPUs employ specialized cache mechanisms to reduce memory latency and maintain high throughput. In many cases, multiple texture samples are required per pixel to prevent visual artifacts, particularly when textures are displayed at resolutions different from their native size.

A critical challenge in rasterization is handling visibility when multiple triangles overlap. Simply writing pixel data in the order triangles are processed would result in incorrect layering, where the most recently processed triangle appears in front regardless of its actual position in 3D space. To address this, modern GPUs implement a depth-buffer or z-buiffer, which stores the distance from the viewer to the closest surface at each pixel. When a new fragment is generated, its depth value is compared to the existing value in the depth buffer. The pixel is updated only if the new fragment is closer to the camera, ensuring correct occlusion and depth perception.

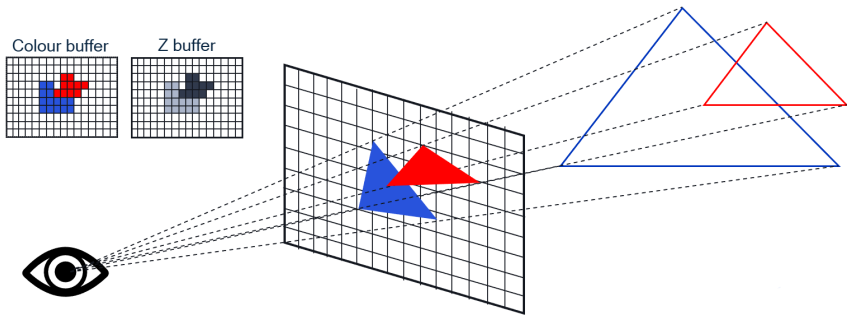

Figure 1.5: z-buffer

Because triangles exist in three-dimensional space, their vertices typically have distinct z-values. The GPU interpolates these values across the surface of the triangle to compute a depth value for each pixel, enabling accurate rendering of intersecting and overlapping geometry. This pixel-by-pixel depth comparison is essential for producing visually coherent scenes.

Despite its efficiency, rasterization is not without limitations. One common artifact is the appearance of jagged or pixelated edges, which occurs when a triangle intersects only a small portion of a pixel. In such cases, the entire pixel may be shaded with the triangles color, leading to harsh transitions. To mitigate this, techniques such as Super Sampling Antia-Aliasing (SSAA) are employed. SSAA subdivides each pixel into multiple sampling points, often 16 or more, and determines

the proportion of those points covered by the triangle. The final pixel color is then computed as a weighted average, resulting in smoother edges and a more visually appealing image.

Modern GPUs have evolved beyond fixed-function hardware, adopting a programmable architecture centered around unified shader cores. These programmable units, known as shaders, can perform a wide range of mathematical operations, including matrix-vector multiplication, exponentiation, and square root calculations. This flexibility enables efficient implementation of both traditional graphics tasks and general-purpose computations. As a result, GPUs have become powerful parallel computing platforms capable of supporting advanced rendering techniques and high-performance scientific applications alike.

1.1.4 Fragment Shading

The fragment shading stage represents the final and one of the most visually impactful phases of the graphics rendering pipeline. At this point, the GPU determines the final color and appearance of each pixel fragment that corresponds to a triangle projected onto the screen. While rasterization identifies which pixels are covered by each triangle, fragment shading is responsible for enriching those pixels with realistic lighting, texture, and material effects.

Assigning a uniform color to all fragments of a triangle may suffice for basic rendering, to achieve high visual fidelity, the GPU must simulate the complex interactions between light and surfaces. This involves evaluating how light behaves when it strikes different materials, how it reflects, refracts, or diffuses, and how these effects vary across the surface of an object.

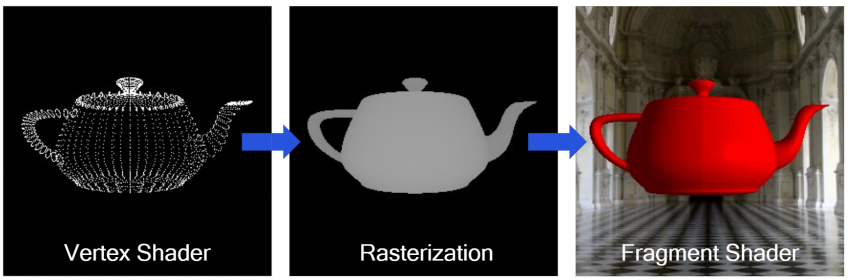

Figure 1.6: Fragment Shading Example

A foundational concept in illumination modeling is the dependency of surface brightness on the angle between the surface and the incoming light. Surfaces that

face directly toward a light source appear brighter, while those angled away receive less illumination. This behavior is mathematically governed by the dot product between the light direction vector and the surface normal vector. The surface normal, a vector perpendicular to the surface at a given point, plays a crucial role in determining how light interacts with that point.

In scenes with multiple light sources, the GPU computes the contribution of each light independently. These contributions are then summed to determine the total illumination at each fragment. This process allows for dynamic lighting effects, including ambient, diffuse, and specular components, which together create a more nuanced and realistic rendering of the scene.

A challenge in basic shading implementations is the use of a single normal vector per triangle, resulting in what is known as flat shading. In flat shading, the entire triangle is shaded uniformly, which can lead to abrupt transitions between adjacent triangles and a faceted appearance on curved surfaces. While computationally efficient, flat shading fails to capture the smooth transitions of light across curved surfaces, leading to unrealistic visual artifacts.

To overcome this limitation, modern shading techniques employ interpolated normals. These are derived by averaging the normals of adjacent triangles and assigning a unique normal to each vertex. During fragment shading, the GPU interpolates these vertex normals across the surface of the triangle, producing a smooth gradient of shading. This technique, known as Gouraud shading or Phong shading depending on the interpolation method, significantly enhances the perception of curvature and depth.

The interpolation process relies on barycentric coordinates, a mathematical framework for expressing any point within a triangle as a weighted combination of its vertices. For each pixel fragment, the GPU calculates its relative position within the triangle and uses the corresponding weights to interpolate attributes such as normals, texture coordinates, and colors. The resulting interpolated normal is then used in lighting equations to compute the final color of the fragment.

Beyond basic lighting, the fragment shading stage supports a wide array of advanced visual effects. Specular highlights simulate the bright spots of light that appear on shiny surfaces, while texture mapping applies detailed images to surfaces to mimic complex materials like wood, metal, or fabric. Shadowing techniques determine whether a fragment is occluded from a light source, adding depth and

realism to the scene.

Modern GPUs feature programmable shader cores that allow developers to write custom fragment shaders[6]. These shaders can implement physically based rendering models, simulate subsurface scattering, perform real-time reflections, and more. The flexibility of programmable shaders has transformed fragment shading from a fixed-function operation into a rich domain for artistic and scientific innovation.

1.1.5 Qualcomm's GPU Adreno

Qualcomm has played a pivotal role in shaping the mobile GPU landscape through its Adreno series, which has become a cornerstone of its Snapdragon processors. The history of Qualcomm's GPU technology is marked by continuous innovation, performance enhancements, and architectural refinements aimed at delivering high-efficiency graphics processing for mobile devices.

The Adreno GPU lineage traces back to ATI Technologies, which originally developed the Imageon graphics processors for mobile devices. After AMD acquired ATI, the Imageon division was sold to Qualcomm in 2009, marking the inception of the Adreno brand, an anagram of Radeon [7]. Qualcomm integrated Adreno GPUs into its Snapdragon SoCs, enabling a unified platform for mobile computing and graphics.

Over the years, Qualcomm has introduced several architectural improvements to the Adreno GPU. These enhancements include increased shader cores, a specialized processing units within a GPU, better memory bandwidth utilization, and support for advanced graphics APIs such as Vulkan and OpenGL ES. Each generation of Adreno has brought significant gains in rendering performance and power efficiency, crucial for mobile gaming and augmented reality applications [8].

The performance trajectory of Adreno GPUs has been impressive. According to Forbes, Qualcomm's mobile GPU innovations have consistently pushed the boundaries of mobile gaming, enabling console-quality graphics on smartphones [1]. The integration of heterogeneous computing elements and AI acceleration has further expanded the capabilities of Adreno GPUs, making them suitable for complex tasks like real-time image processing and neural network inference.

Qualcomm's Snapdragon platform, which houses the Adreno GPU, has become synonymous with high-performance mobile gaming. The company's whitepaper on

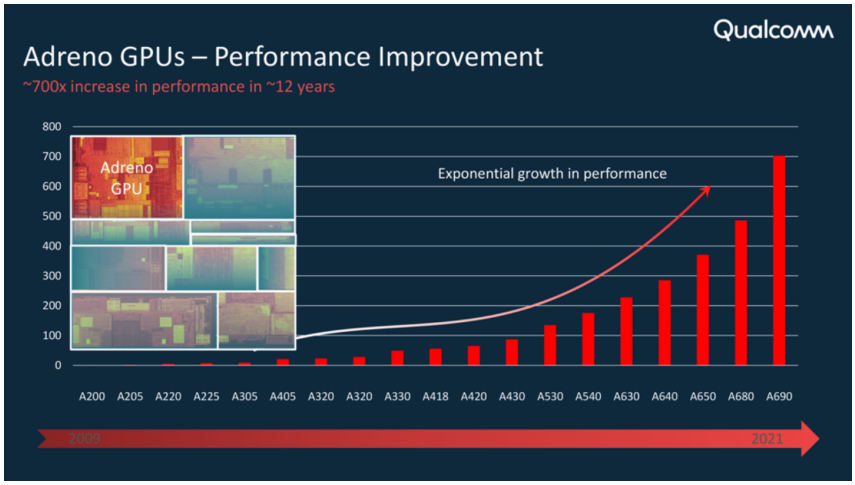

Figure 1.7: Performance evolution of Qualcomm's GPU Architecture over time [1]

mobile gaming highlights how Snapdragon processors deliver superior frame rates, reduced latency, and immersive experiences through optimized GPU performance [9].

1.2 UVM

1.2.1 Background and Evolution of the Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) is a widely accepted framework used in the semiconductor industry to verify digital designs. It provides a structured and reusable way to build test environments, used to ensure that the hardware behaves as expected before manufacturing.

UVM is designed to be flexible and scalable. This means it can be used for small modules, such as a simple logic block, or for large and complex SoCs that include many interconnected components. Regardless of the size or complexity of the design, UVM offers tools and guidelines that help verification teams manage their work efficiently.

UVM is built upon SystemVerilog classes, designed to facilitate the development of robust and reusable testbenches for verifying complex digital designs. It leverages key verification techniques such as object-oriented programming, constrained random stimulus generation, functional coverage, and assertion-based validation to ensure comprehensive and scalable verification environments.

At its core, UVM provides a rich set of libraries, base classes, and guidelines that enable to construct modular and portable testbenches. These components are designed to promote consistency and reusability across projects, thereby reducing development time and improving verification quality.

It originated from the Open Verification Methodology (OVM), which served as the foundational blueprint for its architecture. The initial release, UVM 1.0EA (Early Adopter), launched in 2010, was largely a direct port of OVM, incorporating only minor modifications and enhancements [10]. This early version laid the groundwork for what would become a rapidly evolving methodology.

As UVM matured, it began to integrate proven concepts from other methodologies. One of the most significant additions was the Register Abstraction Layer (RAL), which brought a standardized approach to modeling and verifying memory-mapped registers. UVM also absorbed best practices from both formal documentation and ad hoc verification strategies developed across the industry.

Over time, UVM has continued to evolve, introducing new paradigms for testbench

architecture and test development. While the methodology has embraced modern techniques and tools, it has also maintained backward compatibility with earlier practices. This dual-layered evolution, preserving legacy support while advancing new capabilities, has allowed UVM to remain both robust and adaptable.

1.2.2 Testbench components

A UVM testbench is composed of a set of components, each fulfilling a distinct role in the functional verification of a digital design. These components are instantiated and interconnected using standardized base classes provided by the UVM library. Figure 1.8, illustrates the interconnection of these components within a typical UVM testbench architecture.

At the highest level, the UVM testbench is organized hierarchically, beginning with the UVM Test, which serves as the top-level entity. This component is responsible for configuring the testbench, initiating simulation phases, and defining the test scenario. It acts as the entry point for the verification process and typically includes logic to select and configure the environment, sequences, and DUT interfaces.

Beneath the test layer lies the Environment (env), which acts as a container for all other verification components. The environment encapsulates agents, scoreboards, coverage collectors, and other subcomponents. It provides a structured context in which these elements can be instantiated and connected. By centralizing control and configuration, the environment ensures that all components operate cohesively and according to the test plan.

Within the environment, Agents are responsible for managing the verification of specific interfaces or subsystems. Each agent typically includes three subcomponents: the Driver, the Monitor, and the Sequence [11]. These elements work together to generate stimulus, observe DUT behavior, and coordinate test execution.

The Driver plays a critical role in translating high-level transaction objects into low-level signal activity. It interfaces directly with the DUT, applying stimulus in accordance with the protocol specifications. The driver ensures that the DUT receives valid and meaningful input, and it is often protocol-aware, meaning it understands the timing and sequencing requirements of the interface it drives.

Complementing the driver is the Monitor, which passively observes the signals on the DUT interface. The monitor reconstructs transactions from observed signal

activity and forwards them to other components such as the scoreboard or coverage collector. This passive observation is essential for non-intrusive verification, allowing the testbench to validate DUT behavior without affecting its operation.

The Sequencer is responsible for generating and randomizing sequences of transactions. It coordinates with the driver to deliver stimulus to the DUT in a controlled and repeatable manner. Sequencers can be programmed to produce directed tests, constrained-random scenarios, or corner-case conditions, to achieve coverage goals.

To validate the correctness of DUT behavior, the testbench includes a Scoreboard. This component compares the actual output of the DUT with expected results, which may be generated by a reference model or derived from the input stimulus. The scoreboard performs checks on data integrity, protocol compliance, and functional correctness. It is a key component in determining whether the DUT meets its specification.

Communication between the testbench and the DUT is facilitated through Interfaces, which encapsulate the physical and logical connections. Interfaces define the signal declarations, timing constraints, and protocol-specific logic required for interaction. They provide a clean abstraction layer that separates the testbench from the implementation details of the DUT.

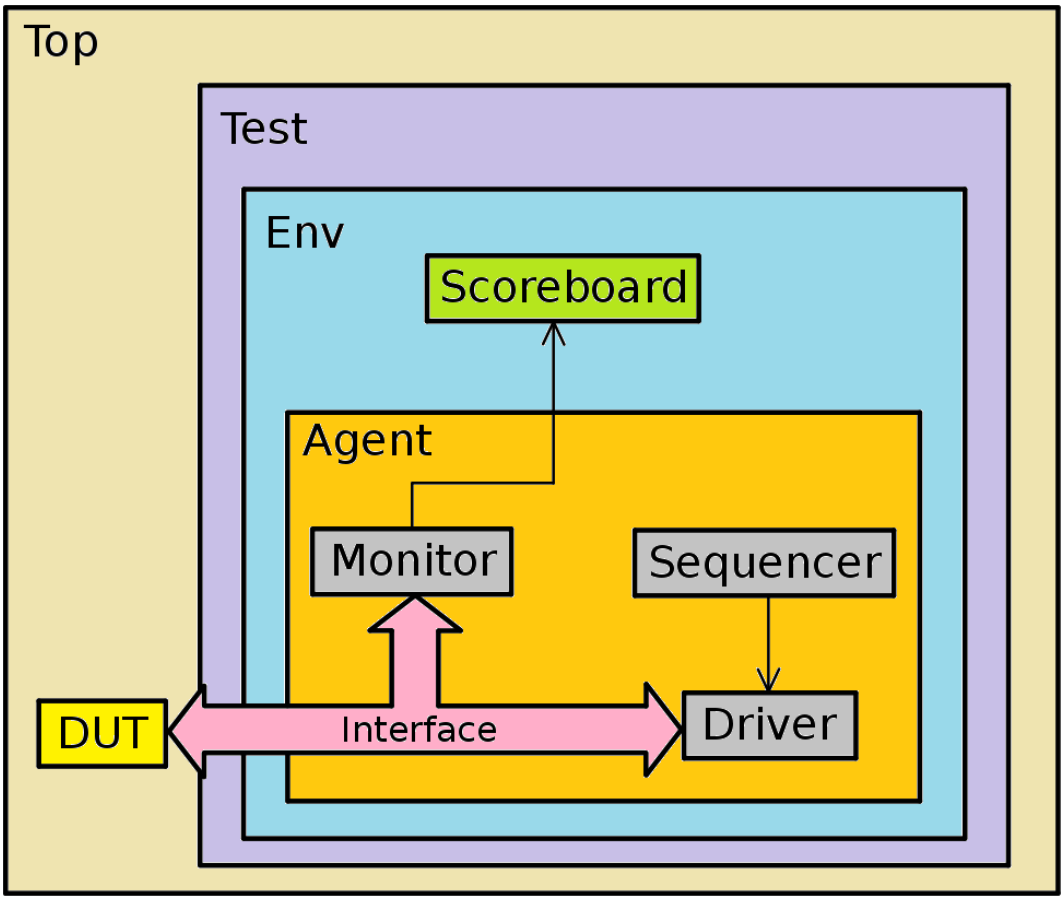

Figure 1.8: Overview of UVM Components

1.2.3 UVM Structural Organization and Phasing Mechanism

In UVM, the construction and execution of a testbench follow a well-defined structure. The components are typically declared in a bottom-up hierarchy. This means that lower-level classes such as sequencer, driver, and monitor are defined first. These are then grouped into an agent, which is subsequently instantiated within the top-level verification component (VC).

Each component is instantiated inside its parents constructor. For example, the agent creates instances of the sequencer, driver, and monitor, while the VC creates an instance of the agent. This bottom-up declaration ensures that compilation dependencies are resolved correctly and that the testbench is built in a modular and maintainable fashion.

To support hierarchical navigation and debugging, UVM components include two important properties:

- Instance Name: A string identifier that typically matches the handle name of the component. This helps in identifying components during simulation and reporting.
- Parent Pointer: A reference to the component that instantiated the current component. This allows traversal of the component hierarchy from any point in the testbench.

These properties enable components to generate their full hierarchical path, which is particularly useful for logging, diagnostics, and error tracing.

Beyond structural organization, each component follows a structured sequence of simulation phases. This sequence is governed by the UVM phasing mechanism, which ensures that all components in the testbench operate in a synchronized and predictable manner throughout the simulation lifecycle[12].

The primary phases include:

- Build Phase: Components are instantiated and configured. This phase sets up the testbench structure, including the creation of agents, drivers, monitors, and sequencers.
- Connect Phase: Hierarchical and functional connections between components are established. Drivers are linked to sequencers, monitors are connected to scoreboards and so on.

- Run Phase: The actual test scenario is executed. Stimulus is generated, transactions are driven to the Design Under Test (DUT), and responses are monitored and analyzed.
- Report Phase: Results are summarized. This may include printing statistics, coverage data, and pass/fail status.

In addition to these core phases, UVM supports optional phases such as pre-run, post-run, pre-reset, and post-reset, which allow for custom behavior to be inserted at specific points in the simulation. These phases are particularly useful for initializing data structures before execution or performing cleanup and analysis afterward.

The phasing mechanism is essential for maintaining order and consistency across all components. It provides a timeline for simulation activities and supports modular development, easier debugging, and enhanced reusability. Furthermore, UVM allows users to define custom phases, enabling tailored simulation flows that meet specific project requirements.

By adhering to this phased approach, UVM ensures that verification environments are not only organized but also scalable and adaptable to a wide range of design complexities.

1.2.4 Reusability in UVM

One of the most powerful features of the UVM is its emphasis on reusability. In modern digital design verification, where projects often span multiple teams, technologies, and timelines, the ability to reuse components across different environments is essential for maintaining efficiency, consistency, and scalability. UVM promotes reusability through its object-oriented architecture and modular design principles [13].

Components such as agents, drivers, monitors, and scoreboards can be reused across multiple projects with minimal modification, significantly reducing development time and effort.

To maximize the reusability of UVM components, it is essential to follow a set of design principles that promote modularity and flexibility. One important practice is the use of virtual interfaces instead of hardwired connections. By declaring interface variables as virtual and assigning them dynamically, components can be reused across different interface instances without modification. This approach decouples the component from specific interface names and allows for greater adaptability in

various testbench configurations.

Another key strategy is to implement shared functionality in a base class. Common properties and methods can be defined once in this base class and inherited by derived components, ensuring consistency and reducing code duplication. This not only simplifies maintenance but also enhances the clarity of the verification architecture.

Furthermore, matching instance names with their corresponding handle names improves readability and facilitates hierarchical navigation, especially during debugging and reporting. Avoiding assumptions about the depth or structure of the component hierarchy also contributes to portability, allowing components to be integrated into different environments without requiring structural changes.

Together, these practices enable to build robust, scalable, and reusable testbenches that can be efficiently adapted to new designs and reused across multiple projects.

UVM provides several built-in mechanisms that further enhance reusability:

- Factory Pattern: The UVM factory allows components to be created dynamically at runtime [14]. This enables testbenches to override default implementations with custom versions without modifying the original source code. The factory pattern supports polymorphism and late binding, which are key principles of object-oriented design.

- Configuration Database: The configuration database provides a centralized way to pass parameters and settings between components. This avoids the need for deep constructor argument lists and allows components to be configured externally, improving modularity and reuse.

- Parameterization: UVM components can be parameterized to support different data types, protocols, or configurations. This allows a single component definition to be reused in multiple contexts by simply changing the parameter values.

- Scalability Through Reuse: Reusability in UVM also contributes to scalability. As designs grow in complexity, verification environments must scale accordingly. UVM supports this through hierarchical composition and modular design. For example, in a typical SoC verification project, the testbench may include multiple agents for different interfaces such as AXI, APB, and UART. Each agent contains its own driver, monitor, and sequencer tailored to the specific

protocol. The top-level environment coordinates these agents, collects coverage data, and manages interactions between subsystems. Scoreboards validate data transfers across interfaces, ensuring that the DUT behaves correctly under various conditions. Because each component is modular and reusable, the testbench can be extended or modified without disrupting the overall architecture.

Chapter 2

Case Study: Register Verification

2.1 Introduction

Register verification is a foundational component of digital design verification, especially in systems where software interacts with hardware through memory-mapped registers [15]. These registers serve as control and status interfaces for various subsystems within a chip or SoC, enabling configuration, monitoring, and command execution.

The verification process ensures that each register behaves according to its specification, responding correctly to read and write operations, initializing with the correct reset values, and enforcing access permissions such as read-only or write-only constraints. It also validates that the register map aligns with the design documentation and that interactions with the registers do not introduce unintended side effects.

Effective register verification involves modeling the register architecture, generating stimulus to exercise different access patterns, and checking the correctness of responses. This includes verifying address decoding, data integrity, and error handling mechanisms.

Coverage metrics are often employed to assess how thoroughly the register space has been tested[16], ensuring that all fields, access types, and edge cases are accounted for. As designs grow in complexity, the need for scalable, reusable, and automated register verification methodologies becomes increasingly critical.

2.2 Common Register Verification Techniques

To ensure comprehensive validation of register behavior, a diverse set of targeted techniques must be systematically applied throughout the verification process. These techniques are essential for identifying and mitigating potential issues that may compromise the reliability, security, and correctness.

The following sections present a selection of register verification techniques commonly employed in hardware design and validation. These techniques are integral to ensuring that digital systems conform to their functional, performance, and reliability specifications, and are supported by both simulation-based and formal verification methodologies.

The write-then-read test serves as the foundational method for register validation. This technique involves writing a predetermined value to a register and immediately reading it back to confirm that the value has been accurately stored and retrieved. It is primarily used to verify the correctness of address decoding, the integrity of the data path, and the enforcement of read/write access permissions. Additionally, this method helps detect issues such as bus contention, incorrect register mapping, and unintended data corruption. It is often employed as an initial smoke test during bring-up phases and is integrated into automated regression suites to ensure consistent behavior across design iterations.

Bitbashing is a granular and exhaustive technique that targets individual bits within a register. It involves writing and reading back a series of carefully selected bit patterns to evaluate the behavior of each bit independently. Common patterns include all zeros (`0x0000...`), all ones (`0xFFFF...`), and alternating bits such as (`0xAAAA...`, `0x5555...`). These patterns are instrumental in detecting stuck-at faults, which are hardware defects where a signal line remains permanently fixed at a logical high or low state. Such faults may arise due to fabrication anomalies, layout parasitics, or signal integrity degradation. Bitbashing also helps identify coupling effects between adjacent bits, unintended toggling due to clock domain crossings, and violations of bit-level isolation [17]. It is particularly effective in designs with high-density register banks and mixed-signal interfaces.

Walking ones and walking zeros tests provide a systematic approach to validating bit isolation and register width correctness. These tests involve writing a single logical 1 or 0 to each bit position in a register sequentially and reading back the value to confirm accuracy. For example, walking ones may include patterns such as `0x0001`, `0x0010`, `0x0100`, and so forth, while walking zeros may include `0xFFFE`,

`0xFFFD`, `0xFFFB`, etc. These patterns help confirm that each bit can be independently controlled and observed, and that no unintended interactions occur due to bit masking, shifting, or encoding logic. This technique is particularly valuable in verifying the correctness of register width declarations and ensuring that no bits are inadvertently omitted, aliased, or misaligned. It also aids in detecting synthesis or optimization artifacts that may alter bit-level behavior.

Access permission checks are essential for enforcing the intended access restrictions of registers. Registers may be configured as read-only, write-only, or read-write, and these configurations must be rigorously validated. The verification process includes attempts to perform illegal accesses, such as writing to a read-only register or reading from a write-only register, and confirming that such operations are appropriately blocked or ignored. Additionally, legal accesses must be verified to ensure they produce the expected results and side effects, such as status flag updates, interrupt triggers, or configuration changes. These checks are critical for maintaining system security, preventing unauthorized modifications, and ensuring predictable behavior in multi-threaded or multi-core environments.

Reset value verification is a fundamental aspect of design initialization. Upon hardware reset, registers are expected to initialize to known default values as specified in the design documentation. This test confirms that the reset mechanism functions correctly, that no residual data from previous operations persists, and that the register values align with the expected post-reset state. This verification is particularly important in systems with power-on-reset sequences, where timing and sequencing of reset signals can affect the initialization behavior. It also helps detect issues related to asynchronous resets, metastability, and improper reset propagation across hierarchical modules.

Illegal access testing extends the scope of validation to include out-of-spec operations [18]. These operations may involve accessing undefined addresses, writing to reserved fields, or performing operations that violate protocol specifications. The objective of this testing is to confirm that the design handles such cases gracefully, either by ignoring the operation, triggering appropriate error responses, or entering a safe recovery state. This form of testing is crucial for ensuring robustness and fault tolerance in safety-critical and mission-critical applications. It also validates the implementation of error reporting mechanisms such as exception flags, error codes, and system logs.

Field-level testing focuses on the validation of individual fields within a register,

especially when registers contain multiple fields with distinct access types, update mechanisms, or encoding schemes. This method involves verifying that each field can be accessed independently, that field-level granularity is maintained, and that encoding and decoding of field values are performed correctly. Field-level testing is particularly important in control registers where multiple configuration options are packed into a single register, and incorrect field interactions can lead to unpredictable system behavior. It also ensures that field boundaries are respected, that reserved bits remain unaffected, and that field-specific side effects are correctly triggered.

Coverage-driven verification complements the aforementioned techniques by providing a quantitative measure of the verification effort. This approach involves tracking which registers, fields, and access types have been exercised during simulation. Functional coverage models are employed to ensure that all relevant scenarios, including edge cases and corner conditions, are adequately tested. These models may include cross-coverage between fields, temporal coverage of access sequences, and conditional coverage based on system states. Coverage analysis helps identify gaps in the verification plan and guides the development of additional test cases to achieve comprehensive validation. It also supports metrics-based reporting for verification closure and compliance audits.

2.3 RGM vs RAL

The evolution of register verification methodologies has mirrored the broader shift in hardware verification from ad hoc scripting to standardized, reusable frameworks. Early approaches relied heavily on manual testbench coding and protocol-specific sequences, which were difficult to scale and maintain. As verification complexity increased, the industry began adopting abstraction layers and reusable components to streamline the process.

One of the earliest structured solutions was the Register and Memory Package (RGM), developed for use with the Open Verification Methodology (OVM)[19]. RGM introduced a centralized database and basic modeling constructs that allowed to simulate register behavior and automate access sequences. While innovative for its time, RGM lacked standardization and flexibility, prompting the need for a more robust solution.

This led to the development and adoption of the UVM, which formalized register modeling through a standardized framework and introduced the Register Abstraction

Layer (RAL) [20]. RAL built upon the concepts pioneered by RGM but added significant enhancements in modularity, coverage integration, and runtime configurability.

While both RGM and RAL aim to simplify and structure register verification, they differ significantly in design philosophy, capabilities, and industry adoption. RGM, as a legacy solution, laid the groundwork for abstraction but is now considered limited in scope and flexibility. RAL, on the other hand, is part of the UVM standard and offers a comprehensive, scalable, and reusable approach to register modeling.

In the following sections, we will explore each methodology in detail examining their architecture, features, and practical implications before presenting a comparative analysis to highlight their strengths and limitations.

2.4 RGM

2.4.1 Introduction

In the realm of digital design verification, the accurate modeling and validation of register and memory behavior is a cornerstone of functional correctness. The Register and Memory Model, commonly referred to as RGM, was conceived to address this need by providing a high-level abstraction layer that simplifies the interaction with these critical components. Initially developed under the Open Verification Methodology (OVM) and later refined for the Universal Verification Methodology (UVM), RGM has become a foundational element in modern verification environments [21].

The UVM implementation of this model, known as `uvm_rgm`, offers a comprehensive framework for modeling, accessing, and verifying registers and memory structures within a design [22]. It enables to define register maps and memory blocks in a structured manner, allowing for symbolic access, automated consistency checks, and seamless integration with testbenches. This abstraction not only enhances readability and maintainability but also facilitates reuse across projects and teams.

One of the key strengths of `uvm_rgm` lies in its ability to mirror the internal state of the DUT[23]. By maintaining a synchronized model of the register and memory contents, it becomes possible to perform high-level operations such as randomized configuration, functional coverage collection, and assertion-based validation. Moreover, the framework supports both address-based and name-based access, which significantly improves the clarity and intent of test scenarios.

During the verification process, `uvm_rgm` initializes the DUT, injects randomized values, and monitors register activity throughout simulation. This includes not only performing read and write operations but also validating the DUTs behavior against expected outcomes and reference models. The ability to trace and debug register interactions at this level of abstraction is invaluable, particularly in complex systems where manual tracking would be error-prone and inefficient.

Furthermore, `uvm_rgm` provides a methodology that encourages automation and modularity. It supports the definition of hierarchical address maps, register files, and memory arrays, all of which can be accessed through reusable sequences. This modularity is essential for scaling verification efforts across large SoCs, where consistency and repeatability are paramount [23].

However, despite its many advantages, RGM is not without its limitations. One of the most prominent challenges lies in its steep learning curve, mastering RGM can be a daunting task. The framework introduces a rich set of concepts, abstractions, and APIs that require a solid understanding of both UVM principles and register modeling practices. Unfortunately, the availability of comprehensive learning resources is limited. Unlike other widely adopted verification tools, RGM lacks extensive documentation, tutorials, and community-driven support.

Moreover, as designs scale in complexity, RGM may encounter performance bottlenecks, particularly when managing large and frequently changing register maps. The regeneration and validation of models in such environments can become a significant maintenance burden. Integration with third-party IPs also presents difficulties, especially when those components follow different modeling conventions or lack compatibility with the RGM infrastructure. Additionally, while RGM aligns well with standard UVM components, it does not natively support some of the more advanced features of UVM, such as layered sequences or dynamic reconfiguration, which may limit its flexibility in sophisticated verification environments.

2.4.2 IP-XACT

The process of defining register and memory models within the RGM framework begins with the construction of a `uvm_rgm` model. This model serves as the backbone of register-level verification and encapsulates several key components, including register fields, register files, address maps, and the central register database. These elements collectively form a hierarchical and structured representation of the memory-

mapped architecture of the design.

To facilitate this modeling process, designers typically rely on IP-XACT, a standardized XML-based format developed to describe and integrate design intellectual property (IP) [24]. IP-XACT provides a formal and machine-readable way to capture the metadata associated with registers and memories, such as their addresses, access types, reset values, and structural relationships. This format has become a cornerstone in the automation of register modeling, enabling tools to generate consistent and reusable verification components directly from specification files.

An IP-XACT file organizes information into arrays of memory maps, each representing a distinct address space within the design. Within these maps, designers can define memory blocks, register files, and individual registers, along with their constituent fields. Each field can be annotated with attributes that govern its behavior, such as read/write permissions, default values, and volatility [25]. This level of granularity ensures that the verification environment accurately reflects the intended functionality of the hardware.

Figure 2.1 illustrates a typical IP-XACT description of a memory map. The hierarchical structure shown in the XML snippet highlights how register files are composed of registers, which in turn contain fields. This organization mirrors the physical layout of the design and allows to interact with the model in a logical and intuitive manner.

The adoption of IP-XACT within the RGM framework not only streamlines the creation of register models but also enhances consistency across teams and projects. By abstracting the register definitions into a standardized format, it becomes possible to automate the generation of UVM components, reduce manual errors, and ensure alignment between design and verification. This methodology is particularly valuable in large-scale systems, where the complexity and volume of registers can quickly become unmanageable without a structured approach.

2.4.3 RGM_DB

Once the IP-XACT register description file has been authored, a dedicated parser utility is employed to translate its contents into a set of SystemVerilog (SV) classes tailored for register verification[26]. This automated conversion process extracts the structural and behavioral information encoded in XML, such as register fields, address maps, and access attributes and instantiates corresponding SV objects that

```
- <spirit:memoryMap>
    <spirit:name>memory_map_name</spirit:name>
- <spirit:addressBlock>
    <spirit:name>regfile_name</spirit:name>
    <spirit:baseAddress>0</spirit:baseAddress>
    <spirit:range>16</spirit:range>
    <spirit:width>8</spirit:width>
- <spirit:register>
    <spirit:name>register_name</spirit:name>
    <spirit:addressOffset>0x0</spirit:addressOffset>
    <spirit:size>8</spirit:size>
    <spirit:access>read-write</spirit:access>
- <spirit:reset>
    <spirit:value>0x00</spirit:value>
    <spirit:mask>0xff</spirit:mask>
  </spirit:reset>
- <spirit:field>
    <spirit:name>field_name</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>2</spirit:bitWidth>
    <spirit:access>read-write</spirit:access>
- <spirit:values>
    <spirit:value>0</spirit:value>
    <spirit:name>LEFT</spirit:name>
    <spirit:description>0</spirit:description>
  /spirit:values>
- <spirit:values>
```

Figure 2.1: IP-XACT description of a register file hierarchy used in RGM

can be integrated into the testbench environment.

Once the register model has been automatically generated from the IP-XACT description, the next critical phase involves its integration into the UVM testbench environment. This step is essential for enabling structured and scalable register-level verification. At the heart of this framework lies the centralized register database, known as `RGM_DB`. This database plays a pivotal role in maintaining shadow copies of the DUTs registers, effectively mirroring their state throughout the simulation. By doing so, it allows the verification environment to perform consistency checks, detect mismatches, and ensure that register accesses conform to the expected behavior defined in the model.

The database is used for coverage collection and functional analysis. By monitoring register activity, such as which registers have been accessed, how often, and under what conditions, `RGM_DB` contributes to the generation of meaningful coverage metrics.

These metrics help assess the thoroughness of the verification effort and guide the refinement of test scenarios to ensure complete functional exploration.

The next step is to integrate the necessary components into the UVM testbench environment. These components include the register database (RGM_DB), the register sequencer, and the interface bus Universal Verification Component (UVC), all provided as part of the uvm_rgm package. The integration process involves the following steps:

- Instantiate the Register Database (RGM_DB): Place the RGM_DB component within the testbench to serve as the central repository for register and memory definitions. This database will be referenced by other components during simulation to access register metadata and perform verification tasks.
- Extend the Interface Bus UVC Master Sequencer: Modify the master sequencer of the interface bus UVC to include the necessary infrastructure for register transactions. This involves adding support for adapter sequences that translate high-level register operations into protocol-specific transactions.
- Create an Adapter Sequence: Develop a custom adapter sequence that interfaces with the modified master sequencer. This sequence acts as a bridge between the register model and the bus protocol, ensuring that register read/write operations are correctly interpreted and executed.
- Instantiate and Connect Register Components: Instantiate the register model components within the testbench and establish Transaction-Level Modeling (TLM) connections between them. This includes linking the register sequencer to the adapter sequence and connecting the register model to the RGM_DB and interface bus components.

By following these steps, the testbench is equipped to perform structured and automated register verification. This setup enables consistent access to register definitions, supports protocol-specific transactions, and facilitates scalable verification across different design configurations

Figure 2.2 for an overview of the connection between DUT and shadow register.

During typical register-level activity in a verification environment, several types of interactions with the device under test (DUT) are commonly required. These operations include:

- Initial Configuration: Driving the DUT into a valid operational state by writing to control registers that enable specific modes or features.

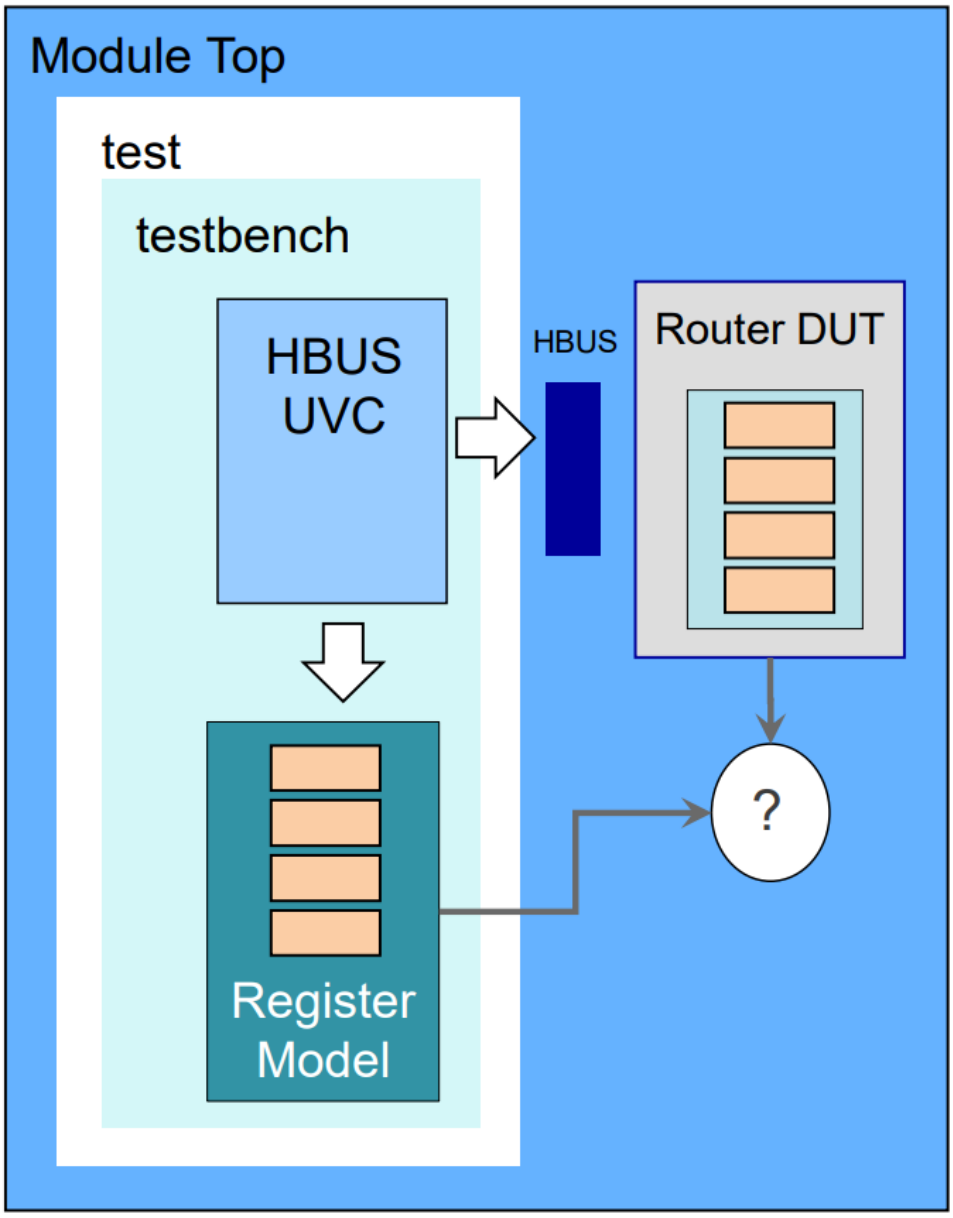

Figure 2.2: DUT and shadow register

- Dynamic Reconfiguration: Modifying the device configuration multiple times during a simulation run. For example, a Direct Memory Access (DMA) controller may be reconfigured to perform different tasks sequentially within the same test scenario.

- Runtime Transactions: Performing standard read and write operations to transfer values between registers and memory locations, often as part of functional stimulus or data flow validation.

To facilitate these tasks, it's typically preferred to access registers using symbolic names or addresses, which improves readability and maintainability of the verification code. The `uvm_rgm` framework supports this by leveraging the standard UVM sequence mechanism. Sequences can be defined to perform register read and write operations using the `uvm_rgm_register_operation` class. This abstraction allows for flexible modeling of both directed and randomized scenarios, enabling comprehensive coverage of register behavior.

By encapsulating register transactions within reusable sequences, the methodology promotes modularity and scalability. It also simplifies the integration of register-level stimulus into larger verification environments, ensuring that register interactions are consistent, traceable, and aligned with the overall testbench architecture.

An example of a uvm_reg architecture is shown in the Figure 2.3, it highlights the modular structure recommended by UVM, where monitoring components are decoupled from the randomization and stimulus generation logic. This separation of concerns ensures that coverage collection and functional checking are performed independently of the mechanisms responsible for injecting transactions into the DUT. Specifically, the monitoring infrastructure observes register activities during simulation and triggers coverage and consistency checks based on these observed events. This architectural approach enhances the clarity, reusability, and maintainability of the verification environment, while also supporting more accurate and comprehensive validation of register behavior.

As illustrated in the diagram, `RGM_DB` serves as the repository for all register-related metadata and runtime state information. It encapsulates key modeling constructs such as `uvm_rgm_address_map`, `uvm_rgm_register_file`, and `uvm_rgm_memory`, each of which contributes to a hierarchical and comprehensive representation of the DUTs memory-mapped architecture.

2.4.4 Modeling and Validation of DUT Registers Using the RGM Framework

The Register and Memory Database (RGM_DB) serves as the central component of the Register and Memory (RGM) framework used in UVM-based verification environments. It functions as the core data structure that encapsulates the entire shadow model of the Design Under Test (DUT), effectively mirroring the state of the DUTs registers and memory. Once the memory and register models are defined, they are instantiated and stored within the RGM_DB, which acts as the root of the register model hierarchy. When a register is written or read via the bus interface,

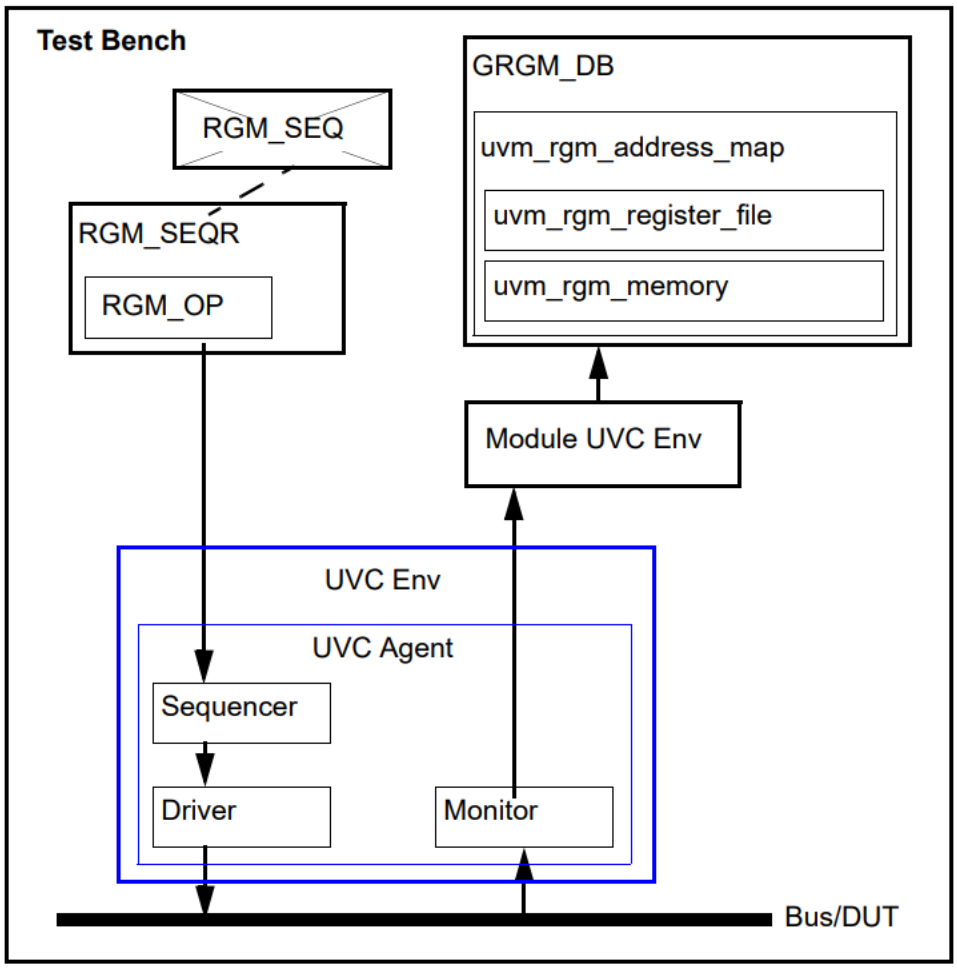

Figure 2.3: uvm_rgm architecture

the RGM_DB is updated accordingly. This shadow model enables:

- Scoreboarding: Comparing DUT responses against expected values.
- Coverage: Tracking which registers and fields were accessed.

As a subclass of the uvm_component base class, the RGM_DB inherits all standard UVM functionalities. These include integration with the configuration database mechanism, participation in the UVM testbench phase flow, and control over messaging and reporting. By centralizing the register and memory models, the RGM_DB ensures consistent access, configuration, and monitoring across the verification environment. This architectural approach not only enhances modularity and reusability but also supports scalable and maintainable register-level verification.

Scoreboards act as reference models that track and validate the behavior of the DUT by comparing expected outcomes, derived from the testbench's internal models, with the actual results observed during simulation. This comparison is essential for

identifying discrepancies and ensuring that the DUT adheres to its specification.

Let's see how this is performed in details. The following steps outline the verification plan:

1. Reference Model Development: Construct a comprehensive reference model that accurately represents the register map. This model should include details such as register addresses, sizes, reset values, and access policies. Additionally, it must incorporate memory block representations to facilitate validation of basic memory access operations.

2. Stimulus Generation via HBUS UVC: Utilize the HBUS Universal Verification Component (UVC) to generate read and write transactions targeting the router registers. Existing sequences from the HBUS UVC library may be reused or adapted to suit the specific requirements of the register.

3. Monitor Integration and Shadow Register Updates: Connect the reference model to the analysis port of the HBUS UVC monitor to capture and interpret HBUS transactions. The model should update shadow registers in accordance with the register's functional specification.

4. Register Value Comparison: After each stimulus sequence, compare the values of the registers in the DUT against those in the reference model to ensure consistency and correctness.

Within the RGM framework, this validation process is tightly coupled with the register and memory models. When a write operation is performed on a register or memory location, the RGM updates its internal shadow model to reflect the new state. Conversely, when a read operation is detected on the bus, the RGM accesses its internal model and compares the expected value with the data returned by the DUT. This mechanism ensures that any deviation from the expected behavior is promptly flagged and can be analyzed further.

The scoreboard itself is typically connected to the RGM database through a Transaction-Level Modeling (TLM) interface. This setup allows the scoreboard to receive real-time updates about register transactions, enabling it to maintain an accurate and synchronized view of the DUTs state. In more complex environments, such as those involving multiple bus protocols (e.g., APB, AXI), custom adapters are used to bridge the communication between the DUT and the RGM infrastructure. These adapters ensure that protocol-specific nuances are correctly handled, and that the scoreboard receives consistent and protocol-agnostic transaction data.

Moreover, the RGM sequencer, which orchestrates register-level sequences, is often linked to the scoreboard through response ports. This connection facilitates the monitoring of transaction outcomes and supports advanced features such as automatic consistency checks and coverage collection. In multi-master environments, the scoreboard must also account for concurrent accesses and ensure that the register model remains coherent across all access points.

In the `uvm_rgm` framework, register sequences are derived from standard UVM sequences but are tailored to provide greater automation and convenience for register-level verification. Each sequence operates on a data item of type `uvm_rgm_reg_op`, which encapsulates all the necessary information for a register transaction. This includes the target register, its address, the direction of the operation (read or write), and the access mode, whether the transaction is performed through the bus interface (frontdoor) or directly via the HDL path (backdoor). These abstractions allow users to interact with registers using symbolic names or addresses, significantly improving code clarity and maintainability.

The `uvm_rgm` package also provides built-in checking capabilities that are both flexible and adaptable to a wide range of verification strategies. One of the key features supporting this is the shadow model, which resides within the `RGM_DB` component. This model maintains a mirrored view of the DUT's register state and is automatically updated or compared against real-time DUT activity. The synchronization between the DUT and the shadow model is facilitated by a dedicated component known as the Module UVC.

The Module UVC acts as an intermediary between the interface UVC monitor and the register model. It is responsible for processing each register transaction detected by the monitor. When a write operation is observed, the Module UVC invokes the `update()` method, passing along the address, data, and any applicable byte-enable masks. This ensures that the shadow model reflects the most recent state of the DUT. Conversely, when a read operation is captured, the Module UVC calls the `compare_and_update()` method. This function compares the value read from the DUT with the expected value in the shadow model and updates the model accordingly. Any discrepancies are flagged as verification errors, unless explicitly masked.

However, not all registers are expected to remain static. Certain registers, such as status or read-only registers, may be modified by the DUT during simulation. In such cases, the shadow model may fall out of sync with the actual hardware state.

While masking these comparisons can suppress false error reports, it does not address the underlying challenge of verifying these dynamic registers. To overcome this, a reference model or predictor is often employed. This model estimates the expected register values based on the simulation context and DUT behavior. Depending on the verification goals, the predictor can be cycle-accurate, updating the shadow model in real time, or partially accurate, validating only specific registers or time windows.

To support these advanced verification strategies, the `uvm_rgm` containers offer a suite of utility functions for setting and retrieving register values programmatically. This allows the reference model to interact with the shadow model in a controlled and predictable manner.

Additionally, the `uvm_rgm` framework provides two hook methods, `pre_access()` and `post_access()`, to handle side effects associated with register access. The `pre_access()` hook is triggered before any update, fetch, or compare operation, giving access to the original register value and the parameters of the transaction. This can be used for legality checks or logging. The `post_access()` hook, on the other hand, is invoked after the operation completes and can be used for post-processing tasks such as validation or reporting.

This architecture provides a robust and scalable solution for register verification, enabling precise control, automated checking, and seamless integration with the broader UVM environment.

2.4.5 Backdoor

In addition to frontdoor access via the bus interface, the `uvm_rgm` framework supports backdoor access to registers, fields, and memories. This mechanism allows direct interaction with the DUT through its hierarchical HDL structure, bypassing the bus protocol and enabling more efficient or targeted verification operations.

Each `uvm_rgm` object, whether a register, register file, or address map, contains an `hdl_path` field, which stores the HDL path as an associative array of strings. The full HDL path of an object is computed by concatenating the HDL paths of all its parent containers, forming a complete reference to the objects location in the DUT hierarchy. For register fields, the HDL path can be explicitly defined using the final argument of the `uvm_rgm_fld_macro`.

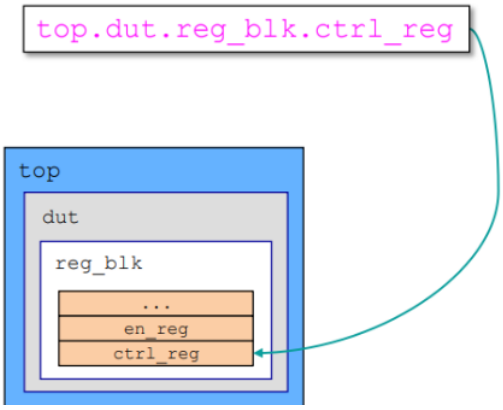

Figure 2.4: HDL Backdoor Path

Typically, the HDL path is set per instance within its container. For example, a register file instance such as `rf_1` should invoke the `set_hdl_path()` method for each contained register. However, this configuration is flexible and can be modified from any point in the testbench environment by calling `object.set_hdl_path()`.

Backdoor write operations are implemented as tasks, meaning they do not consume simulation time. If the forced value needs to persist on the signal for a specific duration, the user can set the static variable `force_time` to the desired interval. The backdoor force will remain active for that duration and will be automatically released afterward.

To enable backdoor access, the HDL path must be correctly assigned to the target register or field. Additionally, the register sequence must be configured to use the backdoor mechanism. This is achieved by setting the `hdl_connection` field of the sequence to `BACKDOOR`. When this setting is applied, the register sequencer accesses the DUT using the HDL path rather than the bus interface.

This approach is particularly useful for verifying read-only registers, injecting values for fault simulation, or bypassing protocol constraints during early-stage verification. It also enables faster access and more precise control over register behavior, making it a valuable tool in complex verification environments.

2.5 RAL

Before the introduction of the Register Abstraction Layer (RAL), the Register and Memory Model (RGM) served as the primary methodology for modeling and

verifying register-level behavior in digital designs. RGM provided a structured framework for defining registers, memories, and address maps, and enabled simulation environments to mirror and validate the state of the DUT. However, as verification methodologies evolved and the complexity of SoC designs increased, several limitations of RGM became apparent.

One of the key challenges with RGM was its tight coupling to specific verification environments and its limited flexibility in adapting to diverse design flows. RGM required significant manual effort to integrate with UVM components, and its learning curve was steep due to the scarcity of comprehensive documentation and examples. Moreover, RGM lacked native support for some of the advanced features introduced in UVM, such as layered sequences, callbacks, and dynamic reconfiguration of register models.

To address these limitations, the Accellera Systems Initiative introduced the Register Abstraction Layer (RAL) as part of the UVM standard. RAL was designed to provide a more unified, extensible, and user-friendly approach to register modeling. It abstracts the underlying register implementation and offers a consistent API for accessing and manipulating register contents, regardless of the bus protocol or DUT architecture. This abstraction enables to write portable and reusable test sequences, simplifies integration with UVM components, and enhances automation in register-level verification [27].

RAL also introduced improved support for functional coverage, and predictive modeling, making it more suitable for modern verification environments. By standardizing the way registers are defined, accessed, and validated, RAL has become the preferred methodology in UVM-based testbenches, effectively superseding RGM in many practical applications [28].

RAL was designed with three pragmatic goals in mind[27]:

1. Abstraction and Reuse: RAL enables stimulus and checking to be written against logical register names and access policies, rather than physical addresses or bus-specific details. This abstraction allows test sequences and checkers to remain valid even as the address map or bus interface changes, promoting reuse across both IP and integration levels.

2. Protocol-Agnostic Access: RAL introduces a standard API (`read`, `write`, `peek`, `poke`, `update`, etc.) that decouples the intent of register operations from the underlying transport mechanism. Adapters bridge RAL operations

to any bus functional model (BFM), supporting both *frontdoor* (bus-based) and *backdoor* (direct HDL path) accesses. This protocol independence is crucial for supporting a wide range of bus architectures and for enabling direct, instantaneous register updates or observations when needed.

3. Predictable Mirrors and Checking: RAL standardizes the semantics of register mirroring, supporting actual, mirrored, and desired values for each register. It provides both implicit and explicit prediction mechanisms, as well as built-in sequences and coverage hooks. This standardization addresses portability and consistency issues that previously plagued ad-hoc register libraries, ensuring that register checking and coverage collection are robust and comparable across projects.

2.5.1 RAL Register Model

In addition to other auxiliary attributes, four primary properties are used to manage field values within the UVM Register Abstraction Layer (RAL):

1. Desired: This property stores the intended value for a register field. It enables efficient updates by allowing the register model to be preloaded with the required values, which can then be applied to the DUT registers through a single update method.

2. Mirrored: This property represents the expected value of the DUT register. The register model attempts to keep this value synchronized with the DUT, although perfect alignment is not always achievable due to asynchronous updates or external modifications.

3. Reset: Reset values are maintained in an associative array indexed by strings. By default, the array contains a single entry at index `"HARD"`, corresponding to the hardware reset value. Additional reset types (e.g., `"SOFT"`) can be introduced through vendor-specific extensions, allowing multiple reset states to be modeled.

4. Value: This is the only publicly accessible property and should be used exclusively within randomization constraints. Direct access to other properties is discouraged; instead, the corresponding register methods should be employed for interaction.

Each field also adheres to an *access policy*, which is derived from the register specification. UVM provides several standard policies, while vendor extensions and custom definitions allow for specialized or unique behaviors [29].
Figure 2.5 illustrates the structure of the RAL register model.

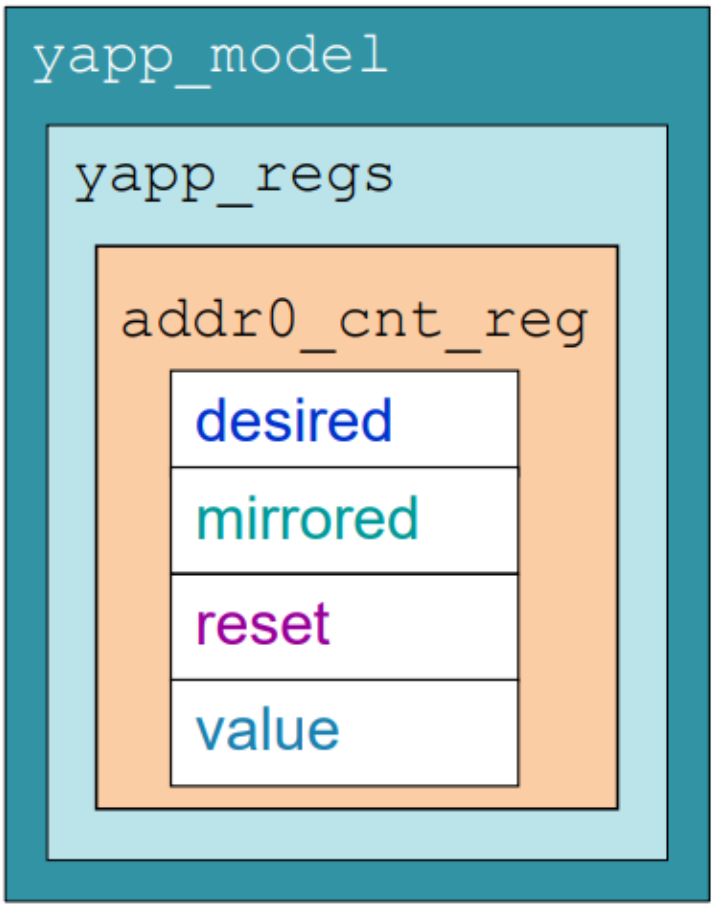

Figure 2.5: RAL Register Model

Register Abstraction Layer or RAL provides originated from UVM, it provides a set of base classes and methods with a set of rules which eases the effort required for register access.

2.5.2 Backdoor in RAL and Register Access Methods

Frontdoor access initiates a read operation on the register model to retrieve information such as address and access policy, thereby constructing a generic register transaction. This transaction is then translated into a DUT-specific operation by the UVC adapter and executed via the UVC sequencer.

In contrast, backdoor access involves direct method calls that read from or write to the register variables within the DUT using the hierarchical path specified by `hdlpath`.

For both frontdoor and backdoor operations, the register model's mirrored and desired values are updated through prediction to reflect the expected state of the DUT post-transaction. In the case of frontdoor access, this update is governed by the current prediction mode. Specifically, under implicit prediction, the update is performed by the register method itself. For backdoor access, the model update is consistently executed by the method.

Backdoor access is designed to emulate the behavior of frontdoor operations while adhering to access policies. For example:

- A backdoor write to a read-only register is ignored.
- A backdoor read of clear-on-read bits results in both the register model and DUT bits being set to zero.

Consequently, a backdoor read may also trigger a read-modify-write cycle on the DUT registers.

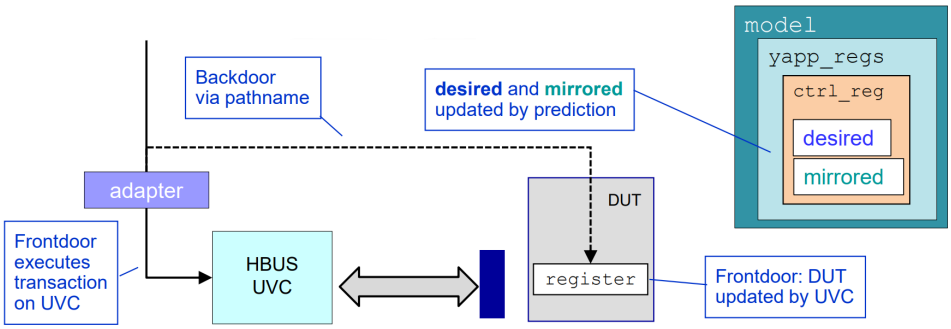

Figure 2.6: Frontdoor vs Backdoor

The standard access methods for registers are `read()` and `write()`. In contrast, the `peek()` and `poke()` methods perform backdoor reads and writes directly on the DUT register using the hierarchical path specified by `hdlpath` [30]. These methods are strictly backdoor and do not initiate any transactions on a UVC.

Unlike conventional backdoor access, `peek()` and `poke()` operations disregard access policies. For instance, it is possible to `poke()` a read-only register.

The register model is updated based on the value sampled or deposited through `peek()` or `poke()`, ensuring it reflects the actual state of the DUT register.

The `poke()` method deposits a value directly onto the DUT register variable, ignoring access restrictions. The value is written to both the DUT and the register

model. For example, poking a zero into Write-Zero-to-Set bits will clear the bits instead of setting them. As a result, `poke()` can assign values to DUT registers that would be otherwise illegal or impossible under normal operation.

The `peek()` method samples the current value of the DUT register variable without modifying it. Access policies are again ignored. For example, peeking at Clear-On-Read bits does not clear them.

The `predict()` method allows manual updates to the `mirrored` and `desired` values within the register model. This is particularly useful when setting the expected value prior to a `read()` operation, such as in cases where the register is internally modified by the DUT.

To retrieve the current mirrored value, the `get_mirrored_value()` method can be used. This is commonly applied in read-modify-write operations where the existing value must be accessed before performing an update.

By default, both `predict()` and `get_mirrored_value()` bypass register access policies. For example, `predict()` can be used to write to a read-only register without restriction.

The `set()`, `get()`, and `randomize()` methods operate exclusively on the `desired` value within the register model. These methods do not interact with the DUT or any UVC interface. Their primary purpose is to configure the register model with a set of required values, which can later be applied to the DUT registers using a single `update()` method call.

The `set()` method respects access policies when assigning values to `desired`. If necessary, the `mirrored` value is used as the current register state to compute the appropriate `desired` value.

The `randomize()` method enables the application of random values to a single field, an individual register, or the entire register model, facilitating flexible and dynamic test scenarios.

A summary of the register access methods is provided in Figure 2.7, which outlines the behavior and characteristics of each method.

Functionality	Method name	Path		Change		Check	Notes
		F	B	D	M		
Basic access	`write()` `read()`	Y	Y	Y	Y	*	Access policies honored *Can check read value against **mirrored**
Backdoor DUT access	`peek()` `poke()`	N	Y	Y	Y	N	Access policies ignored

Model access only	`set()`
	`get()`
	`randomize()`
		-	-	Y	N	N	Access policies honored
							`set`/`randomize` change desired only

Access model expected results	`predict()` `get_mirrored_value()`	-	-	Y	Y	N	`predict` changes **mirrored** and **desired**
Update DUT to match Model	`update()`	Y	Y	N	Y	N	Updates DUT if **desired** is different from **mirrored**. Frontdoor is write(), backdoor is poke()
Mirror model to match DUT	`mirror()`	Y	Y	Y	Y	*	Updates Register Model Frontdoor is read(), backdoor is peek() *Can check read value against **mirrored**

Key: F – frontdoor B – backdoor D – desired M – mirrored

Figure 2.7: Register Access Methods

2.5.3 Prediction

Prediction plays a critical role in UVM-based verification by ensuring that the register model remains synchronized with the actual state of the DUT [31]. This synchronization enables the register model to serve as a reliable reference for checking and validation throughout the simulation process.

There are four primary prediction modes: implicit (or auto), explicit, passive, and manual.

Implicit prediction allows the use of register sequences and built-in tests without a dedicated predictor component. In this mode, register methods update both the register model and the DUT. The register adapter is engaged only when register model access methods are invoked for frontdoor read/write operations. However, implicit prediction is disabled by default. When enabled, only registers accessed via register methods are updated in the model. Registers accessed through UVC sequences are not reflected in the model, which can be problematic when UVC sequences are required.

Explicit prediction requires integration of the register model with both the bus UVC sequencers and the corresponding bus UVC monitor analysis ports. In this mode, implicit prediction is turned off, and updates to the mirrored values are performed externally by a `uvm_reg_predictor` component, one for each bus interface. The predictor receives bus operations observed by a connected bus monitor via the analysis port, performs a reverse-lookup using the observed address to identify the accessed register, and then explicitly calls the registers `predict()` method to update the model. Although this integration requires additional effort, it

ensures that all bus operations, whether initiated by the register model or a third-party bus agent, are observed and reflected in the register model.

When the predictor detects a write operation, it updates the register model with the value written to the DUT register.

Passive prediction also uses a predictor component but does not connect the register model to the UVC sequencer. Instead, the register model is updated using UVC transactions captured by the UVC monitor. However, register sequences cannot be used to initiate transactions on the UVC. Only UVC sequences are permitted for accessing DUT registers and memory. Due to its limitations and minimal code savings compared to explicit prediction, passive prediction is rarely used.

Manual prediction refers to design-specific or custom approaches that complement or replace the standard UVM prediction modes. These methods are typically implemented when none of the built-in modes fully address the needs of the design. Manual prediction may involve user-defined logic to update the register model based on DUT behavior, and is especially useful in cases where prediction must be tightly coupled with unique design features or verification strategies.

Among these modes, explicit prediction is generally recommended. It supports the use of register methods for improved readability and abstraction, and enables UVC register access transactions, which are often necessary for vertical reuse and system-level simulation.

It is important to note that accurate prediction may not be feasible for certain registers, such as read-only status registers set internally by the DUT. These are referred to as *volatile registers*.

2.5.4 Active Monitoring for volatile registers

In advanced SystemVerilog UVM-based verification, maintaining an accurate mirror of the DUT register state within the register model is essential for robust checking and coverage. While most registers can be predicted using standard UVM prediction modes (implicit, explicit, passive, or manual), certain registers, notably read-only status or interrupt registers set internally by the DUT, are inherently *volatile*. Their values may change asynchronously, outside the scope of bus transactions, making them unpredictable by conventional means. For such cases, UVM provides

a mechanism known as active monitoring, which leverages user-defined backdoors to automatically update the register model whenever the DUT register changes value [27].

Active monitoring is implemented by attaching a custom backdoor to the register model. This backdoor observes the DUT register variable and updates the mirrored value in the register model as soon as a change is detected. This approach is particularly useful for a small number of key volatile registers whose values may influence other registers or the overall DUT behavior.

The process involves:

- Defining a user backdoor class that extends `uvm_reg_backdoor`.
- Overriding the `read_func()` method to sample the DUT variable and update the register model.
- Implementing a `wait_for_change()` task that blocks until the DUT variable changes, triggering an update.
- Optionally, overriding `is_auto_updated()` to indicate that the register is actively monitored.

A typical user-defined backdoor for active monitoring might look as follows:

```
class active_monitor_bkdr extends uvm_reg_backdoor;
    function new(string name = "");
        super.new(name);
    endfunction

    function void read_func(uvm_reg_item rw);
        rw.value = new[1];
        rw.value[0] = DUT.reg_six; // Hierarchical path to DUT register
        rw.status = UVM_IS_OK;
    endfunction

    local task wait_for_change(uvm_object element);
        @(DUT.reg_six); // Wait for value change event
    endtask

    function bit is_auto_updated(uvm_reg_field field);
        return 1;
    endfunction
endclass
```

The backdoor is then instantiated and attached to the relevant register in the testbench:

```
active_monitor_bkdr r6_bd = new("r6_bd");
reg_rm.registers.reg_six.set_backdoor(r6_bd);
r6_bd.start_update_thread(reg_rm.registers.reg_six);
```

While the above example hardwires the backdoor to a specific DUT variable, this approach can be limiting if multiple registers require active monitoring or if the backdoor needs to be packaged for reuse. To address this, the backdoor can be generalized using SystemVerilog *virtual interfaces*. The interface encapsulates the register variable, and the backdoor accesses the variable through the interface, allowing for scalable and reusable active monitoring across multiple registers.

Active monitoring is best reserved for a small number of volatile registers due to the simulation overhead of monitoring value changes. It is not intended as a replacement for standard prediction modes for all registers. Registers that are actively monitored should not be updated by other prediction mechanisms to avoid inconsistencies.

Chapter 3

Backdoor implementation

3.1 Backdoor Access

From this point forward, a comparative analysis is conducted to evaluate three distinct register access methodologies employed in hardware verification environments. These methods differ primarily in the mechanism used to interact with the register model, while the underlying test logic and verification objectives remain consistent across all cases. The purpose of this comparison is to assess the behavioral and performance implications of each access mode under controlled test conditions.

The access methods under evaluation are as follows:

- RGM sequencer
- RAL frontdoor
- RAL backdoor

Each method represents a different abstraction level and access path to the register model. The RGM sequencer operates within the Register Generation Model framework, typically using sequences that interact with the register database through protocol-compliant transactions. The RAL frontdoor method utilizes the UVM Register Abstraction Layer to perform register accesses via the bus interface, simulating real-world interactions with the design under test (DUT). In contrast, the RAL backdoor method bypasses the bus interface and directly manipulates register values within the simulation environment, offering faster execution and simplified access for certain verification tasks.

To ensure a fair and consistent comparison, two test scenarios are defined, referred to as Test T1 and Test T2. These tests differ in the number of polling

accesses issued to a specific status register.

Polling is a technique commonly used in hardware verification to monitor the state of a device or component by repeatedly reading a register until a desired condition is met. This condition may reflect readiness, completion, or error status, and is typically encoded in a dedicated status field. Polling involves issuing successive read operations to the target register field, evaluating the returned value against an expected condition, and continuing the process until the condition is satisfied. This approach enables synchronization between the testbench and the DUT without relying on interrupt-driven mechanisms. However, polling can introduce performance overhead due to repeated bus transactions and may increase simulation time, particularly if the condition requires many cycles to be fulfilled.

In this analysis, each access method, RGM sequencer, RAL frontdoor, and RAL backdoor, is employed to perform status register reads during the polling process. To maintain consistency and control timing behavior, a configurable delay is introduced between each polling attempt. This delay is passed as a command-line argument during test execution and stored in a variable referred to as `polling_delay`.

The implementation of each access method is illustrated using simplified pseudocode examples, highlighting the procedural differences and structural characteristics of the respective approaches.

The expected outcome of this comparative analysis is that the use of RAL backdoor access will result in a reduction in both simulation time and overall test duration. This is primarily attributed to the nature of backdoor access, which bypasses the standard bus interface and directly manipulates register values within the simulation environment. By eliminating protocol-level transactions and associated signal activity, backdoor access significantly accelerates register interactions, making it particularly advantageous in scenarios involving frequent polling or large-scale register operations.

In contrast, both RAL frontdoor and backdoor access methods are expected to incur higher memory and computational overhead. This is primarily due to the additional tasks and components that these access methods introduce into the UVM testbench. During the build phase of the simulation, these tasks are instantiated and integrated into the testbench infrastructure, resulting in increased memory usage. Furthermore, the presence of these tasks contributes to greater simulation complexity, as they involve additional layers of abstraction, transaction handling,

and callback mechanisms that must be managed throughout the test execution.

Therefore, while frontdoor and sequencer accesses offer higher fidelity and protocol realism, backdoor access provides a more efficient alternative for performance-sensitive verification tasks. The trade-off between accuracy and efficiency must be carefully considered based on the specific objectives and constraints of the verification campaign.

3.2 Synopsys VCS profiling

The Synopsys VCS® (Verilog Compiler Simulator) is a high-performance simulation and verification platform widely adopted in the semiconductor industry. It provides advanced simulation capabilities, constraint-solving engines, and profiling tools that enable efficient debugging, performance tuning, and verification closure [32].

These features allow users to analyze simulation behavior at both compile-time and runtime, enabling data-driven decisions for improving verification efficiency.

The profiling tools allow users to visualize coverage hotspots, correlate them with simulation performance, and prioritize test development accordingly. Intelligent Coverage Optimization (ICO) further enhances this process by identifying redundant tests and focusing resources on under-covered areas [33].

In the context of this thesis, VCS profiling will be employed to analyze simulation performance and identify potential inefficiencies in the testbench and register access mechanisms. Profiling will be enabled by passing the vcs_profiling argument to the simulation run command. This directive activates the internal profiling engine of VCS, allowing for the collection of detailed runtime statistics and performance metrics. The resulting data will be used to support the comparative analysis of access methodologies and to guide optimization efforts.

When conducting simulation profiling using VCS, it is considered best practice to perform memory profiling and time profiling in separate simulation runs. This separation is recommended due to the potential for interference between the two profiling modes, which can lead to inaccurate or misleading results.

Memory profiling in VCS typically involves tracking dynamic memory allocations, object instantiations, and peak memory usage across the simulation timeline. This process introduces additional instrumentation and logging overhead, which can significantly

alter the timing behavior of the simulation. As a result, any time measurements collected during a memory profiling run may not accurately reflect the true performance characteristics of the testbench or DUT.

Conversely, time profiling focuses on capturing execution latency, transaction durations, and simulation throughput. It requires minimal interference with the simulation flow to ensure that the collected timing data is representative of actual performance. Introducing memory profiling instrumentation during this process can distort the timing profile, particularly in high-activity or memory-intensive scenarios.

To ensure the integrity of both profiling analyses, this thesis adopts the approach of isolating memory and time profiling into distinct simulation runs. This methodology allows for more accurate diagnostics and enables targeted optimizations in each domain without cross-contamination of results.

3.3 RGM sequencer pseudocode

The analysis begins by examining how register access is implemented using the RGM sequencer. This method operates within the RGM framework and utilizes UVM sequences to perform protocol-compliant register transactions. The sequencer coordinates the execution of read and write operations, interacting with the register model and the DUT through a structured and reusable sequence interface.

Below is a pseudocode representation of the polling logic implemented using the RGM sequencer. This logic is designed to repeatedly read a status register until a specified condition is met or a timeout threshold is reached:

```
if (polling counter_override is set)
    polling counter = polling counter override
else if (polling counter is passed via command line)
    polling counter = value from command line

initialize match = false
initialize count = 0

while (not match):
    count += 1
    Wait for a short time defined by polling_delay
    read data from register
    compare masked bits of read data with expected value
    if match:
        exit loop
    else:
        if data is stable:
            increment mismatch counter
        else:
            reset mismatch counter

        if mismatch counter reaches polling counter:
            if scenario is disabled:
                log info and exit
            else:
                report error and dump debug info
                exit loop
```

The polling logic illustrated above is designed to monitor a hardware register until a specific condition is satisfied or a timeout occurs. The process begins by determining the maximum number of polling attempts, which is either overridden by a predefined value or passed dynamically via a command-line argument. This value is stored in the variable `polling_counter`.

Once the polling configuration is established, the algorithm initializes a match flag and a counter to track the number of polling iterations. It then enters a loop where it waits for a short delay, defined by the variable `polling_delay`, before issuing a read operation to the target register. The read value is masked and compared against the expected condition. If a match is detected, the loop terminates successfully.

If the condition is not met, the algorithm evaluates the stability of the read data. Stable mismatches increment a mismatch counter, while unstable readings reset it. This mechanism helps distinguish between transient fluctuations and persistent mismatches. If the mismatch counter reaches the polling threshold, the algorithm checks whether the scenario is configured to tolerate such failures. If tolerance is enabled, the system logs the event and exits gracefully; otherwise, it reports an error, dumps relevant debug information, and terminates the loop.

This polling strategy enables synchronization between the testbench and the DUT without relying on interrupt-driven mechanisms. It also provides configurable control over timing and error handling, making it suitable for a wide range of verification scenarios.

3.4 RAL Frontdoor and RAL Backdoor pseudocode

In the context of the RAL framework, the polling implementation is designed to support both frontdoor and backdoor access methods through a unified codebase. The selection of the access method is determined dynamically at runtime via a command-line argument provided during test setup. This approach promotes flexibility and reuse, allowing the same test logic to be applied across different access paths without duplicating code.

The pseudocode shown below illustrates the polling mechanism implemented using RAL. It demonstrates how the testbench conditionally selects the appropriate access method and performs repeated reads of a status register until a match is

found or a timeout occurs.

```
Set count = max_count

Loop:
    Wait for polling_delay
    If using RAL_backdoor:
        If register is target register:
            Read register using backdoor
            Adjust read value
        Else:
            Read register using RAL_frontdoor
    Else:
        Read register using RAL_frontdoor

    Apply mask to read value

    If masked value equals expected value:
        Exit loop

    If count == 1:
        Report fatal error and exit
    Else:
        Decrement count
```

Similar to the RGM-based implementation, this pseudocode defines a polling loop that repeatedly reads a register until its masked value matches an expected condition or the maximum number of polling attempts is exhausted. The process begins by initializing a counter to the maximum allowed attempts, referred to as `max_count`.

During each iteration, the testbench waits for a predefined delay, specified by the variable `polling_delay`, before performing a read operation. If the backdoor access method is selected and the register being polled is the designated target, the register is accessed using the backdoor mechanism. This may involve direct manipulation of the register model without engaging the DUTs bus interface, and the read value may be adjusted to account for simulation-specific conditions.

For all other cases, including non-target registers or when backdoor access is not selected, the register is accessed using the RAL frontdoor method. This involves issuing a transaction through the bus interface, simulating real hardware behavior and ensuring protocol compliance. After the read operation, the value is masked and compared against the expected result. If a match is detected, the loop exits successfully.

If the condition is not met, the counter is decremented. Once the counter reaches one and no match has been found, the testbench reports a fatal error and exits the loop. This mechanism ensures that the polling process does not continue indefinitely and provides clear feedback in the event of a failure.

It is important to note that within the first conditional block, backdoor access is only applied to the specific target register. This limitation arises because the HDL path required for backdoor access has not been configured for other registers. As a result, the implementation defaults to using the RAL frontdoor method for all non-target registers. This fallback ensures compatibility and avoids simulation errors, as frontdoor access does not require explicit HDL path configuration and can be applied uniformly across the register model.

3.5 Pseudocode for fixed case delay

In this type of test, the polling task has been slightly modified to improve its effectiveness and alignment with the expected timing behavior of the hardware. Rather than initiating the first read operation immediately upon invocation, the

task introduces a short fixed delay before polling begins. This initial delay is approximately equal to the time typically required by the sequencer to complete a read transaction.

Only after this fixed delay, the task proceed to apply the variable delay defined by the `polling_delay` parameter. This adjustment helps mitigate the risk of early mismatches that may occur if the register is read prematurely, before the status value has been updated by the hardware. By deferring the first read operation, the polling mechanism is better synchronized with the hardwares response latency, thereby reducing the likelihood of false negatives and improving the reliability of the test outcome.

It is important to note that the RGM sequencer-based polling task does not require this initial fixed delay prior to initiating the first read. This is because, in UVM, register reads performed through the sequencer inherently consume simulation time due to the transaction-based nature of the access. Specifically, the sequencer initiates a sequence item that is processed by the driver, which then interacts with the DUT through the appropriate protocol interface. This interaction introduces a natural delay before the read operation completes.

As a result, the polling task implemented using the RGM sequencer applies the variable delay defined by `polling_delay` only after the first read operation, rather than before it. This behavior ensures that the timing characteristics of the polling loop remain consistent with the underlying access mechanism and avoids redundant delays that could otherwise extend the simulation unnecessarily.

The following section outlines the specific modifications introduced in the two versions of the polling pseudocode, highlighting the differences in timing behavior and access sequencing between the RAL and RGM implementations.

- RGM sequencer:

```
if (polling counter_override is set)
    polling counter = polling counter override
else if (polling counter is passed via command line)
    polling counter = value from command line

initialize match = false
initialize count = 0

while (not match):
    count += 1
    read data from register
    compare masked bits of read data with expected value
    if match:
        exit loop

        else:
        if data is stable:
            increment mismatch counter
        else:
            reset mismatch counter

        Wait for a short time defined by polling_delay

        if mismatch counter reaches polling counter:
            if scenario is disabled:
                log info and exit
            else:
                report error and dump debug info
                exit loop
```

- RAL frontdoor/backdoor:

```
Set count = max_count

Wait for fixed delay

Loop:
    If using RAL_backdoor:
        If register is target register:
            Read register using backdoor
            Adjust read valuea
        Else:
            Read register using RAL_frontdoor
    Else:
        Read register using RAL_frontdoor

    Apply mask to read value

    If masked value equals expected value:
        Exit loop

    If count == 1:
        Report fatal error and exit
    Else:
        Decrement count
        Wait for polling_delay
```

Chapter 4

Results

4.1 Test T1 - Normal case

The first test serves as a preliminary validation of the GPU functionality and can be considered a hardware-level equivalent of a Hello World test. Its primary objective is to confirm that the basic register interactions and status signaling mechanisms are functioning correctly. In this test, approximately ten polling attempts are performed on a designated status register to monitor its transition to a ready or completed state. Successful execution of this test establishes a baseline for subsequent performance evaluations.

Following this initial validation, performance analysis can be conducted using profiling tools integrated into the verification environment. These tools enable detailed measurement of simulation behavior across multiple dimensions, including execution time and memory usage. To ensure accuracy and isolate the impact of each factor, it is essential to perform time profiling and memory profiling in separate test runs.

Time profiling focuses on measuring execution latency, capturing the duration of register transactions, polling loops, and overall test completion. This analysis helps identify bottlenecks in transaction handling, synchronization delays, and inefficient polling strategies.

In contrast, memory profiling evaluates the resource footprint of the testbench, including memory consumed by instantiated components, register models, sequences, and logging mechanisms. It provides insight into the scalability of the verification environment and highlights areas where memory usage can be optimized.

Running these analyses independently avoids interference between timing and

memory behaviors, which could otherwise obscure the root causes of performance issues. For example, memory-intensive components may slow down simulation time, while timing-sensitive sequences may trigger additional memory allocation. By separating the profiling tasks, more precise diagnostics can be achieved, enabling targeted optimizations in both execution speed and resource efficiency.

Notice that when performing time profiling, the focus will be on both simulation time and total duration of the test:

- Simulation Time: This represents the design's own time base that advanced during the run (e.g., μs/ns or clock cycles of the DUT).
- Total Duration of the Test (Wall-Clock Time / Elapsed Time): This refers to the real time consumed by the job from start to finish on the host machine (including UVM build/reset, polling loops, DPI/PLI operations, logging, etc.).

4.1.1 Time profiling

In this initial phase of the analysis, a slight performance advantage was observed when employing the RAL backdoor access method compared to the RAL frontdoor. As illustrated in Figure 4.1, the results demonstrate a clear trend, the RAL backdoor exhibited a marginal decrease in simulation time, measured at approximately 0.05%, relative to the frontdoor approach. This suggests that, under the given test conditions, the backdoor mechanism may offer a more efficient alternative in terms of simulation runtime.

Additionally, it was noted that the sequencer activity is temporally aligned with the RAL frontdoor operations. The corresponding plot reveals that the sequencer execution is effectively superimposed on the frontdoor timeline, indicating that both mechanisms require an equivalent duration to complete the test sequence. This alignment implies that, despite the minor difference in overall simulation time, the procedural execution of the test remains consistent across both access methods.

The total duration required to complete a simulation test is significantly influenced by the characteristics of the *cluster* on which the test is executed. In the context of distributed computing environments, a cluster refers to a collection of interconnected compute servers that operate collectively to execute jobs submitted by users. These clusters are typically managed by job scheduling systems such as LSF (Load Sharing Facility), which allocate resources based on queue configurations, host types, and user-defined constraints. Each cluster may differ in terms of available memory, CPU performance, queue policies, and resource allocation strategies, all of which

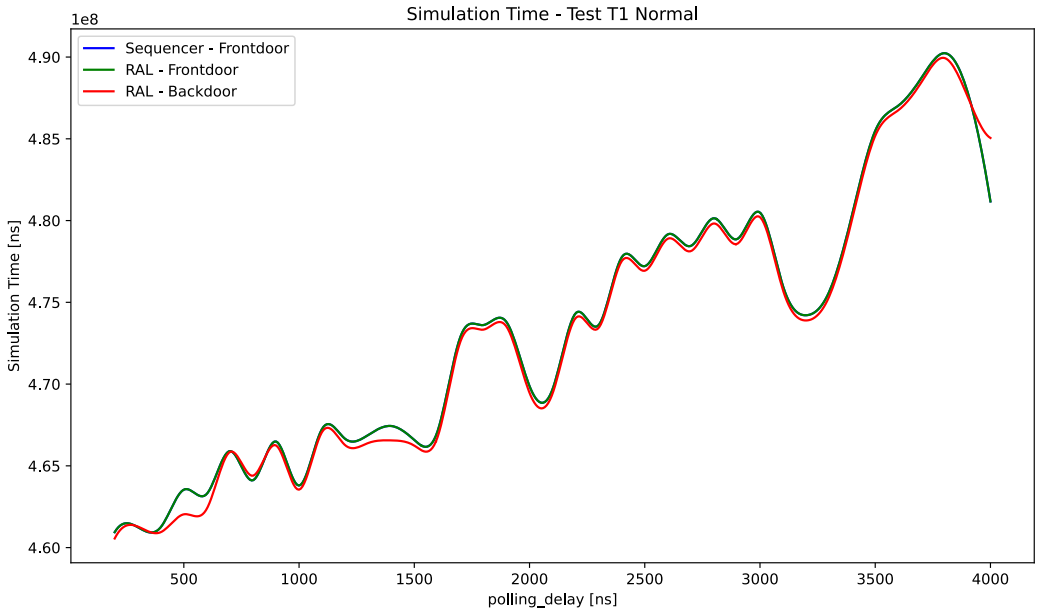

Figure 4.1: Test T1 normal case - Simulation Time

contribute to variations in simulation runtime.

Empirical measurements conducted during the evaluation revealed that the RAL backdoor access method yields a notable improvement in total test duration when compared to other access mechanisms. Figure 4.2 provides a visual representation of the observed behavior. Specifically, the backdoor approach demonstrated an average reduction of approximately 11.58% in total test duration relative to the sequencer-based method, and a less substantial reduction of 2.82% when compared to the RAL frontdoor method. These results underscore the importance of both access methodology and cluster configuration in optimizing simulation efficiency.

4.1.2 Memory profiling

An evaluation of memory usage across the three access methodologies, sequencer, frontdoor, and backdoor, reveals notable differences in resource consumption. As anticipated, the sequencer-based approach exhibits the lowest memory footprint among the three. This outcome aligns with expectations, given the streamlined nature of sequencer operations which typically involve fewer protocol layers and reduced transaction overhead.

In contrast, both the frontdoor and backdoor methods demonstrate comparable levels of memory consumption, with only marginal differences observed between

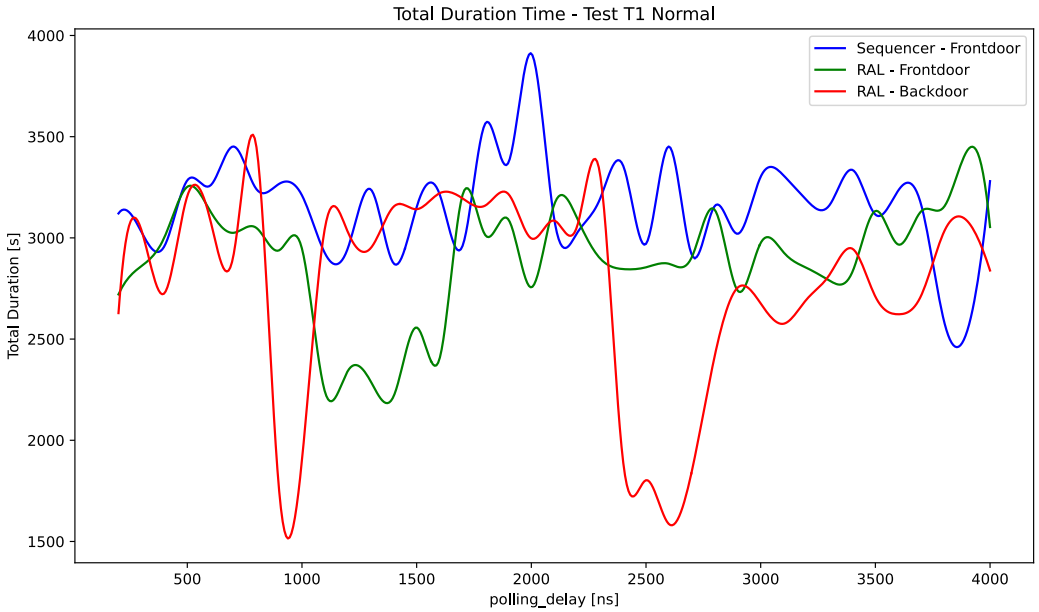

Figure 4.2: Test T1 normal case - Test Total Duration

them. Quantitative analysis indicates that the sequencer method achieves a memory saving of approximately 0.928% when compared to the backdoor approach. Additionally, the backdoor method is found to be 0.153% more memory-intensive than the frontdoor method. The data shown in Figure 4.9 supports the previously discussed hypothesis.

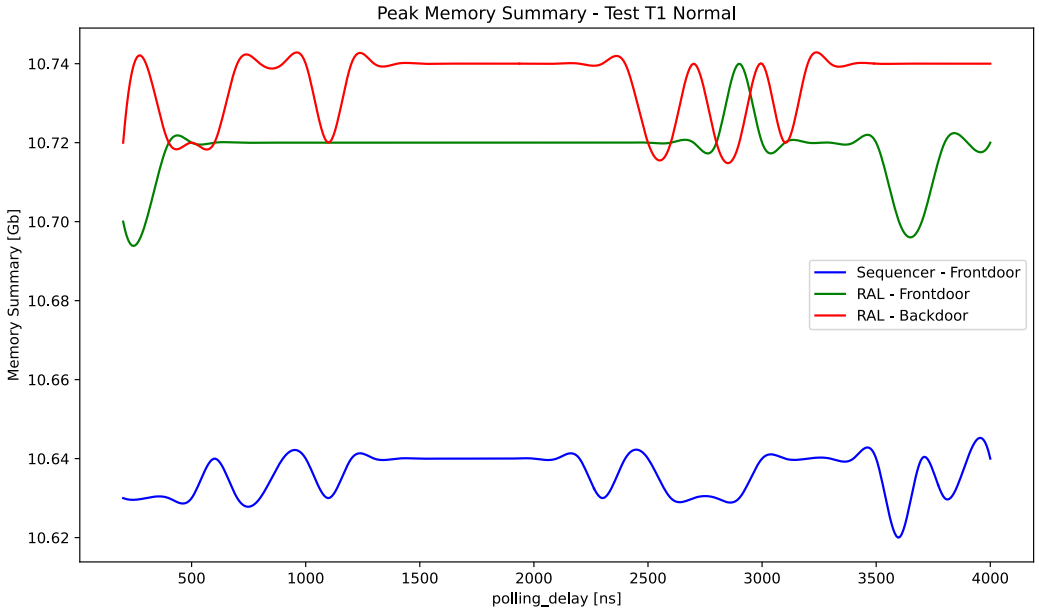

Figure 4.3: Test T1 normal case - Peak Memory Summary

4.2 Test T1 - Fixed delay case

4.2.1 Time profiling

Compared to the baseline scenario discussed in the previous chapter, the current test configuration reveals a more pronounced difference in simulation performance across the evaluated access methods. As anticipated, the RAL backdoor approach consistently demonstrates superior efficiency, requiring less simulation time on average than both the sequencer-based and RAL frontdoor methods.

This improvement can be attributed, in part, to modifications introduced in the polling task used within this test. This adjustment mitigates the risk of premature register reads, which could otherwise result in early mismatches if the hardware has not yet updated the status value. By deferring the first read, the polling task achieves improved synchronization with the hardware's response latency, thereby enhancing the reliability of the test outcome.

Quantitative results from this test indicate that the RAL backdoor method achieves a reduction in simulation time of approximately 1.10% compared to the sequencer, and 0.04% compared to the RAL frontdoor. A graphical summary is presented in Figure 4.4. These findings reinforce the efficiency of the backdoor approach, particularly in scenarios where precise timing alignment and reduced simulation overhead are critical.

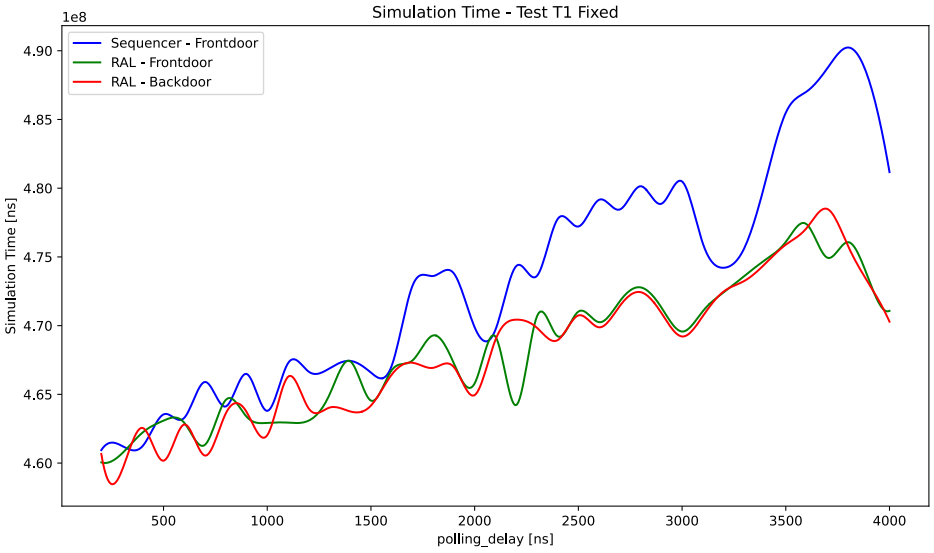

Figure 4.4: Test T1 fixed case - Simulation Time

The total duration of simulation remains strongly influenced by the characteristics

of the cluster on which the test is executed. In the present test scenario, the RAL backdoor method demonstrated a reduction in total test duration of approximately 17.26% when compared to the sequencer-based approach. This result is consistent with expectations, given the reduced protocol overhead typically associated with backdoor accesses.

However, an unexpected outcome was observed when comparing the backdoor method to the RAL frontdoor. Specifically, the backdoor exhibited an increase in total test duration of approximately 3.47% relative to the frontdoor. The results are depicted in Figure 4.5. While this result deviates from the anticipated trend, it is not considered anomalous. The discrepancy is likely attributable to the inherent variability introduced by the cluster environment, which can affect job scheduling and resource contention in unpredictable ways.

These findings reinforce the importance of considering cluster-specific factors when interpreting simulation performance metrics, particularly in comparative analyses involving different access methodologies.

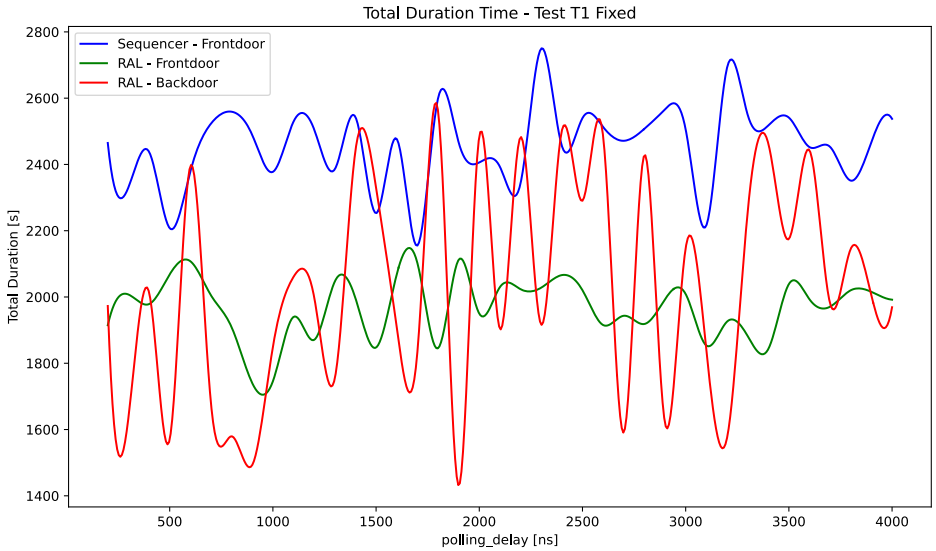

Figure 4.5: Test T1 fixed case - Test Total Duration

4.2.2 Memory profiling

The memory profiling results obtained in this test configuration exhibit a high degree of consistency with those observed in the previous case. This outcome was anticipated, given the structural and procedural similarities between the two test setups.

In this instance, the RAL backdoor method demonstrates a slightly higher memory usage compared to the other approaches. Specifically, it consumes approximately

1.06% more memory than the sequencer-based method and 0.184% more than the RAL frontdoor. Figure 4.9 highlights the key aspects of the analysis. These differences, while minor, are consistent with the expected behavior of backdoor accesses, which may involve additional internal handling or bypass mechanisms that contribute to increased memory utilization.

Overall, the results reinforce the notion that memory consumption remains relatively stable across comparable test configurations, with only marginal variations attributable to the specific access methodology employed.

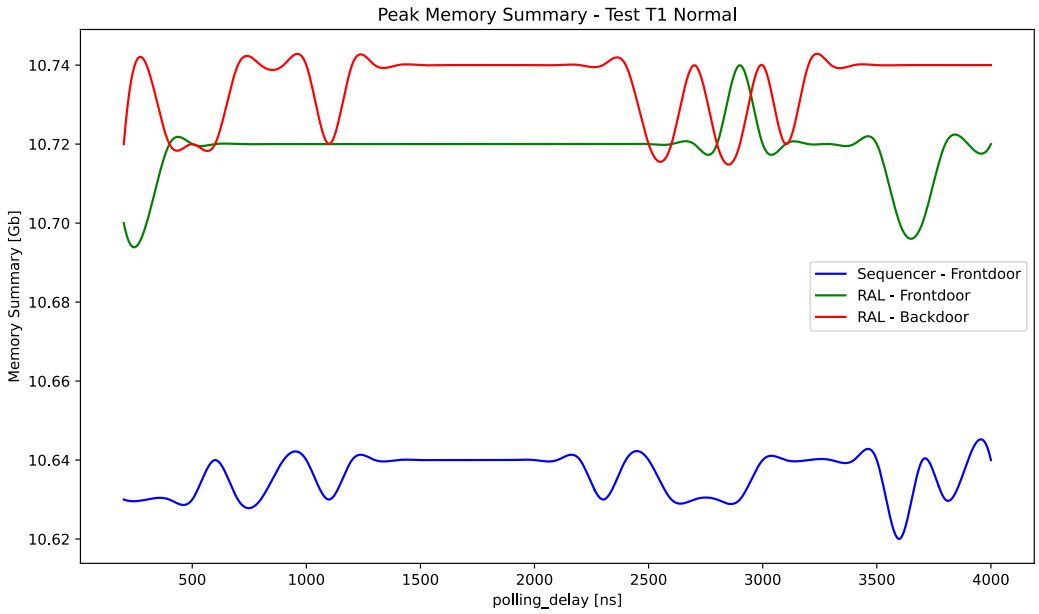

Figure 4.6: Test T1 normal case - Peak Memory Summary

4.3 Test T1 - Real application case

4.3.1 Time profiling

The results obtained in this test scenario, which closely resembles real-world operating conditions of the device, provide some of the most significant insights into the comparative performance of the access methods under evaluation. In particular, the behavior of the system under high values of the `parser_poll_delay` parameter reveals a substantial advantage associated with the RAL backdoor approach.

For instance, when `parser_poll_delay` is set to 3400, the RAL backdoor method achieves a reduction in simulation time of approximately 3% compared to the sequencer-

based method, and 0.62% compared to the RAL frontdoor. These improvements are consistent with expectations, as such delay values are representative of the timing conditions encountered during normal operation of the hardware. The backdoor method benefits from its ability to bypass protocol overhead, thereby reducing latency and improving simulation efficiency.

On average, across the range of tested delay values, the RAL backdoor demonstrates an improvement of approximately 0.772% relative to the sequencer. However, when compared to the RAL frontdoor, the average improvement is negligible. The outcome is captured in Figure 4.7. This suggests that while the backdoor method offers clear advantages in specific timing scenarios, its overall performance may converge with that of the frontdoor under certain conditions.

These findings underscore the importance of aligning test configurations with realistic hardware behavior in order to accurately assess the efficiency of different access methodologies.

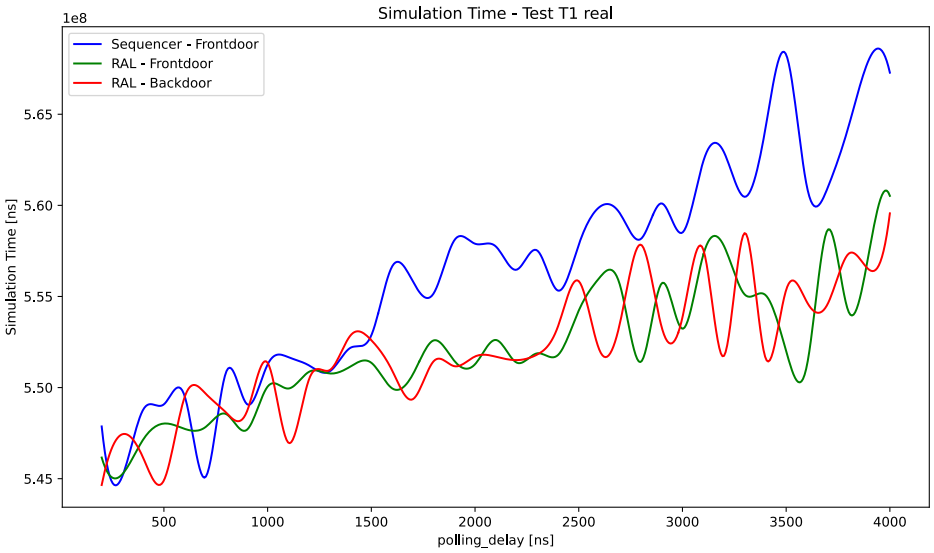

Figure 4.7: Test T1 real case - Simulation Time

The evaluation of total test duration in this test configuration yields results that are consistent with those observed in previous scenarios. Although the overall trend remains similar, the performance gap between the RAL backdoor and the sequencer-based frontdoor approach is less pronounced in this case.

Quantitative analysis indicates that the RAL backdoor method achieves a reduction in total test duration of approximately 3.10% when compared to the RAL frontdoor,

and a more substantial improvement of 6.556% relative to the sequencer. These results reaffirm the efficiency of the backdoor access mechanism, particularly in minimizing simulation overhead. Figure 4.8 serves to illustrate the main findings.

The reduced difference with the frontdoor method may be attributed to specific test conditions or cluster-related factors that influence execution time. Nonetheless, the consistent advantage of the backdoor approach across multiple configurations highlights its suitability for scenarios where simulation performance is a critical consideration.

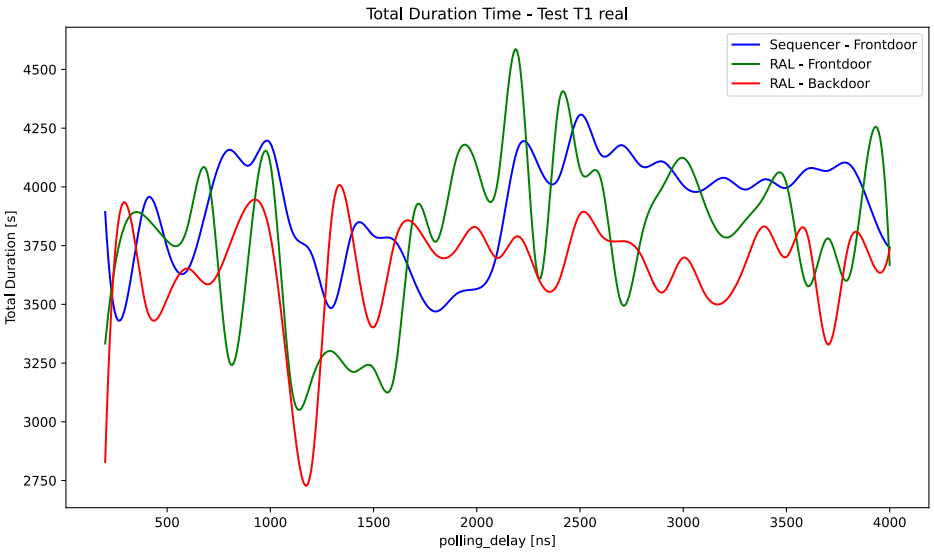

Figure 4.8: Test T1 real case - Test Total Duration

4.3.2 Memory profiling

As stated in earlier sections, memory consumption across different access methodologies tends to remain relatively stable when the underlying test structure and operational flow are preserved. The current configuration continues to support this observation, with results that align closely with those previously reported.

In this particular case, the RAL backdoor method exhibits a marginal increase in memory usage compared to the other approaches. Specifically, it consumes approximately 0.177% more memory than the RAL frontdoor and 0.951% more than the sequencer-based method. The pattern can be discerned from Figure 4.9. Nonetheless, the overall trend confirms that memory usage remains largely consistent across comparable test setups, with only minor deviations influenced by the access

strategy employed.

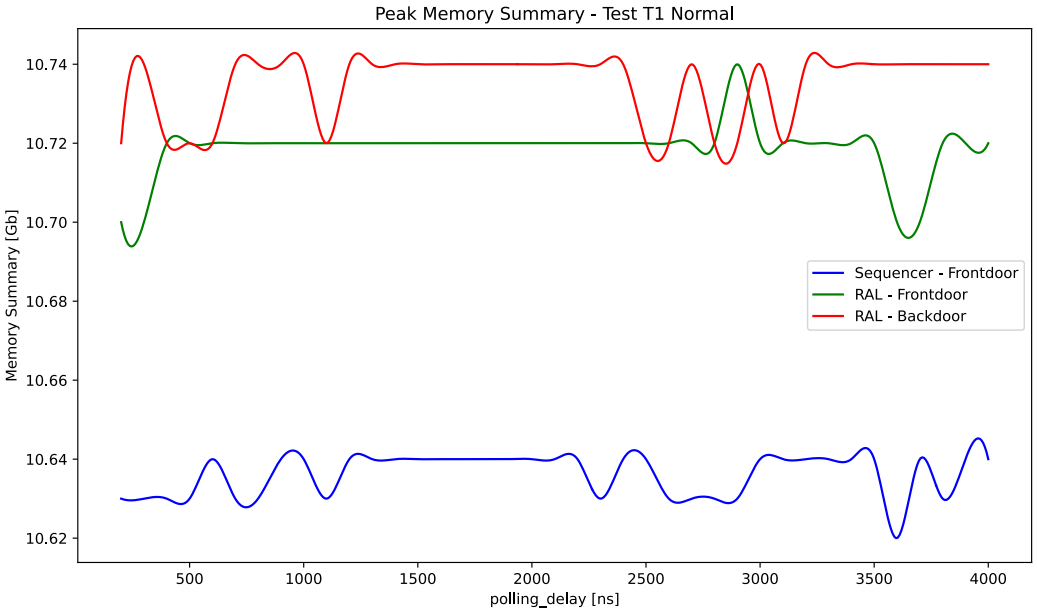

Figure 4.9: Test T1 normal case - Peak Memory Summary

4.4 Test T2 - Normal case

As stated in earlier sections, the complexity of a test configuration has a direct impact on both simulation duration and profiling strategy. The test referred to as `T2` represents a significantly more intricate scenario compared to the previously analyzed cases. This increased complexity arises primarily from the higher number of polling attempts directed at the targeted status register, which intensifies the simulation workload and amplifies the sensitivity to access method efficiency.

Given this setup, it is reasonable to anticipate a more pronounced performance advantage for the RAL backdoor method relative to the sequencer and RAL frontdoor approaches. The reduced protocol overhead and direct memory access characteristics of the backdoor mechanism are expected to yield measurable improvements in simulation time under such demanding conditions.

Due to the substantial runtime associated with executing `T2`, the scope of profiling has been deliberately limited. Only timing profiling has been conducted, and exclusively for the baseline configuration and the fixed delay variant. This decision reflects the practical constraints imposed by the test's duration and the extensive effort required

for comprehensive analysis.

The results obtained from this focused profiling are intended to provide representative insights into the behavior of the access methods under high-load conditions, while maintaining feasibility within the overall verification workflow.

However, the results obtained reveal that the average improvement associated with the RAL backdoor method is slightly lower than initially expected. This reduction in relative gain is attributed to the considerable overhead introduced by other operations performed during the test, which diminish the impact of access method optimization. Despite this, the backdoor approach continues to demonstrate consistent efficiency, particularly in scenarios where polling intensity is high and protocol simplification is beneficial.

4.4.1 Time profiling

The outcomes of the initial test indicate that the substantial overhead introduced by additional polling operations on unrelated registers diminishes the observable performance gains. This phenomenon is consistent with expectations, as such overhead can obscure the benefits of targeted optimizations. Nevertheless, the enhancement provided by the RAL backdoor mechanism, when compared to access via the Sequencer, remains evident. This improvement was quantified at 0.13%. On average, the performance metrics of the RAL backdoor and RAL frontdoor approaches are approximately equivalent, with the latter demonstrating a marginal advantage of 0.092%. A detailed view is available in Figure 4.10.

Regarding the total test duration, certain anomalies were observed during measurement. These irregularities are likely attributable to the overhead introduced by concurrent tests or system-level factors. Specifically, the total time required to complete a test is significantly influenced by the characteristics of the *cluster* on which the test is executed. These clusters allocate computational resources based on queue configurations, host specifications, and user-defined constraints. Variations in available memory, CPU performance, queue policies, and resource allocation strategies across clusters contribute to inconsistencies in simulation runtimes.

In the present case, it was observed that the RAL backdoor access method incurred a longer execution time compared to both the RAL frontdoor and Sequencer-based access methods. Quantitatively, the RAL backdoor was measured to be 21.21% more time-consuming than the RAL frontdoor, and 6.97% more time-consuming

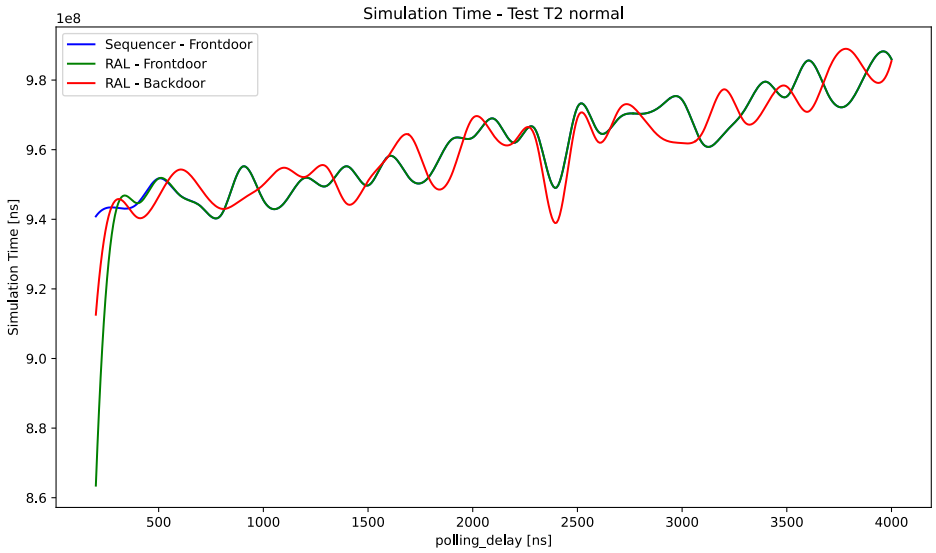

Figure 4.10: Test T2 normal case - Simulation Time

than the Sequencer-based access. The graphical output in Figure 4.11 complements the textual explanation

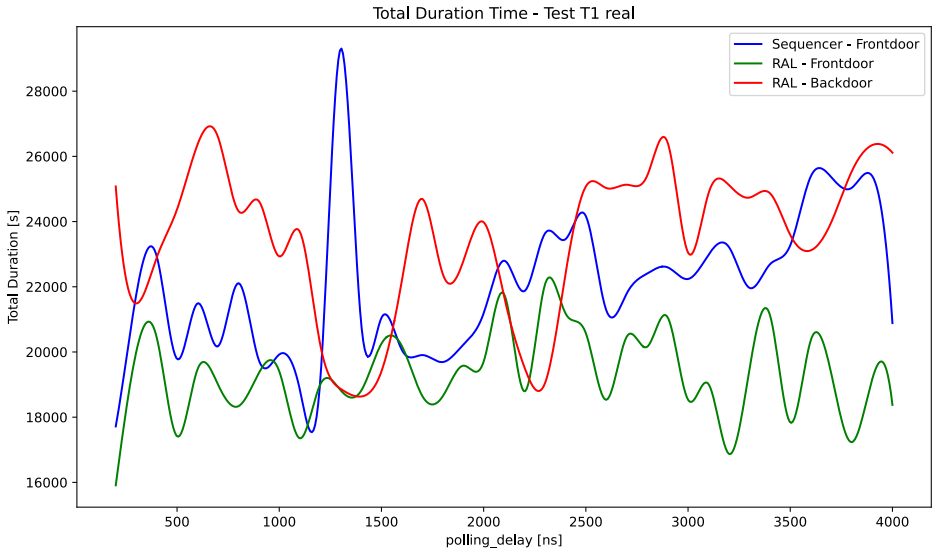

Figure 4.11: Test T2 normal case - Test Total Duration

4.5 Test T2 - Fixed case

4.5.1 Time profiling

In this test scenario, a performance improvement was observed when compared to the baseline case, which aligns with expectations. Specifically, the RAL backdoor access method demonstrated a measurable enhancement over the Sequencer-based access, with a recorded gain of 0.406%. Despite this, the RAL frontdoor approach exhibited a slightly superior performance relative to the backdoor, with an average advantage of 0.08%. Figure 4.12 encapsulates the core result of this section. These results suggest that while the RAL backdoor offers benefits over traditional Sequencer access, the frontdoor method remains marginally more efficient under the tested conditions.

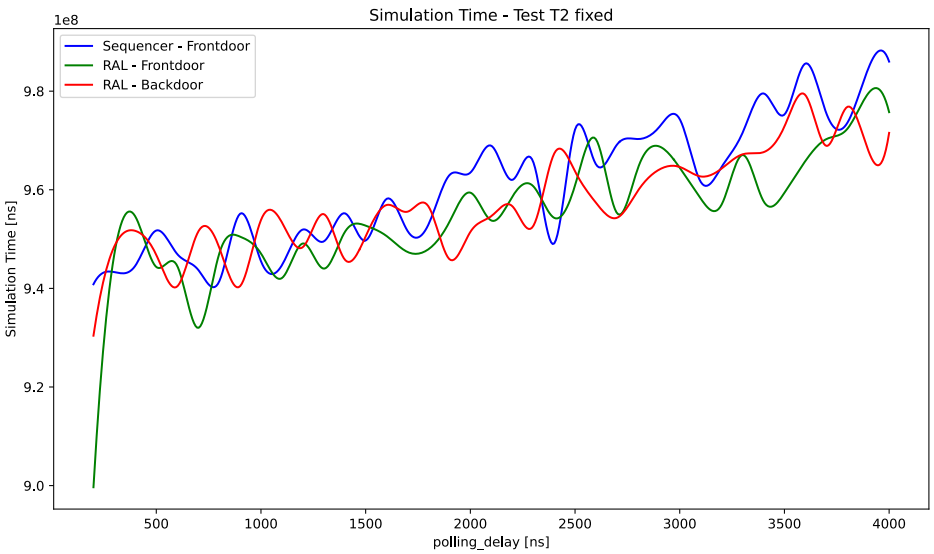

Figure 4.12: Test T2 fixed case - Simulation Time

In the second test scenario, a more pronounced improvement was observed. Although the total execution time associated with the RAL backdoor access method remained higher than that of both the RAL frontdoor and Sequencer-based approaches, the performance gap was notably reduced compared to previous measurements. Specifically, the RAL backdoor was found to be 3.62% more time-consuming than the RAL frontdoor and 4.94% more time-consuming than the Sequencer-based access. This represents a significant improvement relative to the earlier test, where the RAL backdoor exhibited a 21.21% and 6.97% increase in total duration compared to the RAL frontdoor and Sequencer-based methods, respectively. Figure 4.13 outlines the results of the analysis. These findings suggest that under certain conditions, the efficiency of the RAL backdoor can approach that of the alternative access methods.

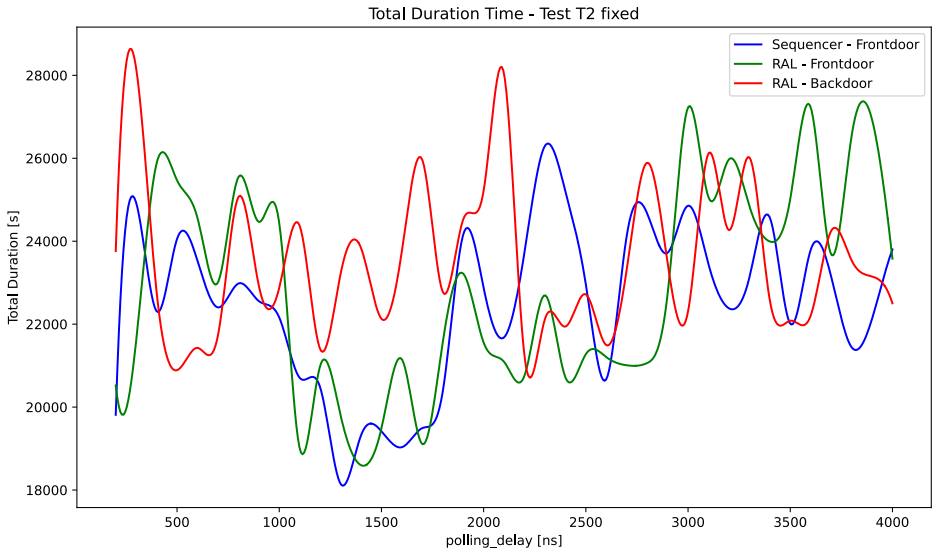

Figure 4.13: Test T2 fixed case - Test Total Duration

These observations suggest that, had a realistic scenario also been implemented for test T2, a more substantial performance improvement could have been achieved. This enhancement would likely have manifested in both the simulation runtime and the overall duration of the test. The implication is that the current test configuration may not fully capture the potential benefits of the proposed access method, and that further gains could be realized under conditions that more closely resemble practical deployment environments.

Chapter 5

Conclusion

This thesis has presented a comprehensive evaluation of three distinct register access methodologies, RGM sequencer, RAL frontdoor, and RAL backdoor, within the context of hardware verification environments. The analysis was conducted through a series of structured test scenarios, each designed to isolate and measure the behavioral and performance characteristics of the access methods under varying operational conditions.

The comparative study began with the implementation of a baseline test (Test T1 - Normal Case), which served as a foundational reference for subsequent evaluations. In this scenario, the RAL backdoor method demonstrated a marginal improvement in simulation time over the RAL frontdoor, while maintaining comparable memory usage. The sequencer-based approach, although slightly slower in simulation time, exhibited the lowest memory footprint. These findings confirmed the hypothesis that backdoor access can offer performance benefits by bypassing protocol overhead, albeit with a modest increase in memory consumption.

Subsequent tests introduced refinements to the polling mechanism, including the addition of a fixed delay to better align with hardware response timing (Test T1 - Fixed Case). This modification resulted in a more pronounced performance gain for the RAL backdoor, which achieved a 1.10% reduction in simulation time compared to the sequencer. However, an unexpected increase in total test duration relative to the frontdoor method was observed, likely due to variability in cluster resource allocation. These results highlighted the sensitivity of simulation performance to environmental factors such as job scheduling and compute node characteristics.

The real application scenario (Test T1 - Real Case) provided the most representative conditions for assessing access method efficiency. Under these conditions, the RAL backdoor consistently outperformed the sequencer and, in most cases, the frontdoor

method. The average simulation time improvement over the sequencer was measured at 0.772%, with a peak gain of 3% observed at higher polling delay values. These results underscore the suitability of the backdoor approach for realistic verification workloads, where protocol simplification and reduced latency are critical.

In contrast, the more complex test configuration (Test T2) introduced a higher number of polling operations, thereby increasing the simulation workload and amplifying the impact of access method efficiency. In the normal case, the RAL backdoor exhibited only a slight improvement over the sequencer (0.13%) and was marginally outperformed by the frontdoor (0.092%). Moreover, the total test duration revealed a significant overhead for the backdoor, which was 21.21% and 6.97% more time-consuming than the frontdoor and sequencer, respectively. These results suggest that the benefits of backdoor access may be diminished in high-load scenarios where other sources of overhead dominate.

However, the fixed delay variant of Test T2 demonstrated a recovery in backdoor performance. The simulation time improvement over the sequencer increased to 0.406%, and the performance gap with the frontdoor narrowed to 0.08%. Additionally, the total duration overhead was reduced to 3.62% and 4.94% compared to the frontdoor and sequencer, respectively. These findings indicate that with appropriate timing adjustments, the efficiency of the backdoor method can be restored even in complex test environments.

The analysis also revealed that memory usage remained relatively stable across all test configurations, with only minor variations attributable to the access method employed. The sequencer consistently consumed the least memory, while the backdoor and frontdoor methods exhibited slightly higher usage due to additional abstraction layers and internal handling mechanisms.

Overall, the results of this study support the conclusion that the RAL backdoor access method provides a viable and efficient alternative for performance-sensitive verification tasks. Its advantages are most evident in scenarios that closely mirror real-world hardware behavior, where protocol overhead can be minimized and simulation latency reduced. However, the effectiveness of the backdoor approach is contingent upon the complexity of the test, the configuration of the polling mechanism, and the characteristics of the execution environment.

It is also worth noting that the current implementation of Test T2 may not fully reflect the potential of the RAL backdoor method. Had the test been configured

to more accurately emulate a realistic application scenario, it is likely that greater performance improvements would have been observed. Such enhancements would be expected to manifest in both simulation runtime and total test duration. This observation reinforces the importance of aligning test environments with practical deployment conditions to better capture the true benefits of the evaluated methodologies.

Future work may explore the integration of dynamic access method selection based on runtime profiling, as well as the development of adaptive polling strategies that further optimize simulation performance. Additionally, extending the analysis to include power consumption could provide a more holistic assessment of access method trade-offs in hardware verification workflows.

Appendix A

Scripts Python

Python script used to generate graphs:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import make_interp_spline

title = "Simulation Time - Test T1 Normal"
name = "T1_norm_time_sim"
# Your data
import numpy as np
x = np.array([200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100,
              1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100,
              2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000, 3100,
              3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000])

y1 = np.array([
   ...
])

y2 = np.array([
    ...
])

y3 = np.array([
    ...
])
```

```
# Create a smooth x-axis
x_smooth = np.linspace(x.min(), x.max(), 500)

# Interpolate y-values
y1_smooth = make_interp_spline(x, y1)(x_smooth)
y2_smooth = make_interp_spline(x, y2)(x_smooth)
y3_smooth = make_interp_spline(x, y3)(x_smooth)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(x_smooth, y1_smooth, label='Sequencer - Frontdoor', color='blue')
plt.plot(x_smooth, y2_smooth, label='RAL - Frontdoor', color='green')
plt.plot(x_smooth, y3_smooth, label='RAL - Backdoor', color='red')

plt.xlabel('polling_delay [ns]')
plt.ylabel('Simulation Time [ns]')
# plt.ylabel('Total Duration [s]')

# plt.ylabel('Total Duration')
plt.title(f"{title}")

# Add legend
plt.legend()

# Disable grid
plt.grid(False)

# Titolo e layout
plt.tight_layout()
plt.savefig(f"{name}.pdf", dpi=2400)
plt.show()
```

Bibliography

[1] Anshel Sag. Qualcomms mobile gpu innovations power the future of gaming. https://www.forbes.com/sites/moorinsights/2021/08/30/qualcomms-mobile-gpu-innovations-power-the-future-of-gaming/, 2021.

[2] Medium. Gpu hystory, 2018.

[3] Intel. Introduction to the xe-hpg architecture, 2021.

[4] Ana Mihut, Christoph Kubisch, and Manuel Kraemer. Advanced api performance: Mesh shaders, 2021.

[5] NVIDIA Developer Blog. Using mesh shading to optimize your rasterization pipeline, 2021.

[6] AMD. 7 series fpgas configuration user guide (ug470), 2024.

[7] Wikipedia contributors. Adreno. https://en.wikipedia.org/wiki/Adreno, 2024.

[8] Qualcomm. Integrated gpu chipset | qualcomm adreno gpu. https://www.qualcomm.com/products/technology/processors/adreno, 2024.

[9] Qualcomm. The rise of mobile gaming on android: Qualcomm snapdragon technology leadership. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/the-rise-of-mobile-gaming-on-android-qualcomm-snapdragon-technology-leadership.pdf, 2024.

[10] Accellera Systems Initiative. Universal verification methodology (uvm) 1.2 class reference manual. https://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf, 2014.

[11] Accellera Systems Initiative. Uvm user guide 1.2. https://accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf, 2014.

[12] Accellera Systems Initiative Forum. Uvm phasing discussion. https://forums.accellera.org/topic/509-uvm-phasing/, 2011.

[13] Tom Fitzpatrick. Uvm: Ready, set, deploy! https://accellera.org/images/resources/videos/UVM_Ready_Set_Deploy_2012.pdf, 2012.

[14] Accellera Systems Initiative. Uvm presentation dac 2011. https://accellera.org/images/activities/committees/uvm/UVM_Presentation_DAC2011_Final.pdf, 2011.

[15] Accellera Systems Initiative. *Universal Verification Methodology (UVM) User Guide*, 2017. Available at https://accellera.org/downloads/standards/uvm.

[16] Janick Bergeron. Writing testbenches: Functional verification of hdl models. *Springer*, 2003.

[17] Randy Katz. Contemporary logic design. *Pearson*, 2004.

[18] Brian Bailey. Error handling in hardware verification. *EE Times*, 2010. Available at https://www.eetimes.com/error-handling-in-hardware-verification/.

[19] Mentor Graphics. *OVM Register and Memory Package User Guide*, 2009. Available at https://www.mentor.com/products/fv/verification-ip/ovm/.

[20] Accellera Systems Initiative. *UVM Register Layer Reference Manual*, 2017. Available at https://accellera.org/downloads/standards/uvm.

[21] Sharon Rosenberg. Register this! experiences applying uvm registers. In *DVCon Proceedings*, 2011. Cadence Design Systems.

[22] Mark Litterick and Marcus Harnisch. *Advanced UVM Register Modeling*, 2016. Verilab GmbH.

[23] 1Library. Using the uvm_rgm package - register and memory package, 2021.

[24] Accellera Systems Initiative. *IP-XACT User Guide*, 2018.

[25] Wikipedia contributors. Ip-xact, 2024.

[26] Accellera Forum. Register package - rgm vs other options, 2020.

[27] Accellera Systems Initiative. Uvm cookbook: Register layer, 2011.

[28] Siemens EDA (Mentor Graphics). Uvm register layer (ral) overview, 2020.

[29] ChipVerify. Uvm register abstraction layer, 2023.

[30] Verification Guide. Introduction to uvm ral - verification guide, 2023.

[31] Accellera Systems Initiative. Uvm ral - uvm systemverilog discussions - accellera systems initiative forums, 2012.

[32] Synopsys. Vcs datasheet. `https://www.synopsys.com/content/dam/synopsys/gated-assets/verification/vcs-ds.pdf`, 2024.

[33] Synopsys. Vcs: Functional verification solution. `https://www.synopsys.com/verification/simulation/vcs.html`, 2024.