Politecnico di Torino

Politecnico di Torino
Master Degree in Electronic Engineering

Behavioral Modeling of dynamic virtual FIFOs

Candidate:
Massimo Giacobbe

Supervisors:
Prof. Guido Masera
Eng. Joao Leonardo Fragoso

Academic Year 2024-2025
Politecnico di Torino
Dipartimento di Elettronica e Telecomunicazioni

Politecnico di Torino

Politecnico di Torino

Master Degree in Electronic Engineering

Behavioral modeling of dynamic virtual FIFOs

Candidate:
Massimo Giacobbe

Supervisors:
Prof. Guido Masera
Eng. Joao Leonardo Fragoso

Academic Year 2024-2025

Ai miei genitori e ai miei nonni, grazie

Abstract

This thesis presents a behavioral model for dynamic virtual First-In-First-Out (FIFO) buffers within a descriptor-based memory system architecture, designed to support efficient context switching and preemption in high-performance hardware environments. The hardware module manages a shared pool of register buffers across multiple virtual FIFOs each representing either a cluster (processing destination) or a heap (free buffer pool) using descriptor RAM to dynamically link and organize buffer allocations.

Clusters are the destinations where incoming transactions are processed. To enable efficient dispatching, the system implements a "Serial In, Parallel Out" behavior: transactions arriving from a single input are dynamically divided and routed to different clusters. The descriptor-based implementation supports dynamic FIFO sizing, allocating required memory based on traffic demand of each cluster, this reduces the overall memory requirements and chip area, with only minimal delay overhead.

The behavioral model replicates the internal buffer allocation and deallocation mechanisms of the memory system, maintaining a software-level representation of the heap and cluster FIFOs. It is designed to ease debugging by improving traceability of transactions, hiding the RTL complexity from the verification team, allowing developers to monitor buffer activity and identify mismatches or protocol violations more effectively. The model is integrated into a standalone testbench environment that autonomously performs save/restore checks based solely on peripheral and debug signal sequences, without relying on external triggers. This facilitates the detection of context mismatches and protocol errors during preemption scenarios.

The testing framework includes randomized and corner-case scenarios such as ping-pong context switching and back-pressure preemption, ensuring robust validation. By abstracting memory system behavior in a cycle-approximate manner, the model enables faster simulation and easier traceability of buffer

transactions, ultimately aiding in the verification and debugging of complex FIFO-based designs.

The final behavioral model is validated against a suite of standalone tests and is intended for use in regression environments for broader system-level verification.

Contents

Background

0.1 Design Verification

Pre-silicon verification is a critical phase in VLSI design, especially due to the high non-recurring engineering (NRE) costs associated with fabrication. To mitigate risk and reduce development time, modern verification methodologies aim to automate and standardize the verification process as much as possible.

SystemVerilog introduced several software engineering concepts into hardware verification, including object-oriented programming, constrained random stimulus generation, and coverage-driven verification. Among the methodologies built on SystemVerilog, the Open Verification Methodology (OVM), developed by Cadence and Mentor Graphics, was one of the first widely adopted frameworks.

0.1.1 Universal Verification Methodology (UVM)

To address limitations in OVM and unify verification practices across vendors, the Universal Verification Methodology (UVM) was introduced. UVM provides a standardized, modular, and reusable framework for building testbenches, enabling teams to collaborate more effectively. Its transaction-level modeling (TLM) abstraction allows components to be treated as black boxes, promoting reuse and scalability.

A typical UVM testbench consists of:

- **Transactors**, which interface directly with the Device Under Test (DUT), driving inputs and sampling outputs.
- **Higher-level components**, which generate stimuli and verify DUT behavior against a reference model.

All DUT-facing components are encapsulated within an `agent`, which is itself part of the `environment`, instantiated by the `test`.

0.1.1.1 UVM Phasing

UVM defines a phased approach to testbench execution, which automates and standardizes simulation flow. The phases include:

- **Build phase**: Components are constructed and connected.
- **Run phase**: Stimuli are generated, and coverage data is collected.

- **Cleanup phase**: Results are analyzed, coverage is reported, and the simulation concludes.

0.1.1.2 Stimulus Generation: Sequence Items and Sequences

Stimuli are generated using `sequence_item` and `sequence` classes.

Sequence Item A `sequence_item` encapsulates the data to be sent to the driver. Input fields are typically declared as `rand`, while output fields are not. Constraints can be applied to guide randomization.

```
// Example of the data contained in a sequence_item
typedef struct packed {
    logic [nbit-1:0] a;
    logic [nbit-1:0] b;
    logic cin;
} i_var;

rand i_var rand_in;
logic [nbit-1:0] s;
logic cout;
```

Sequence A `sequence` is a class that defines a `body()` task, which creates and randomizes a specific set of items for direct tests or random set of items for improving coverage, then sends them to the sequencer.

```
// Body task of a sequence class
virtual task body();
    p4_sequence_item seq_item;
    for (int i = 0; i < nTrans; i++) begin
        seq_item = p4_sequence_item::type_id::create("
           seq_item");
        if (!seq_item.randomize())
            `uvm_error("RANDOMIZE_FAILED", "Failed to
               randomize seq_item")
        start_item(seq_item);
        finish_item(seq_item);
    end
endtask
```

0.1.1.3 Driver and Monitor

The `driver` receives sequence items from the sequencer and drives them to the DUT via an interface. After execution, it calls `item_done()` to signal completion.

The `monitor` passively observes DUT outputs through the same interface and broadcasts them to analysis components such as the scoreboard and coverage collectors.

0.1.1.4 Agent and Scoreboard

An `agent` encapsulates the driver, monitor, and sequencer. It exposes analysis ports that allow external components to connect without needing internal knowledge of the agent's structure, enhancing modularity and reuse.

The `scoreboard` is the core of functional verification. It compares DUT outputs against expected results generated by a behavioral model. Typically, it consists of:

- A **predictor**, which models expected behavior.
- An **evaluator**, which compares actual vs. expected outputs.

In the context of this thesis, the scoreboard plays a crucial role in verifying the behavioral model of dynamic virtual FIFOs, ensuring that the DUT behaves correctly under various stimulus conditions.

Pre-silicon verification is a critical step in VLSI design due to the high non-recurring production costs. As a result, automating as much of the verification process as possible is essential to reduce development time and improve efficiency.

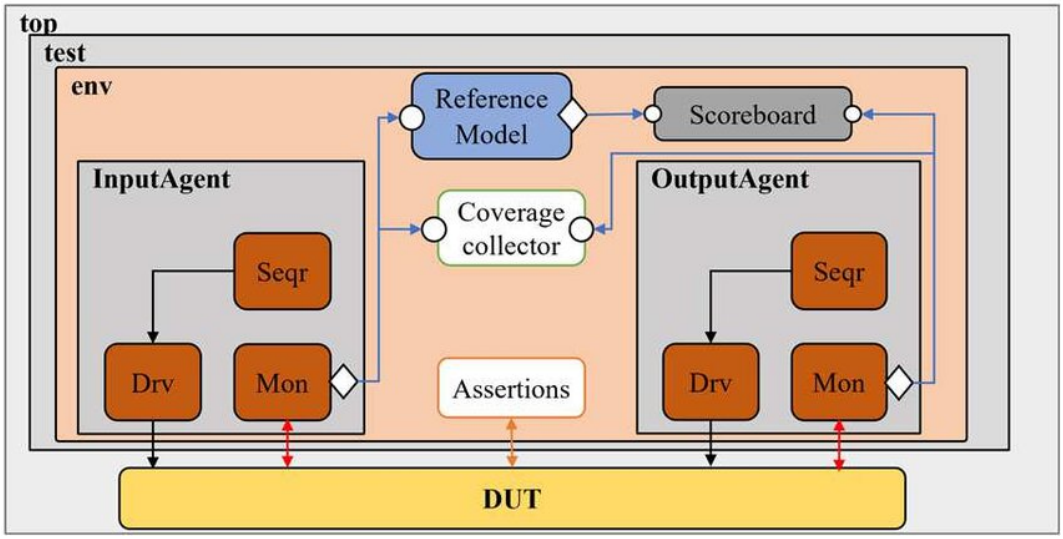

Figure 1: UVM environment (Source: https://www.researchgate.net/figure/Typical-UVM-block-level-testbench_fig1_370640353)

0.2 Graphics Processing Unit

In order to handle graphic workloads, a large number of similar operations needs to be performed. Conventional CPUs are not able to properly handle these workload, which prompted the need for hardware accelerators, GPUs, that are able to handle tens of thousands of operations per clock cycle, provided the operations are all of the same kind.

Graphics Pipeline

The main objective of a GPU is to decide what color each pixel on the screen should be to properly represent the 3D scene on a 2D medium, to achieve this a specilized pipeline is required

- **Vertex Shader**: Processes each vertex individually, applying transformations such as translation, rotation, and scaling. It typically converts object-space coordinates to screen-space coordinates.
- **Tessellation**: Subdivides coarse geometry into finer pieces, allowing for smoother surfaces and more detailed models. This stage is optional and used mainly for complex surfaces.
- **Geometry Shader**: Operates on entire primitives (e.g., triangles), allowing for the creation or modification of geometry. It can add or discard

primitives dynamically.

- **Rasterizer**: Converts vector-based primitives into fragments (potential pixels). It determines which pixels are covered by each primitive and interpolates vertex attributes across the surface.
- **Pixel Shader**: Also known as the fragment shader, it computes the final color of each fragment based on lighting, texture, and other effects. The output is used to update the framebuffer.

After fragment processing, additional operations such as depth testing, blending, and stencil testing are performed before the final image is written to the screen. This entire pipeline is highly parallelized, allowing GPUs to render complex scenes efficiently.

0.2.0.1 Vertex shader

Modern GPUs break down 3D models into triangles. These triangles are the basic units processed by the pipeline. Each triangle is defined by three vertices, and vertex attributes (like position, color, normals) are interpolated across its surface during rasterization. Through matrix operations, the vertices can be moved and rotated to the desired position

0.2.0.2 Tessellation

In some more complex structures, the number of trianglese could be too low to have a satisfactory image, the tessellation stage improves the geometry by subdividing triangles, and can be applied selectively, to avoid imposing unnecessary workload on the GPU and optimizing performance

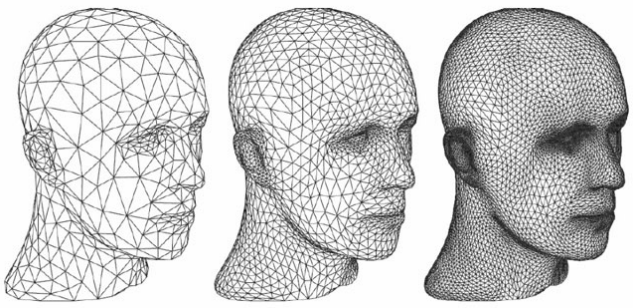

Figure 2: Illustration of tessellation in computer graphics (Source: computergraphics.stackexchange.com)

0.2.0.3 Geometry Shader

A Geometry Shader (GS) is a programmable stage in the graphics pipeline, written in GLSL, that processes entire primitives—such as points, lines, or triangles—after the vertex shader and before vertex post-processing. Unlike vertex shaders, which operate on individual vertices, geometry shaders take a whole primitive as input and can emit zero or more output primitives, allowing for dynamic geometry manipulation, main usages consist onehalfspacing

- Layered rendering: enabling a single primitive to be rendered across multiple layers or images without switching render targets.
- Transform feedback: capturing processed geometry data for reuse or computation, especially before the advent of compute shaders.

0.2.0.4 Rasterizer

Rasterization is the process of converting triangles in actual pixels, through the process of identifying which triangle is occupying most of the pixel. Precision can be increased using antialiasing, which consists of deciding the color of each pixel as the weighted average of the colors occupying the pixel

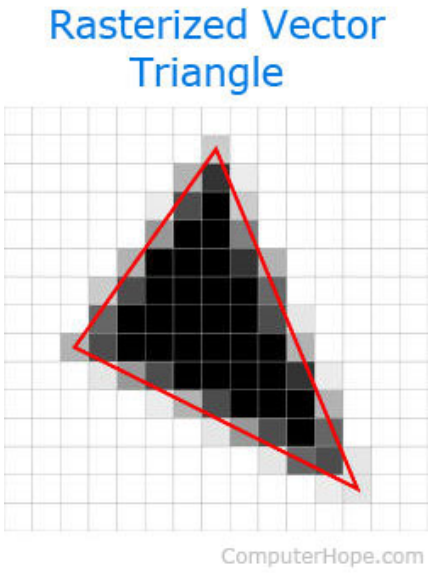

Figure 3: Illustration of rasterization(Source: https://www.computerhope.com/jargon/r/rasterize.htm)

0.2.1 Pixel Shader

The pixel shader, also known as a fragment shader in OpenGL terminology, is a programmable stage in the graphics pipeline responsible for determining the final color and other attributes of each pixel rendered to the screen. Operating after the rasterization stage, the pixel shader receives interpolated data from previous stages—such as vertex or geometry shaders—and executes user-defined code to compute per-pixel outputs like color, depth, and transparency. Unlike vertex or geometry shaders, which operate on geometric primitives, the pixel shader is focused entirely on screen-space fragments. It is typically written in shading languages such as GLSL (OpenGL), HLSL (DirectX), or Metal Shading Language (Apple). Its primary role includes:

- Applying lighting models (e.g., Phong, Blinn-Phong)
- Executing texture sampling and blending
- Implementing effects such as bump mapping, shadowing, and post-processing filters

Chapter 1

State of the Art

1.1 Register Transfer Level (RTL) Architecture

RTL design describes digital circuits in terms of data flow between registers and the logical operations performed on that data. It is the foundation for synthesizable hardware and is typically written in languages such as Verilog or VHDL.

In advanced designs, hardware description implementations may include dynamic linked list structures for managing queues. These structures use dedicated registers and RAM blocks to store data, pointers, and availability information. Control logic orchestrates memory allocation, pointer updates, and queue management.

Such architectures enable efficient buffer handling, scalable queue systems, and high-performance data processing. They are particularly useful in systems with constrained memory resources or complex transaction flows.

1.2 Structure of the Underlying RTL Implementation

The underlying RTL implementation is designed to manage dynamic buffer allocation efficiently using a hardware-based linked list mechanism. Each queue in the system is implemented as a linked list, with associated head-of-queue (HOQ), tail-of-queue (TOQ), and link-length (LL) registers. These registers track the start, end, and size of each queue respectively.

The system includes:

- **Buffer RAM**: Stores the actual data entries, typically in a wide format (e.g., 128 bits).
- **Link RAM**: stores the links between buffers, in each location is present the id to the next buffer in the cluster.
- **Control Logic**: Coordinates read/write operations, updates pointers, and manages queue lengths.

This dynamic allocation strategy avoids memory fragmentation and underutilization common in statically allocated systems. It allows queues to grow and shrink as needed, provided sufficient buffer space is available, and supports

scalable designs with more than 16 queues without compromising access performance. The number of entries per buffer can also be adjusted to optimize for memory usage, power and performance based on the desired trade off.

1.2.1 Limitations of Traditional FIFO Designs

Fixed-size FIFO implementations, while simple, suffer from several drawbacks in modern high-performance systems:

- **Memory Fragmentation:** Static allocation often leads to underutilized memory blocks, especially under variable traffic conditions.
- **Scalability Constraints:** Supporting a large number of FIFOs requires significant chip area and control logic.
- **Limited Flexibility:** Adapting to dynamic workloads or preemption scenarios is difficult without reconfiguration.

These limitations motivate the use of dynamic FIFO architectures, such as descriptor-based memory pools, which allow flexible allocation and deallocation of buffers.

1.2.2 Motivation for Behavioral Modeling

Behavioral modeling provides a high-level abstraction of system functionality, enabling faster simulation and easier debugging. In the context of dynamic virtual FIFOs, it allows:

- **Cycle-approximate simulation** for faster verification.
- **Isolation of protocol violations** without RTL complexity.
- **Autonomous context-aware testing** for preemption and save/restore scenarios.

This abstraction bridges the gap between conceptual design and RTL implementation, making it a valuable tool in early-stage verification.

1.3 Protocol Interfaces

Communication between components in hardware systems often relies on standardized signaling protocols. These typically include:

- `srdy` (source ready): Indicates that the sender has valid data to transmit.
- `rrdy` (receiver ready): Signals that the receiver is ready to accept data.
- `data`: The actual payload being transferred.

Control interfaces manage operations such as save/restore routines and context switching. During normal operation, transactions are queued and compared across interfaces. In case of context restoration, the system validates the restored state and issues diagnostic messages based on configuration settings.

1.4 Testbench Architecture and Execution Flow

Modern testbenches are built to support scalable, reusable, and modular verification environments. While the introduction outlines the foundational concepts of UVM, this section delves deeper into the architectural and operational aspects of testbenches.

1.4.1 Component Hierarchy and Communication

A UVM testbench is composed of layered components that communicate using transaction-level modeling (TLM) interfaces. These components include:

- **Sequencer**: Manages the flow of sequence items to the driver.
- **Driver**: Converts high-level transactions into pin-level activity on the DUT interface.
- **Monitor**: Observes DUT activity and extracts transactions for analysis.
- **Agent**: Encapsulates the sequencer, driver, and monitor, and exposes analysis ports.
- **Environment**: Instantiates agents and connects them to scoreboards and coverage collectors.

Communication between components is handled via TLM ports and exports, allowing decoupled and flexible data exchange.

1.4.2 Transaction Flow and Synchronization

The transaction flow begins with the generation of randomized or directed stimuli in sequences. These are passed to the sequencer, which schedules them for execution. The driver receives the sequence items and drives them onto the DUT interface. After execution, the monitor captures the DUT response and forwards it to analysis components.

Synchronization is managed through UVM phases and handshake mechanisms such as `start_item()`, `finish_item()`, and `item_done()`.

1.4.3 Scoreboarding and Reference Modeling

The scoreboard is central to functional verification. It compares DUT outputs against expected results generated by a reference model. This involves:

- **Predictor**: Simulates expected behavior based on input transactions.
- **Evaluator**: Matches actual outputs to predicted ones and flags mismatches.

Advanced scoreboards may support out-of-order matching, temporal checks, and error classification.

1.4.4 Coverage and Assertions

Verification completeness is measured using coverage metrics and assertions, which together provide both quantitative and qualitative insights into the design's correctness and robustness.

1.4.4.1 Functional Coverage

Functional coverage tracks whether specific scenarios, corner cases, or protocol behaviors have been exercised during simulation. It is implemented using SystemVerilog `covergroup` constructs, which define bins for each condition or value of interest.

Covergroups can be embedded in:

- **Sequence items**: To track stimulus generation patterns.
- **Monitors**: To observe DUT behavior and protocol compliance.

- **Scoreboards**: To verify that expected outputs are produced under specific conditions.

Example:

```
// Covergroup for input combinations
covergroup input_cov;
    coverpoint a {
        bins low = {0};
        bins high = {[1:$]};
    }
    coverpoint b {
        bins low = {0};
        bins high = {[1:$]};
    }
    cross a, b;
endgroup
```

Cross coverage is especially useful for ensuring that combinations of inputs are exercised, which is critical in designs with conditional logic or multiple control paths.

1.4.4.2 Code Coverage

Code coverage is collected using simulation tools and includes:

- **Line coverage**: Tracks which lines of RTL code were executed.
- **Branch coverage**: Verifies that all conditional branches were taken.
- **Toggle coverage**: Ensures that all bits in the design toggled during simulation.
- **FSM coverage**: Confirms that all states and transitions in finite state machines were visited.
- **Expression coverage**: tracks what percent of boolean expressions are executed during the simulation.

These metrics are typically reported by simulators and integrated into regression dashboards. They help identify dead code, untested logic, and areas requiring additional stimulus.

1.4.4.3 Assertions

Assertions are formal checks embedded in the RTL or testbench to validate design behavior. They are classified as:

- **Immediate assertions**: Checked at a specific simulation time.
- **Concurrent assertions**: Span multiple cycles and validate temporal relationships.
- **Assumptions**: Used in formal verification to constrain input behavior.
- **Coverage assertions**: Track whether specific conditions have occurred.

Example:

```
// Immediate assertion
assert property (@(posedge clk) (req && ack) |-> (ready));

// Concurrent assertion
property handshake;
    @(posedge clk) disable iff (!reset_n)
    req |-> ##[1:3] ack;
endproperty
assert property (handshake);
```

Assertions are critical for catching protocol violations, illegal states, and timing errors early in the simulation. They also serve as documentation for design intent and can be reused in formal verification environments.

1.4.4.4 Integration and Reporting

Coverage and assertion results are collected and analyzed using verification management tools. These tools provide:

- **Coverage reports**: Summarize functional and code coverage metrics.
- **Assertion dashboards**: Highlight passed, failed, and inactive assertions.
- **Gap analysis**: Identify untested areas and suggest additional test scenarios or points to be wqaived.

In regression environments, these metrics are used to track verification progress, prioritize bug fixes, and guide test development. Coverage closure is often a milestone in the verification lifecycle, indicating readiness for tape-out or formal review.

Behavioral models may be used to accelerate simulation and isolate issues before RTL is finalized.

1.5 Patent-Based Approaches to Dynamic FIFO Architectures

1.5.1 Hardware Implementation of N-Way Dynamic Linked Lists (US7035988B1)

The patent [1] presents a hardware-based solution for implementing multiple dynamic linked lists, primarily targeting queue management in constrained memory environments. The architecture comprises a register file with head-of-queue (HOQ), tail-of-queue (TOQ), and link-length (LL) registers for each queue. These registers facilitate tracking the start, end, and size of each linked list.

The data elements are stored in a buffer RAM, while the linkage between entries is maintained in a separate pointer RAM. A free pointer RAM manages the pool of available memory locations, accessed in a FIFO manner. Control logic orchestrates read/write operations, pointer updates, and dynamic allocation/deallocation of memory.

This design enables efficient memory usage by allowing queues to grow and shrink dynamically. It avoids the pitfalls of static allocation, such as fragmentation and underutilization. The architecture supports up to 16 queues, with scalability to larger numbers through parameterization. The separation of data and pointer storage enhances access speed and modularity.

1.5.1.1 Key Components

- **Register File:** Contains HOQ, TOQ, and LL registers for each queue.
- **Buffer RAM:** Stores the actual data entries.
- **Next Pointer RAM:** Maintains linkage between entries.

- **Free Pointer RAM:** Manages available memory locations.
- **Control Logic:** Coordinates operations and ensures consistency.

1.5.1.2 Advantages

- Dynamic allocation reduces memory waste.
- Hardware implementation ensures high performance.
- Modular design facilitates scalability.

1.5.2 Dynamic FIFO for Simulation (US7346483B2)

Patent [2] introduces a dynamic FIFO mechanism tailored for simulation environments. The primary goal is to optimize memory usage and prevent deadlocks during simulation of complex systems. The FIFO starts with a small initial size and expands incrementally based on traffic demand and wait periods.

The architecture supports autonomous resizing of the FIFO buffer when predefined wait periods expire without data consumption. This adaptive behavior ensures that simulation resources are used efficiently, avoiding unnecessary memory allocation while maintaining throughput.

The dynamic FIFO is particularly useful in scenarios involving multiple models with varying execution domains and timing behaviors. It facilitates integration across system-level, RTL, and gate-level simulations, supporting languages such as SystemC, Verilog, and VHDL.

1.5.2.1 Key Features

- **Wait Period Management:** FIFO size increases when wait periods expire.
- **Adaptive Sizing:** Balances memory usage and simulation speed.
- **Cross-Domain Compatibility:** Supports integration across different simulation levels.

1.5.2.2 Use Cases

- Simulation of graphics pipelines and compute-intensive systems.
- Verification of context switching and preemption mechanisms.
- Integration with UVM-based testbenches.

1.5.3 Multiple Virtual FIFO Arrangement (US5426639)

Patent [3] proposes a dynamic buffering architecture for packet switches, where each data source connected to a port circuit is assigned a virtual FIFO whose capacity can expand or contract based on traffic demands. The system partitions internal memory into a set of data buffers, some of which are statically assigned to channels while the rest form a pool of free buffers. When a buffer assigned to a channel becomes full, additional buffers are dynamically linked from the pool to extend its capacity. Conversely, when a buffer is emptied, it is returned to the pool. The architecture uses channel records to track the head and tail pointers of each virtual FIFO, and buffer link records to maintain the chaining of buffers. This enables the system to support a large number of sources (e.g., 512 channels) without requiring a fixed buffer per source, thereby reducing memory overhead and improving scalability. The dynamic linking mechanism is managed by sequencers and controllers that monitor buffer fullness and trigger allocation or deallocation accordingly. Importantly, the design avoids static allocation and supports variable service grades (latency, bandwidth, reliability), making it suitable for heterogeneous traffic environments. The virtual FIFO concept allows for efficient memory utilization and flexible adaptation to varying data rates, aligning well with behavioral models of dynamic FIFO systems.

1.5.4 System with Multiple Dynamically-Sized Logical FIFOs Sharing Single Memory (US6269413)

Patent [4] introduces a system architecture that enables multiple independent logical FIFO buffers to share a single memory structure, with dynamic allocation of storage capacity among them. The design eliminates the need for fixed-size, dedicated buffers per FIFO, thereby reducing hardware redundancy and optimizing memory usage. The system comprises:

- A main register file that stores both payload data and link data, forming a linked list for each logical FIFO.
- Separate write and read pointer register files, each maintaining pointers for every logical FIFO
- A free register identifier, which dynamically tracks available memory locations using one of three mechanisms: a priority encoder, a conventional FIFO buffer, or a dedicated logical FIFO.

Each FIFO entry includes a payload field and a link field. When data is enqueued, the system:

1. Selects the appropriate write pointer.
2. Stores the data in the payload field of the destination register.
3. Updates the link field and write pointer with the address of the next free registe

When data is dequeued, the system: selects the appropriate read pointer, retrieves the payload and updates the read pointer using the link field.

This architecture supports simultaneous and independent read/write operations, making it suitable for high-throughput environments. It also includes mechanisms for empty/full detection using counters and synchronization logic, especially useful when read and write operations occur in different clock domains. The system is highly scalable and adaptable, aligning well with behavioral models of dynamic virtual FIFOs. It provides a robust framework for managing multiple data streams efficiently, which is directly relevant to your thesis on memory pool modeling.

1.6 Comparative Analysis

The four patents under review address dynamic FIFO management across different domains—hardware implementation, simulation environments, and shared memory architectures. Each offers a unique strategy for handling buffer allocation, scalability, and integration.

- **US7035988B1** focuses on hardware-efficient linked list FIFO queues for embedded systems.

- **US7346483B2** introduces dynamic FIFO resizing for simulation models.
- **US5426639A** presents a virtual FIFO system with dynamic buffer linking for packet switches.
- **US6269413B1** describes a multi-FIFO system sharing a single memory with dynamic allocation and pointer management.

Feature	**US7035988B1**	**US7346483B2**	**US5426639A**	**US6269413B1**
Domain	Hardware	Simulation	Networking Hardware	Shared Memory FIFO
Allocation Strategy	Linked List	Incremental Growth	Dynamic Buffer Linking	Linked List per FIFO
Memory Management	Pointer RAM	Wait-Based Resizing	Buffer Pool with Linkage	Central Register File
Scalability	Fixed Queues (e.g., 16)	Dynamic	Up to 512 Channels	Arbitrary FIFO Count
Integration	RTL Designs	UVM, SystemC, Verilog	Packet Switches	RTL, SoC FIFO Controllers

Table 1.1: Comparison of Dynamic FIFO Patents

Relevance to Modern FIFO Architectures

These patents collectively inform the design of modern FIFO systems across hardware and simulation domains. The hardware-centric approaches of US7035988B1, US5426639A, and US6269413B1 provide scalable and memory-efficient solutions for embedded systems, packet switches, and shared memory architectures. Their use of linked lists and dynamic buffer chaining aligns with current trends in low-latency, high-throughput designs.

Meanwhile, US7346483B2 enhances simulation workflows by enabling adaptive FIFO sizing based on runtime behavior. This is particularly relevant for pre-silicon validation, regression testing, and mixed-abstraction modeling.

Together, these methodologies offer complementary perspectives on dynamic FIFO design. Their principles can be extended to support advanced features such as context-aware preemption, fault recovery, and multi-threaded

behavioral modeling—directly relevant to the memory pool abstraction in your thesis.

References

- US7035988B1: Marino, J. "Hardware Implementation of an N-Way Dynamic Linked List", Network Equipment Technologies, 2006.
- US7346483B2: Toma, H. et al. "Dynamic FIFO for Simulation", Synopsys Inc., 2008.
- US5426639A: Follett, D. et al. "Multiple Virtual FIFO Arrangement", AT&T Corp., 1995.
- US6269413B1: Sherlock, D. "System with Multiple Dynamically-Sized Logical FIFOs Sharing Single Memory", Hewlett-Packard, 2001.

1.7 Queuing Mechanisms on GPUs

1.7.1 The Broker Queue

Kerbl et al. [5] introduce the *Broker Queue* (BQ), a linearizable FIFO queue designed for fine-granular work distribution on GPUs. Traditional lock-free or blocking queues suffer from performance degradation or inflexibility under massive parallelism. BQ addresses these limitations by introducing a **broker mechanism** that mediates access to the queue via a shared counter (`Count`), enabling predictable and scalable behavior.

1.7.1.1 Design Principles

The Broker Queue is built upon the following principles:

- **Linearizability**: Ensures that concurrent operations appear atomic and ordered.
- **Static memory usage**: Avoids dynamic allocation, which is costly on GPUs.
- **Execution model compatibility**: Supports individual threads, warps, and cooperative thread groups.

- **Multi-queue support**: Enables advanced scheduling strategies such as work stealing.

1.7.1.2 Variants

The authors propose two variants of the Broker Queue:

- **Broker Work Distributor (BWD)**: Sacrifices linearizability for performance, suitable for work distribution scenarios.
- **Broker Stealing Queue (BSQ)**: Adds work stealing capabilities for dynamic task redistribution across multiple queues.

1.7.1.3 Performance Evaluation

In synthetic benchmarks and real-world workloads such as PageRank, BQ demonstrates superior performance compared to traditional lock-free and blocking queues. It achieves up to three orders of magnitude speedup and maintains low contention even under high thread concurrency.

1.7.1.4 Modeling Relevance

The Broker Queue's ring-buffer-based design, ticketing system, and assurance-based concurrency control provide a rich behavioral abstraction for modeling dynamic virtual FIFOs. These mechanisms are directly applicable to the simulation and analysis of queue state transitions and thread interactions in the proposed model.

1.8 Dynamic Memory Allocation on GPUs

1.8.1 Ouroboros: Virtualized Queues

Winter et al. [6] present *Ouroboros*, a virtualized queueing structure for dynamic memory allocation on GPUs. Unlike static allocators, Ouroboros supports multiple allocation sizes and dynamic growth/shrinkage of queues, making it highly adaptable to varying workloads.

1.8.1.1 Architectural Components

Ouroboros introduces two virtualized queue architectures:

- **Virtualized Array-Hierarchy Queue (VAQ)**: Uses a small pointer array to manage queue chunks.
- **Virtualized Linked-Chunk Queue (VLQ)**: Uses linked chunks with minimal static overhead.

It employs a **bulk semaphore** to efficiently manage concurrent access and allocation, and supports both **page-based** and **chunk-based** memory reuse.

1.8.1.2 Performance Evaluation

Ouroboros achieves:

- Speed-ups of 11× to 412× over CUDA's native allocator.
- Memory footprint reduction of up to 32× compared to faimGraph.
- Superior performance in real-world graph algorithms such as PageRank and Static Triangle Counting.

1.8.1.3 Modeling Relevance

The virtualized queue architecture and concurrency primitives in Ouroboros provide a layered behavioral abstraction for modeling dynamic memory pools. Its modular design—chunks, pages, and queues—aligns well with the hierarchical modeling approach used in this thesis.

1.9 Comparative Analysis

Table 1.2: Comparison of Broker Queue and Ouroboros

Feature	Broker Queue	Ouroboros
Design Focus	Work distribution	Memory allocation
Queue Type	Linearizable FIFO	Virtualized index queues
Concurrency Control	Broker + ticketing	Bulk semaphore
Memory Model	Static ring buffer	Dynamic chunk/page system
Variants	BWD, BSQ	VAQ, VLQ
Performance	Fastest linearizable GPU queue	Fastest dynamic allocator
Modeling Potential	Scheduling abstraction	Memory reuse abstraction

Both systems demonstrate how GPU-specific constraints—such as limited memory, high contention, and lack of fine-grained scheduling—can be addressed through innovative queueing and allocation strategies. Their design choices offer valuable insights for the behavioral modeling of dynamic virtual FIFOs, particularly in terms of concurrency control, memory reuse, and execution paradigms.

Chapter 2

Verification Environments

To ensure the correctness and robustness of the behavioral model, a comprehensive verification environment is developed using the Universal Verification Methodology (UVM). This environment not only validates the functional behavior of the model but also serves as a platform for Register Transfer Level (RTL) verification, enabling deeper insights into the architectural implementation. The verification framework is designed to support autonomous checking mechanisms, leveraging debug and peripheral signals to facilitate both functional and architectural validation.

2.1 Overview of Verification Strategy

The verification strategy is centered around modularity, scalability, and reusability. The environment is structured to accommodate multiple verification phases, including unit-level testing, integration testing, and system-level validation. Key components such as monitors, scoreboards, and coverage collectors are instantiated in a hierarchical manner to ensure thorough verification across all levels of abstraction.

The verification flow begins with the instantiation of the Device Under Test (DUT) within a UVM testbench. Stimuli are generated using configurable sequences that emulate realistic transaction patterns. These transactions are injected into the DUT through well-defined interfaces, and the responses are captured and analyzed using monitors and scoreboards.

2.2 Testbench Structure

The testbench is architected to differentiate between various interfaces connected to the DUT, primarily categorized into transaction interfaces and control interfaces.

2.2.1 Transaction Interface

The transaction interface is responsible for handling the data flow into and out of the DUT. It includes:

- Input transaction stream
- Output transaction stream

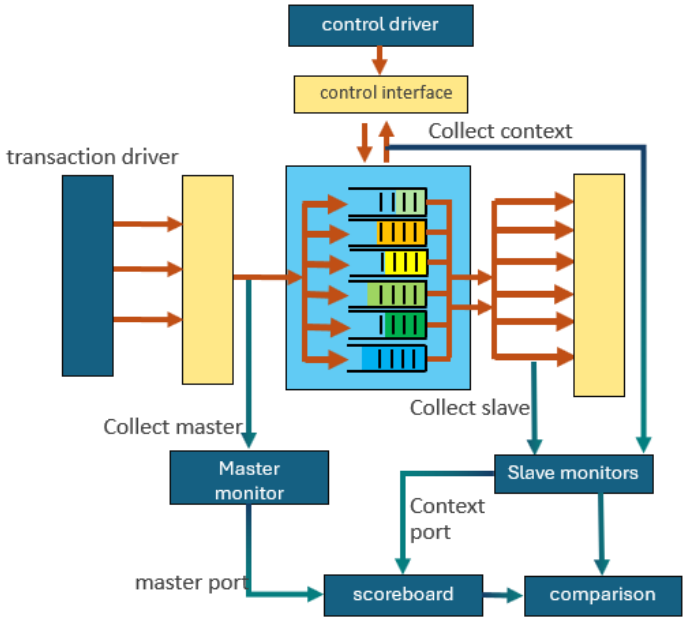

Figure 2.1: Testbench structure

- Handshake protocol signals

Each input and output cluster is equipped with a dedicated interface to ensure isolation and accurate tracking of data movement. The handshake protocol ensures synchronization between the DUT and the testbench, enabling precise control over transaction timing.

2.2.2 Control Interface

The control interface facilitates communication of control signals to the DUT. It encompasses:

- Initialization commands
- Debug access signals
- Internal state information

This interface plays a crucial role in context-aware operations such as save and restore, preemption, and fault recovery. It allows the testbench to manipulate the internal state of the DUT, thereby enabling advanced verification scenarios.

2.3 Monitors and Scoreboard

Monitors are instantiated for each interface to observe and capture the data transactions. They operate in a non-intrusive manner, ensuring that the DUT behavior remains unaffected. The monitors perform the following functions:

- Capture input and output transactions
- Detect context changes via debug signals
- Extract state information for reconstruction

The scoreboard is implemented using SystemVerilog queues and serves as the reference model for output verification. It maintains a record of expected transactions and compares them against the actual DUT outputs. The scoreboard operations include:

- Pushing incoming transactions to the back of the queue
- Popping transactions from the front upon request
- Resetting and reconstructing queues during context switches

This mechanism ensures that the DUT adheres to the expected behavior under various operational conditions, including context switching and fault recovery.

2.4 Test Scenarios

A diverse set of test scenarios is implemented to validate the DUT under different operating conditions. These scenarios are designed to uncover corner cases and timing-critical issues. The test scenarios include:

- **Random Test**: Sends randomized transactions through the DUT to validate general functionality.
- **Back Pressure Test**: Holds output transactions until clusters are full, then releases them to test flow control.
- **Halt Test**: Saves all registers, resets the DUT, and restores the state to verify halt and resume functionality.

- **Preemption Test**: Saves and restores only essential registers to simulate task preemption.
- **Ping-Pong Test**: Switches context and resumes the original context to test context switching mechanisms.
- **Back Pressure Preemption Test**: Combines back pressure and preemption to test complex scenarios.

Each scenario is parameterized to allow variations in transaction patterns, timing, and control signals. This flexibility enables exhaustive testing of the DUT under realistic conditions.

2.5 Advanced Verification Features

Inspired by industrial verification practices, the environment incorporates advanced features such as:

- **State Snapshotting**: Captures the internal state of the DUT at specific checkpoints for debugging and analysis.
- **Fault Injection**: Introduces faults into the DUT to test error handling and recovery mechanisms.
- **Randomized Context Switching**: Randomly switches contexts to validate robustness under unpredictable conditions.
- **Microcode Validation**: Integrates microcode checkers to detect errors in control logic execution.

These features enhance the coverage and reliability of the verification process, ensuring that the DUT meets stringent quality standards.

2.6 Summary

The verification environment provides a robust and flexible framework for validating the behavioral and architectural aspects of dynamic virtual FIFOs. Its modular design, support for context-aware operations, and advanced verification features make it suitable for comprehensive validation. Future enhancements will focus on increasing automation, improving coverage, and integrating with system-level testbenches to ensure complete verification of the DUT.

Chapter 3

behavioral model

3.1 Why We Use Behavioral Modelling in Digital Electronics and Computer Architecture

- **Introduction** Behavioral modelling is a high-level abstraction technique used to describe the functionality of digital systems without specifying their structural or timing details. Instead of detailing how a system is built, behavioral models focus on what the system does its response to inputs and its overall behavior over time. This modelling approach is particularly valuable in the early stages of design, simulation, and verification.

- **Accelerating Design and Simulation** One of the primary advantages of behavioral modelling is its ability to significantly accelerate simulation. Gate-level or RTL simulations of complex systems can be slow and resource-intensive. Behavioral models, operating at a higher level of abstraction, enable faster simulation cycles, allowing designers to iterate quickly and identify functional issues early in the development process.

- **Managing Complexity**Modern digital systems are composed of numerous interconnected components. Behavioral modelling helps manage this complexity by allowing designers to validate system functionality before delving into implementation details. This abstraction is especially useful when dealing with dynamic structures such as FIFOs, and linked lists, where functional correctness is critical.

- **Facilitating Debugging and Verification** Behavioral models are essential tools for debugging and verification. They serve as reference models that mirror the expected behavior of the system. During testing, discrepancies between the behavioral model and the actual implementation can highlight bugs or design flaws. Behavioral models also integrate seamlessly into testbenches, enabling validation of features like context switching, save/restore routines, and preemption logic.

- **Supporting Incremental Development** Behavioral modelling supports incremental development by allowing components to be prototyped and tested in isolation. This modular approach is particularly effective in environments where individual subsystems such as memory management units or control interfaces are developed independently. Once validated

behaviorally, these components can be integrated into the full system with confidence.

- **Cross-Domain Integration** Behavioral models act as bridges between different specification languages and execution domains. They facilitate integration across tools and languages such as SystemC, Verilog, VHDL, and C++, enabling a unified simulation environment. This flexibility allows teams to choose the most appropriate language for each subsystem while maintaining overall coherence.

- **Avoiding Deadlocks and Resource Waste** Fixed-size buffers and FIFOs can lead to deadlocks if not properly sized. Behavioral models, especially dynamic ones, adapt to runtime conditions, resizing and adjusting wait periods to prevent blocking and improve throughput. This adaptability informs hardware implementation decisions, ensuring efficient resource utilization.

- **Autonomous and Context-Aware Testing** Behavioral models enable autonomous testing environments where the system can detect and respond to events such as context switches or save/restore operations without external triggers. This capability is crucial for verifying complex behaviors like fault injection, preemption, and corner-case handling. Behavioral models simulate realistic scenarios, collect coverage data, and validate microcode logic all while maintaining abstraction and readability.

- **Cycle-Accuracy vs. Functional Fidelity** While behavioral models may lack cycle accuracy, they excel in functional fidelity. This trade-off is acceptable during early design and verification phases. Once the behavioral model is validated, it can be refined or replaced with more detailed RTL models for timing analysis and synthesis.

- **Conclusion** Behavioral modelling is a strategic tool in digital design. It enables faster simulation, effective debugging, complexity management, and efficient resource allocation. In the context of dynamic virtual FIFOs, behavioral modelling plays a central role in validating behavior, supporting autonomous test environments, and guiding hardware implementation. It bridges the gap between conceptual design and physical realization, making it indispensable in modern computer architecture.

3.2 Formalization of Tradeoffs in Design

In the design of dynamic virtual FIFOs, several parameters interact to determine performance, resource utilization, and implementation complexity. This section formalizes the tradeoffs between buffer depth, number of buffers, and total memory, providing a framework for evaluating design choices.

3.2.1 Parameter Definitions

Let the following variables define the system:

- B = buffer depth (number of registers per buffer)
- N = number of buffers
- R = register width (in bits)
- T = total memory used
- d = descriptor size (in bits)
- D = total descriptor memory
- U = utilization efficiency (fraction of memory actively used)
- L = latency per transaction (in cycles)
- F = number of FIFOs (clients + heap)

3.2.2 Memory Usage and Overhead

The total memory used for data storage is:

$$T = B \times N \times R \tag{3.1}$$

The descriptor memory overhead is:

$$D = N \times d \tag{3.2}$$

Utilization efficiency is defined as:

$$U = \frac{\text{Active Buffers}}{N} \tag{3.3}$$

3.2.3 Tradeoff Functions

3.2.3.1 Memory vs. Flexibility

In a fixed allocation scheme:

$$T_{\text{fixed}} = F \times B \times R \tag{3.4}$$

In a shared pool configuration:

$$T_{\text{shared}} = B \times N \times R \tag{3.5}$$

The difference in memory usage is:

$$\Delta T = T_{\text{shared}} - T_{\text{fixed}} \tag{3.6}$$

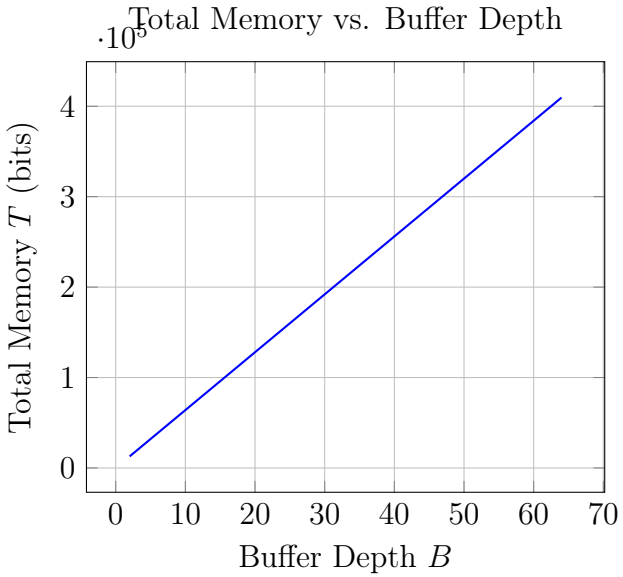

Figure 3.1: Total memory usage as a function of buffer depth

3.2.3.2 Descriptor RAM Overhead vs. Granularity

As buffer depth decreases, descriptor count increases:

$$D(B) = \frac{T}{B \cdot R} \times d \tag{3.7}$$

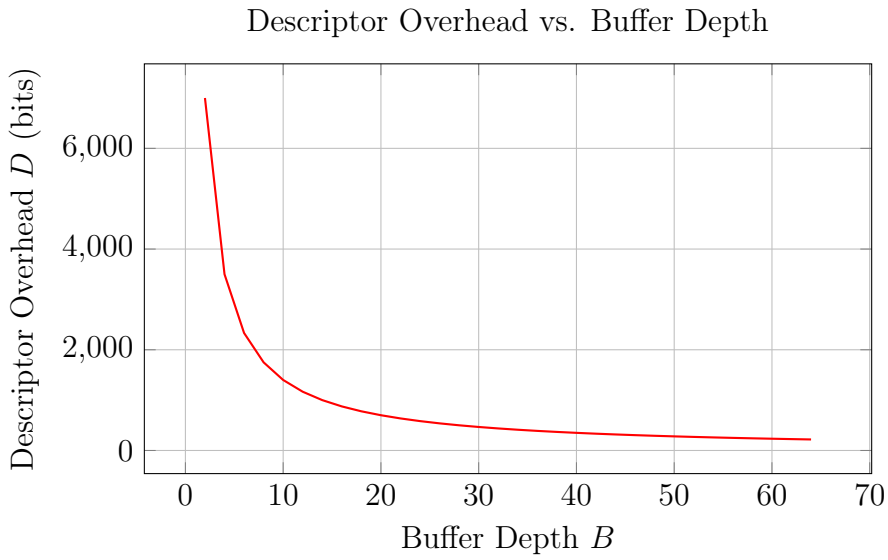

Figure 3.2: Descriptor overhead increases with finer granularity (smaller buffer depth), assuming $T = 100000$ bits and $d = 7$ bits per descriptor.

3.2.4 Cost Function

To evaluate configurations, we define a cost function:

$$C(B, N) = w_1 \cdot T + w_2 \cdot D + w_3 \cdot L + w_4 \cdot (1 - U) \tag{3.8}$$

where w_i are weights reflecting design priorities.

3.2.5 Optimization Problem

The design optimization can be posed as:

$$\begin{aligned} \text{Minimize:} \quad & C(B, N) && (3.9)\\ \text{Subject to:} \quad & T \leq T_{\max} && (3.10)\\ & D \leq D_{\max} && (3.11)\\ & L \leq L_{\text{target}} && (3.12)\\ & U \geq U_{\min} && (3.13) \end{aligned}$$

This formalization supports simulation-based exploration and guides implementation decisions.

3.3 Combined Cost Function Analysis

To evaluate the tradeoffs between buffer depth (B), number of buffers (N), and overall system performance, we define a combined cost function that aggregates key metrics into a single scalar value. This function helps identify optimal configurations for the design by balancing memory usage, descriptor overhead, latency, and utilization efficiency.

3.3.1 Cost Function Definition

The cost function is defined as:

$$C(B, N) = w_1 \cdot T + w_2 \cdot D + w_3 \cdot L + w_4 \cdot (1 - U) \tag{3.14}$$

where:

- $T = B \cdot N \cdot R$ is the total memory usage (with $R = 50$ bits per register),
- $D = N \cdot d$ is the descriptor overhead (with $d = 7$ bits per descriptor),
- $L(B) = L_0 - \alpha \cdot \log(B)$ is the latency model (with $L_0 = 20$, $\alpha = 3$),
- $U = \frac{N-32}{N}$ is a simplified utilization model assuming 32 buffers are idle,
- w_1, w_2, w_3, w_4 are weights reflecting design priorities.

In this example, we use:

$$w_1 = 1.0, \quad w_2 = 0.5, \quad w_3 = 0.8, \quad w_4 = 0.2$$

3.3.2 Interpretation

The cost function allows us to compare configurations and identify the "sweet spot" where memory usage, latency, and overhead are jointly optimized. A lower value of $C(B, N)$ indicates a more efficient configuration. The heatmap below visualizes this cost function across a range of buffer depths and buffer counts.

3.4 Implementation of the Behavioral Model

The behavioral model developed in this thesis is a comprehensive SystemVerilog-based simulation framework designed to emulate the dynamic behavior of virtual FIFO systems. It is structured to reflect the operational intricacies of clustered buffer management, transaction flow control, and state introspection. The model is highly parameterized, allowing for flexible configuration and scalability across different use cases and system sizes.

3.4.1 General Architecture and Parameterization

At the heart of the behavioral model lies a top-level module that orchestrates the interaction between clusters, buffers, and control interfaces. This module is defined with a rich set of parameters that govern its structural and functional behavior. These parameters include the number of clusters (virtual FIFOs), the total depth of the system (i.e., the number of transaction entries), the size of each buffer (number of transactions per buffer), and the width of each transaction (bit-width). Additional parameters define the depth of internal FIFOs used for buffer allocation and deallocation, the size of the debug memory, and the width of address and data buses used for save and restore operations.

The model calculates several derived parameters, such as the number of buffers (computed as the total depth divided by buffer size), and the number of bits required to index buffers and lines within buffers. These derived values are used throughout the model to ensure consistent addressing and indexing.

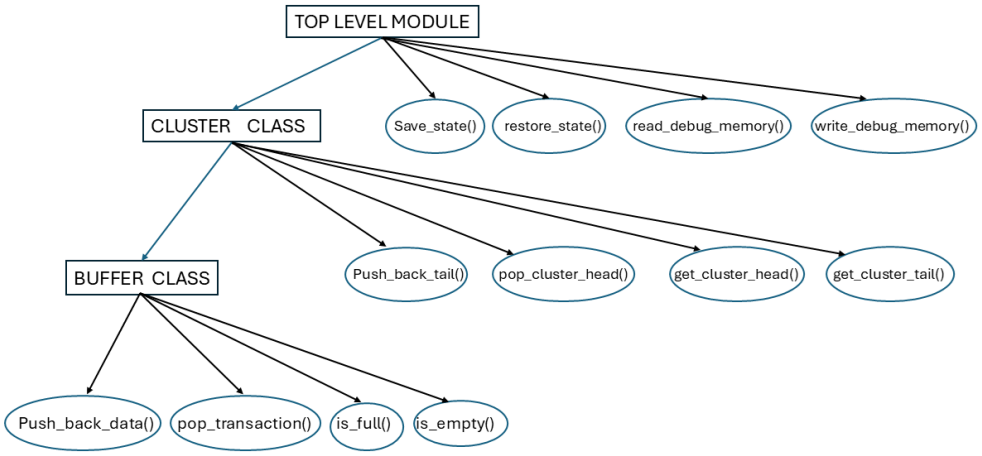

Figure 3.3: Class hierarchy and method relationships in the behavioral model. The top-level module instantiates clusters, which contain buffers. Each class exposes methods for transaction management, state queries, and debug operations.

3.4.2 Cluster and Buffer Composition

Each cluster in the model is instantiated as an object that contains a dynamic array of buffer objects. These buffers are responsible for storing transactions and maintaining internal pointers that track the head and tail positions. The cluster objects expose a variety of utility functions that allow external modules to query their state, including functions to retrieve the buffer IDs and line indices of the head and tail, the total number of active buffers, and encoded representations of the cluster head and tail positions.

Buffers are initialized with default values and are equipped with mechanisms to determine whether they are full or empty. The head pointer indicates the next transaction to be read, while the tail pointer indicates the next location to write a transaction. When a buffer becomes full, it signals the need for a new buffer to be allocated. Conversely, when a buffer becomes empty after a transaction is read, it is returned to a shared heap for reuse.

3.4.3 Initialization and Heap Management

During the initialization phase, each cluster is assigned a single buffer, and the remaining buffers are stored in a shared heap. This heap acts as a reservoir

from which clusters can draw additional buffers as needed. The initialization process involves creating buffer instances, assigning them unique identifiers, and distributing them between clusters and the heap. This setup ensures that each cluster starts with a minimal allocation and can dynamically expand its storage capacity based on workload demands.

The model includes logic to determine whether all clusters and the heap are empty, which is used to control reset behavior and stabilization phases. This logic iterates through all buffers in all clusters and the heap, checking for the presence of transactions.

3.4.4 Transaction Flow Control

Transactions are inserted into clusters through a controlled interface that checks for readiness and stabilization conditions. When a transaction is ready to be inserted, it is directed to the tail buffer of the target cluster. If the tail buffer is full, a new buffer is allocated from the heap and appended to the cluster. This dynamic allocation mechanism allows the system to adapt to varying workloads and ensures that clusters can grow as needed.

Transactions are extracted from the head of each cluster. The model checks whether the head buffer contains any valid transactions. If it does, the transaction is retrieved and the head pointer is advanced. If the buffer becomes empty after the extraction, it is returned to the heap for reuse. This recycling of buffers ensures efficient memory utilization and prevents resource leakage.

3.4.5 Debug Interface and Save/Restore Protocol

The behavioral model includes a comprehensive debug interface that supports both read and write operations to match current RTL design. This interface is used to capture the internal state of the model for diagnostic or checkpointing purposes. The save and restore process follows a multi-step protocol:

1. Enable buffer reservation and allow upstream data to be written.
2. Activate stabilization to drain fetch and return FIFOs.
3. Wait for the system to signal that stabilization is complete.
4. For save or debug read operations, set the address and initiate a read request. The model responds with the corresponding data.

5. Apply and release reset to prepare for restore.

6. For restore operations, set the address and initiate a write request. The model acknowledges the write.

7. Disable stabilization and buffer reservation to resume normal operation.

The debug memory is populated with data from all clusters and the heap, including transaction contents, buffer link information, and cluster metadata. This data is indexed and stored in a structured format that allows for efficient retrieval and analysis.

3.4.6 Concurrency and Synchronization Mechanisms

To manage concurrent access and ensure data consistency, the model employs a combination of clocked and combinational logic. Read and write operations are gated by readiness signals, and stabilization flags are used to control the flow of transactions during critical operations such as save and restore. This design ensures that the model behaves predictably under concurrent access scenarios and maintains the integrity of stored transactions.

The model also includes mechanisms to pulse acknowledgment signals in response to read and write requests. These pulses are synchronized with the system clock and ensure that external modules receive timely and accurate feedback.

3.4.7 Design Flexibility and Performance Trade-offs

The behavioral model is designed to support optional features such as preemption and debug read access. When preemption is disabled, the model operates at full throughput. When enabled, it introduces additional control logic that may reduce performance if only a single cluster is active. This trade-off allows the model to balance flexibility and efficiency based on the specific requirements of the system.

The model also supports slice-based configurations, which allow for partitioning of resources and more granular control over buffer allocation and transaction routing.

3.4.8 Conclusion

In summary, the behavioral model provides a robust, scalable, and highly configurable framework for simulating dynamic virtual FIFO systems. Its modular architecture, dynamic buffer management, and comprehensive debug capabilities make it suitable for integration into complex RTL environments. The design reflects a deep understanding of hardware modeling principles and offers a practical solution for managing shared resources in clustered architectures. By abstracting the complexities of buffer allocation and transaction flow, the model enables efficient simulation and verification of systems that rely on dynamic resource sharing and virtualized data paths.

Chapter 4

Results

4.1 Overview

This chapter presents the results obtained from the behavioral model simulations and verification environment. The focus is on functional correctness, coverage metrics, performance profiling, and memory utilization.

4.2 Code Coverage

To assess the thoroughness of the verification environment, code coverage metrics were collected across multiple dimensions. The behavioral model was subjected to randomized and directed test scenarios to ensure broad functional exercise.

- Line coverage: `90%`
- Branch coverage: `95%`
- FSM coverage: `100%`

The results demonstrate strong coverage across line, branch, and FSM dimensions, indicating that the testbench effectively stimulates the design under test and validates key functional paths.

4.3 Simulation Performance

Metric	**Behavioral**	**RTL**
Cycles simulated	100,000,000	20,791,000
Total CPU Time (relative)	4.25	3.06
Memory Usage (relative)	4.98	5.78

Table 4.1: Summary of simulation profiling metrics for random test Behavioral and RTL models

The simulation profiling data reveals distinct behavioral patterns between the two models. The Behavioral model tends to allocate more memory to components such as `transaction_item` and `uvm_event`, which suggests a higher degree of dynamic object creation or reuse. This behavior is consistent with

Metric	**Behavioral**	**RTL**
Cycles simulated	228,750,000	145,931,000
Total CPU Time (relative)	4.31	3.99
Memory Usage (relative)	3.0	3.0

Table 4.2: Summary of simulation profiling metrics for ping pong test in Behavioral and RTL models

Metric	**Behavioral**	**RTL**
Cycles simulated	20,911,000	20,891,000
Total CPU Time (relative)	3.74	3.49
Memory Usage (relative)	4.0	3.0

Table 4.3: Summary of simulation profiling metrics for back pressure test in Behavioral and RTL models

its architectural abstraction, where flexibility and modularity often come at the cost of increased memory overhead.

In contrast, the RTL model exhibits significantly higher memory usage in constructs like `Event`. This could be attributed to more frequent event triggering or less efficient memory reuse mechanisms. The observed count differences (such as a delta of -750) may indicate structural or functional disparities in how resources are instantiated and managed across the two models.

From a performance standpoint, the Behavioral model simulates substantially more cycles in the random test scenario, but this comes with increased CPU time and slightly reduced memory usage compared to RTL. In the ping pong test, both models demonstrate similar memory footprints, although the Behavioral model again simulates more cycles. The back pressure test shows near-identical cycle counts, with the RTL model being marginally more efficient in both CPU time and memory consumption.

These findings underscore the trade-offs between abstraction and efficiency, and highlight the importance of profiling in guiding model optimization and architectural decisions.

- Simulation time (RTL vs Behavioral)
- Speed-up factor
- Profiling insights (e.g., bottlenecks, memory usage)

4.4 Summary

The behavioral model achieved high verification completeness, with line, branch, and FSM coverage all above 90%. Simulation profiling across random, ping pong, and back pressure tests revealed that the behavioral model consistently simulated more cycles than RTL, with trade-offs in CPU time and memory usage. Profiling insights highlighted architectural differences in memory allocation and event handling. Overall, the results confirm the model's functional accuracy and its suitability for early-stage validation and architectural exploration.

Chapter 5

Conclusions

5.1 Summary of Contributions

This thesis presents a robust and multi-layered methodology for modeling and verifying dynamic virtual FIFO systems. A behavioral model was meticulouslydeveloped to encapsulate the functional dynamics of virtual FIFOs, including buffer allocation, transaction flow, and cluster-based data management. The model supports granular control over buffer states such as head/tail indexing, fullness, and emptiness enabling precise simulation of real-world scenarios. The model integrates a modular structure that allows dynamic configuration of clusters, buffers, and transaction pipelines. This flexibility supports both throughput optimization and preemption/debug-read capabilities, depending on buffer sizing and operational mode. A UVM-based verification environment was designed and implemented to rigorously test the model. This environment includes simulation mechanisms that track transaction integrity, buffer transitions, and cluster interactions. Coverage analysis was performed across multiple dimensions functional, structural, and temporal to ensure exhaustive validation. Additionally, the model incorporates debug and save/restore interfaces, allowing for introspection and state recovery. These features are critical for validating system behavior under reset conditions and during preemption events, further enhancing the reliability and testability of the design.

5.2 Limitations

While the behavioral model offers flexibility and abstraction, it does not guarantee cycle accuracy. Certain trade-offs were made to prioritize functional fidelity over timing precision, which may limit its applicability in low-level performance tuning. Additionally, the scope of test scenarios was constrained to representative cases such as random, ping pong, and back pressure tests, leaving room for broader stress testing and corner-case exploration.

5.3 Future Work

Several directions can extend the current work. Integrating the behavioral model with system-level testbenches would enable end-to-end validation in more realistic scenarios. Formal verification techniques could be applied to strengthen correctness guarantees, while synthesis experiments may help assess hardware feasibility. Finally, extending support for multi-threaded or hierarchical FIFO systems would broaden the model's applicability to more complex architectures.

List of Figures

List of Tables

Chapter 6

Acronyms

UVM Universal Verification Methodology

FIFO First In First Out

GPU Graphical Processing Unit

CPU Central Processing Unit

DUT Device Under Test

RTL Register Transfer Level

Bibliography

[1] J. Marino. Hardware implementation of an n-way dynamic linked list. US Patent US7035988B1, Network Equipment Technologies, 2006.

[2] H. Toma et al. Dynamic fifo for simulation. US Patent US7346483B2, Synopsys Inc., 2008.

[3] D. Follett et al. Multiple virtual fifo arrangement. US Patent US5426639A, AT&T Corp., 1995.

[4] D. Sherlock. System with multiple dynamically-sized logical fifos sharing single memory. US Patent US6269413B1, Hewlett-Packard, 2001.

[5] Bernhard Kerbl, Michael Kenzel, Johannes H. Mueller, Dieter Schmalstieg, and Markus Steinberger. The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 56–66. ACM, 2018.

[6] Matthias Winter, Daniel Mlakar, Michael Parger, and Markus Steinberger. Ouroboros: Virtualized queues for dynamic memory management on gpus. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 1–11. ACM, 2020.