Functional Verification of SPI Controller using UVM

Yanqian Yu
ya6212yu-s@student.lu.se

March 25, 2026

Master's thesis work carried out at
Electrical and Information Technology,
Faculty of Engineering LTH | Lund University.

Company Supervisor: Adem Gül, adem@beammwave.com
Faculty Supervisor: Baktash Behmanesh, baktash.behmanesh@eit.lth.se
Examiner: Pietro Andreani, pietro.andreani@eit.lth.se

Abstract

Radio Frequency Integrated Circuits (RFICs) play a crucial role in modern wireless communication systems. However, they can achieve high performance only when their internal registers are dynamically and precisely configured. To maintain reliable control while operating in complex multi-antenna environments, several interface techniques have been developed, among which the Serial Peripheral Interface (SPI) control module is one of the most critical bridges for translating system bus requests into SPI transactions.

In this thesis, we designed and analyzed a comprehensive Universal Verification Methodology (UVM) environment for an SPI_CTRL module consisting of a dual block random access memory(BRAM) architecture and eight independent SPI channels. This work began with the development of a verification architecture to ensure the testbench effectively stimulates the design's internal request FIFO and state machines. Based on this design, we evaluated its behavior across various clock domain crossing (CDC) conditions and continuous read/write requests, gaining insight into its scheduling performance and functional completeness.

This work evaluates the impact of concurrent high-speed AXI4-Lite transactions on the SPI_CTRL's stability. With the increasing demand for high integration and flexibility in wireless systems, handling massive parallel data translation has become an essential requirement. However, dynamic operating conditions and varying request timings introduce complex corner cases that challenge the adaptability and reliability of traditional directed testing methods.

This thesis analyzes the functional coverage of the SPI_CTRL design under varying request traffic and checks if all logical anomalies and timing violations can be eliminated by employing the UVM framework.

Keywords: Universal Verification Methodology (UVM); Serial Peripheral Interface (SPI); Advanced eXtensible Interface (AXI); Functional Verification; Constrained-Random Testing; Dual-BRAM Architecture.

Popular Science Summary

Have you ever wondered why your smartphone connection sometimes drops entirely when you walk behind a building or enter a crowded train station? Modern wireless communication, especially with 5G networks, relies on an incredibly fast and complex coordination between the digital "brain" of a device and its physical antennas. The antennas must constantly adjust their focus and frequencies in fractions of a millisecond to maintain a strong signal.

Inside the microchips that power these devices, there is a dedicated component responsible for this precise coordination. It is called the SPI Controller. You can think of it as a highly stressed translator and traffic dispatcher. The digital baseband processor speaks a fast, parallel language and sends massive amounts of configuration instructions. The analog antennas, however, require these instructions to be delivered slowly, one bit at a time, and at very specific moments in the future.

The SPI Controller must absorb this chaotic digital traffic, store it, and then execute the commands with absolute timing precision. If this controller makes a single mistake—for instance, dropping a command when the queue is too full or sending a signal a microsecond too early—the antennas will be configured to the wrong frequency. The result is an immediate loss of the wireless link. Given how critical this component is, engineers cannot afford to discover bugs after the chip is manufactured. Fixing a physical silicon chip costs millions of dollars and takes months. This is where functional verification comes in. The goal is to prove the design is flawless while it still exists only as software code on a computer screen.

However, traditional testing methods are no longer sufficient. Writing a few specific tests by hand is like checking a new highway by driving one car down the middle lane. It tells you very little about how the system handles a chaotic rush hour.

In this thesis, we took a radically different approach to ensure the SPI Controller was bulletproof. Using the Universal Verification Methodology (UVM), we built an automated, intelligent stress-testing environment. Instead of manually writing tests, we programmed a software engine to generate thousands of random, extreme traffic scenarios. We bombarded the controller with back-to-back requests, forced its internal clocks to operate at their absolute limits, and intentionally tried to crash its scheduling logic.

This aggressive approach successfully exposed several deeply hidden design flaws that traditional methods missed. After identifying and fixing these issues, our automated platform mathematically proved that every critical part of the controller's logic had been thoroughly tested under maximum stress. The result will be a highly reliable digital-to-analog bridge, ensuring that the next generation of wireless devices can maintain stable connections even in the most demanding environments.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor at BeammWave, Adem Gül, for his invaluable guidance, continuous support, and immense patience throughout the course of this Master's thesis.

I am also incredibly grateful to Jinzhuo Wu; without your help, it would not have been possible for me to conduct my thesis work at such an amazing company. I would also like to extend my great thanks to everyone at BeammWave. The welcoming and innovative atmosphere of the company truly left a lasting impression on me.

My sincere thanks also go to my university supervisor, Baktash Behmanesh, and my examiner, Pietro Andreani, for their academic support, constructive feedback, and administrative efforts in facilitating this project.

Last but certainly not least, I dedicate this thesis to my family. To my parents and grandparents, thank you for your unconditional love, unwavering encouragement, and the countless sacrifices that made my education possible. To my partner, Ziye Zhang, thank you for standing by me through the long nights and stressful deadlines. Your unwavering belief in me has been my greatest source of strength.

Yanqian Yu
Lund, Sweden
March 25, 2026

Contents

Chapter 1

Introduction

1.1 Research Background and Significance

The rapid evolution of next-generation wireless communication networks, such as 5G-Advanced and 6G, has driven an increasing demand for highly integrated and dynamically reconfigurable communication systems. Within these modern System-on-Chip (SoC) architectures, RFICs serve as the fundamental front-end components responsible for transmitting and receiving high-frequency signals. To support complex multi-antenna arrays and flexible frequency band switching, the digital baseband system must frequently and precisely configure a vast array of internal registers within the RFIC.

In this architectural context, the SPI_CTRL module acts as a critical communication bridge between the primary digital system and the external RF components. The module is explicitly responsible for programming the RFIC through the SPI interface. These programming operations are entirely dependent on requests that arrive via the AXI4 Lite interface and are subsequently queued within the module. The operational complexity of this module is significant, as it must handle high-speed parallel data from the system bus and translate it into precise serial timing sequences. To manage this data flow efficiently, the module incorporates a dual-request buffer capable of queuing and processing two concurrent requests.

The functional correctness of the SPI_CTRL module is directly tied to the overall viability of the communication chip. Any logical defect such as an unhandled FIFO overflow, asynchronous CDC metastability, or incorrect state machine transitions during concurrent access could lead to erroneous RFIC configurations, ultimately causing critical radio link failures. Consequently, performing rigorous verification on the SPI_CTRL module prior to tape-out is essential to prevent costly silicon respins.

1.2 Current Research Status and Development Trends of Verification Technologies

As the integration density and functional complexity of Very Large-Scale Integration (VLSI) designs grow exponentially, functional verification has emerged as the most significant bottleneck in the chip development lifecycle, often consuming upwards of 70% of total project resources. Traditional verification methodologies, which predominantly rely on Directed Testing written in hardware description languages (HDLs) like Verilog or VHDL, are no longer sufficient. These legacy approaches struggle to adequately stimulate the massive state spaces of modern SoCs and are highly susceptible to missing critical corner cases.

To address these escalating challenges, the semiconductor industry has shifted toward advanced verification methodologies, culminating in the widespread adoption of the UVM. Built upon the SystemVerilog language, UVM introduces Object-Oriented Programming (OOP) paradigms into the hardware verification domain. It standardizes the development of highly reusable, scalable, and modular testbench environments. Key innovations such as Constrained Random Testing (CRT) allow verification engineers to generate massive volumes of valid, yet unpredictable, stimulus combinations, effectively exposing hidden bugs. Furthermore, Coverage-Driven Verification (CDV) provides quantifiable metrics to objectively measure verification completeness, ensuring that all functional specifications and corner cases are rigorously exercised. For an interface-heavy and state-dependent design like the SPI_CTRL module, leveraging UVM is the optimal strategy to ensure robust and exhaustive verification.

1.3 Main Research Contents and Thesis Organization

This thesis focuses on the functional verification of the SPI_CTRL module within a contemporary wireless communication SoC. By utilizing UVM, a comprehensive, scalable, and automated verification environment is designed and implemented from the ground up. The primary research contents and the organization of the chapters are outlined as follows:

- **Chapter 1: Introduction.** This chapter introduces the research background, highlights the critical importance of RFIC configuration modules, reviews the evolution of VLSI verification methodologies, and outlines the thesis structure.
- **Chapter 2: Verification Protocols and Methodological Foundations.** This chapter provides an in-depth theoretical analysis of the standard protocols utilized in the design, specifically the AXI4/AXI4-Lite bus architectures and the SPI. It also introduces the core mechanisms and advantages of UVM.
- **Chapter 3: Functional Analysis of the SPI_CTRL Module.** This chapter dissects the Device Under Test (DUT) based on the design specification. It details the dual-BRAM storage architecture, the AXI transaction request handling logic, the FIFO scheduling mechanisms, and the state transitions of the core SPI controller.

- **Chapter 4: UVM Verification Platform Architecture Design.** This chapter elaborates on the topological structure of the UVM testbench. It details the extraction of verification features and the implementation of key components, including the AXI/SPI interface Agents, the Reference Model, and the Scoreboard.
- **Chapter 5: Design Flaw and Coverage Analysis.** This chapter presents typical bugs uncovered by the verification platform. It also establishes the functional coverage model and analyzes the code coverage convergence results.
- **Chapter 6: Conclusion and Future Work.** The final chapter summarizes the engineering achievements of the thesis, evaluates the performance of the UVM testbench, and proposes potential directions for future optimization.

Chapter 2

Verification Protocols and Methodological Foundations

2.1 Serial Peripheral Interface Protocol and System Integration

2.1.1 Overview of the 4-Wire SPI Architecture

SPI is a synchronous, full-duplex serial communication protocol that operates on a master-slave paradigm **[3]**. It is widely adopted for short-distance, high-speed data transfer between a master microcontroller (in this case, the digital baseband system) and peripheral integrated circuits, such as the target RFICs.

A standard SPI bus utilizes a four-wire architecture to facilitate continuous data transmission without the overhead of complex addressing mechanisms. The four fundamental signals include:

- **SCLK (Serial Clock):** The synchronous clock signal generated by the master to control the data transmission rate.
- **MOSI (Master Out Slave In):** The unidirectional data line used by the master to transmit data to the slave.
- **MISO (Master In Slave Out):** The unidirectional data line used by the slave to send data back to the master.
- **SS/CS (Slave Select / Chip Select):** An active-low control signal driven by the master to select a specific slave device on the bus, initiating and terminating the transaction.

The standard point-to-point connection topology between a single SPI master and a slave device is illustrated in Figure 2.1.

2.1.2 Clock Polarity and Phase)

The flexibility of the SPI protocol lies in its configurable clock timing, which is governed by two primary parameters: Clock Polarity (CPOL) and Clock Phase (CPHA). Together, they define four distinct operational modes (Mode 0 through Mode 3).

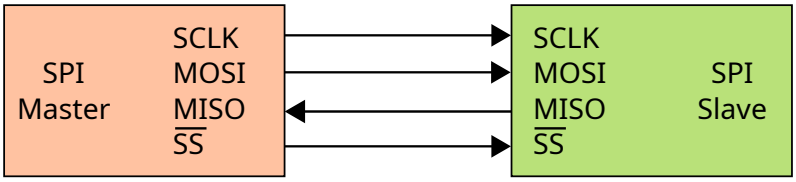


Figure 2.1: Communication Between Single SPI Master and Slave

In this verification project, the targeted RFIC operates under Mode 0 parameters (CPOL = 0, CPHA = 0). This configuration dictates specific timing constraints:

- **CPOL = 0:** The base value of the SCLK remains at logic low (0) when the bus is in an idle state.
- **CPHA = 0:** Data is captured (sampled) on the leading edge of the clock cycle, which corresponds to the rising edge when CPOL is 0. Conversely, data is shifted out (updated) on the trailing edge, which is the falling edge of the clock.

The fundamental timing relationships and data sampling edges under different CPOL and CPHA configurations are demonstrated in Figure 2.2.

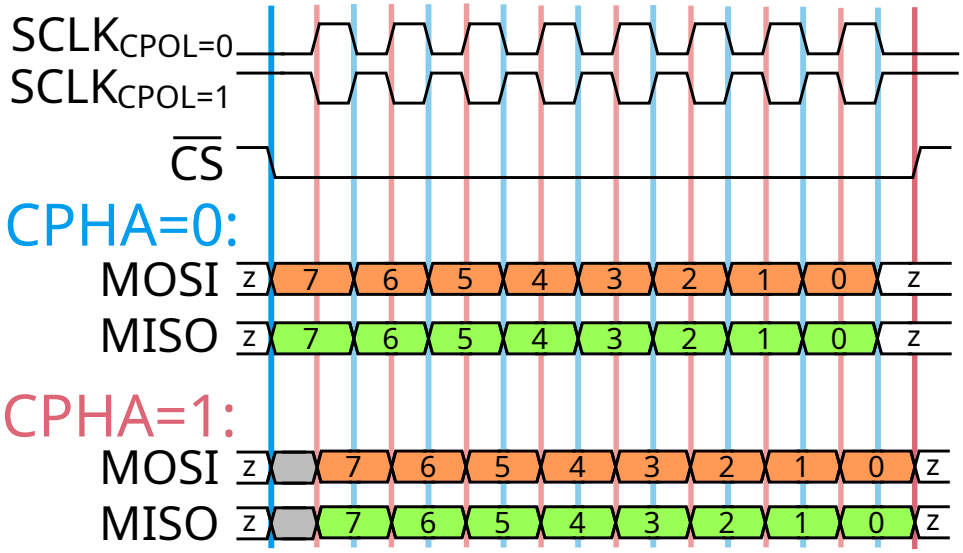


Figure 2.2: Example of SPI Mode 1 Waveform

2.1.3 Custom SPI Implementation in the SPI_CTRL Module

While adhering to the fundamental SPI mechanics, the SPI_CTRL module is heavily customized to support the concurrent programming of complex multi-antenna systems. The module provides NOF_ANTENNAS independent SPI lane outputs. Each lane comprises its own dedicated

MOSI, MISO, SCLK, and SS signals, allowing each lane to be individually addressable for concurrent per-device control. To accommodate this requirement, the standard architecture is expanded into a multi-slave configuration, as explicitly depicted in Figure 2.3.

According to the design specifications, the timing characteristics of the SPI_CTRL module implement strict edge-triggered updates:

- **SS Generation:** The Slave Select signal is active-low and remains continuously asserted for the entire duration of the transaction.
- **Clock Derivation:** The SCLK has a configurable frequency that is derived dynamically from the primary system clock.
- **Data Synchronization:** During a transaction, the MOSI signal is updated on the falling SCLK edges. Simultaneously, the MISO signal returning from the RFIC is sampled on the falling SCLK edges. All signals crossing clock domains utilize synchronizers or FIFOs to mitigate metastability.
- **Transaction Flow:** For a write transaction, the MOSI lane transmits the register address followed immediately by the data payload. For a read transaction, the address is sent over the MOSI lane, and because the RFICs use 16-bit shift registers, the actual read value is received over the MISO lane in the subsequent transaction period.

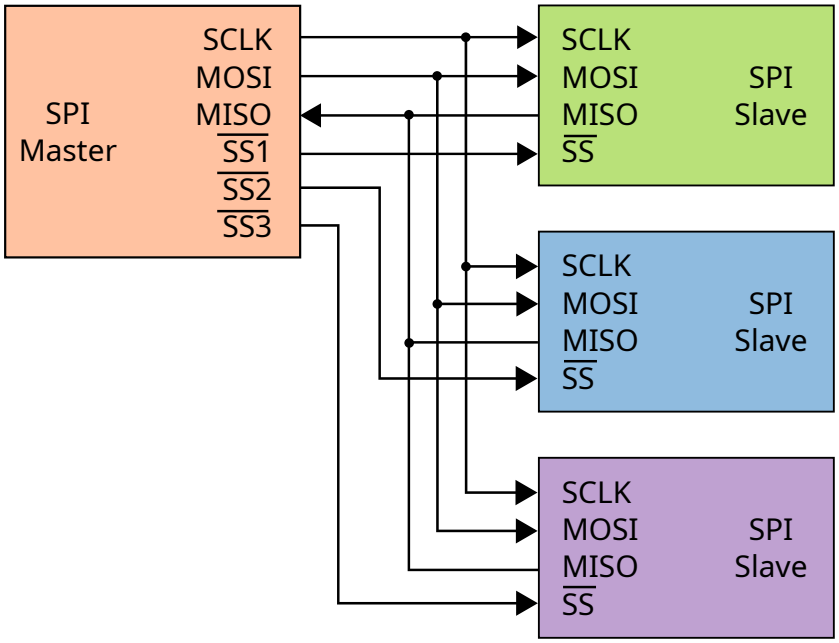


Figure 2.3: Communication Between Single SPI Master and Multiple Slave

2.2 Advanced eXtensible Interface (AXI) Protocol and System Integration

2.2.1 Introduction to AMBA AXI4 Protocol

The Arm Advanced Microcontroller Bus Architecture, or AMBA, is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in SoC designs.

The AXI protocol is engineered to support high-performance, high-frequency system designs **[2]**. It achieves this by employing a point-to-point, unidirectional channel architecture that strictly separates address/control phases from data phases.

The AXI4 specification defines five independent, unidirectional channels:

- **Write Address Channel (AW):** Transmits the target memory address and control information for a write transaction.
- **Write Data Channel (W):** Transfers the actual payload data from the master to the slave.
- **Write Response Channel (B):** Allows the slave to signal the completion and status (e.g., success or error) of the write transaction back to the master.
- **Read Address Channel (AR):** Transmits the target memory address and control information for a read transaction.
- **Read Data Channel (R):** Transfers the requested data from the slave back to the master, along with the read response status.

A fundamental characteristic of all AXI channels is the VALID/READY handshake mechanism. The source asserts the VALID signal to indicate that the address, data, or control information is available. Simultaneously, the destination asserts the READY signal to indicate it can accept the information. The actual transfer of information strictly occurs on the rising clock edge where both VALID and READY are asserted simultaneously, ensuring highly reliable, asynchronous-friendly data flow.

2.2.2 Dual-Bus Architecture: AXI4-Full vs. AXI4-Lite

To optimize both bandwidth and logic footprint, the AMBA specification defines different tiers of the AXI protocol. The SPI_CTRL module integrates a hybrid dual-bus architecture, leveraging two specific AXI4 variations to strictly decouple the data plane from the control plane:

- **AXI4-Full (The Data Plane):** This version supports high-performance burst transactions, allowing a single address phase to be followed by multiple data transfers (up to 256 beats). In the SPI_CTRL architecture, the AXI4-Full interface is exclusively utilized to efficiently stream massive RFIC configuration payloads directly into the internal Asynchronous Dual-Port RAM (DPRAM), maximizing memory write throughput.

- **AXI4-Lite (The Control Plane):** This is a simplified, lightweight subset of the AXI4 protocol. It strips away burst capabilities, supporting only single-beat transactions (typically 32-bit or 64-bit wide). The SPI_CTRL module employs the AXI4-Lite interface for configuring static registers (e.g., clock dividers) and for ingesting the critical 3-word absolute-time trigger commands into the hardware scheduling FIFO.

2.3 Universal Verification Methodology Foundations

2.3.1 Overview of UVM and Coverage-Driven Verification

Standardized by Accellera and built entirely upon the SystemVerilog OOP language**[1, 4]**, UVM provides a robust, highly modular, and reusable library of base classes. The primary philosophy of UVM is the shift from manual directed testing to Coverage-Driven Verification combined with Constrained-Random Testing. Instead of manually writing individual test vectors, verification engineers define legal mathematical boundaries for input stimuli. The UVM engine automatically generates millions of randomized transactions, dramatically accelerating the discovery of obscure corner-case bugs and unexpected architectural deadlocks.

2.3.2 Core Component Hierarchy

A standard UVM testbench is structurally compartmentalized, ensuring a strict separation of concerns between stimulus generation, protocol driving, and automated checking. The fundamental components include:

- **Sequence Item (Transaction):** The lowest level of abstraction. It encapsulates physical pin-level toggles into a high-level object containing randomized variables.
- **Sequencer and Sequence:** The uvm_sequence generates streams of randomized Sequence Items, while the uvm_sequencer acts as an arbiter, routing these items to the driver.
- **Driver:** Acts as the physical bridge. It receives abstract Sequence Items from the Sequencer and translates them into cycle-accurate, pin-level hardware toggles required by the specific bus protocol.
- **Monitor:** A passive observer that continuously snoops the physical interface pins. It decodes the hardware waveforms back into abstract Sequence Items and broadcasts them to the rest of the environment for analysis.
- **Agent:** A logical container that encapsulates the Sequencer, Driver, and Monitor. UVM Agents can be configured as Active or Passive.
- **Scoreboard and Reference Model:** The ultimate automated checking mechanism. The Reference Model predicts the expected behavior of the DUT, while the Scoreboard

mathematically compares these expected results against the actual transactions captured by the Monitors.

2.3.3 Advanced UVM Mechanisms: TLM and Phasing

To facilitate the creation of scalable and dead-lock-free environments, UVM relies on two advanced architectural mechanisms:

- **Transaction-Level Modeling (TLM):** UVM components communicate via TLM ports rather than physical wires. This decoupling is essential for verifying multi-channel, parallel architectures like the SPI_CTRL module.
- **The Phasing Mechanism:** UVM enforces a strict chronological execution flow across all instantiated components through distinct phases (e.g., build_phase, connect_phase, run_phase, check_phase).

Chapter 3

In-Depth Functional Analysis of the SPI_CTRL Write Path

3.1 Architectural Overview and Clock Domain Partitioning

The initialization and dynamic configuration of modern RFICs demand high-bandwidth, strictly time-synchronized serial data transmission. The SPI controller module serves as the critical bridge between the SoC interconnect and the physical RFIC array. To ensure a highly focused verification methodology, the scope of this analysis is deliberately restricted to the SPI Write data and control paths.

A comprehensive architectural block diagram of the SPI_CTRL module, highlighting the strictly decoupled AXI4-Lite control plane, the AXI4-Full data plane, and the internal multi-clock domains, is presented in Figure 3.1.

3.1.1 Decoupled Data and Control Planes

To satisfy the dual requirements of high-throughput data buffering and low-latency execution triggering, the module implements a fully decoupled architecture:

- **The Data Plane:** Interfaced via a high-bandwidth, burst-capable system bus (AMBA AXI4-Full compliant). This path is strictly dedicated to allowing the software driver to proactively load massive RFIC configuration sequences into the module's internal memory.

- **The Control Plane:** Interfaced via a lightweight, memory-mapped bus (AMBA AXI4-Lite compliant). This path is utilized for dynamic register configuration and, most crucially, for issuing structured execution triggers. These triggers encapsulate the target memory bank, the designated frame number, and the absolute execution timestamp.

On the physical output side, the module drives an array of independent SPI lanes (configured for 8 parallel antennas). Each lane consists of standard physical signals (Data Out, Serial Clock, and Chip Select), enabling the concurrent programming of multiple RFICs.

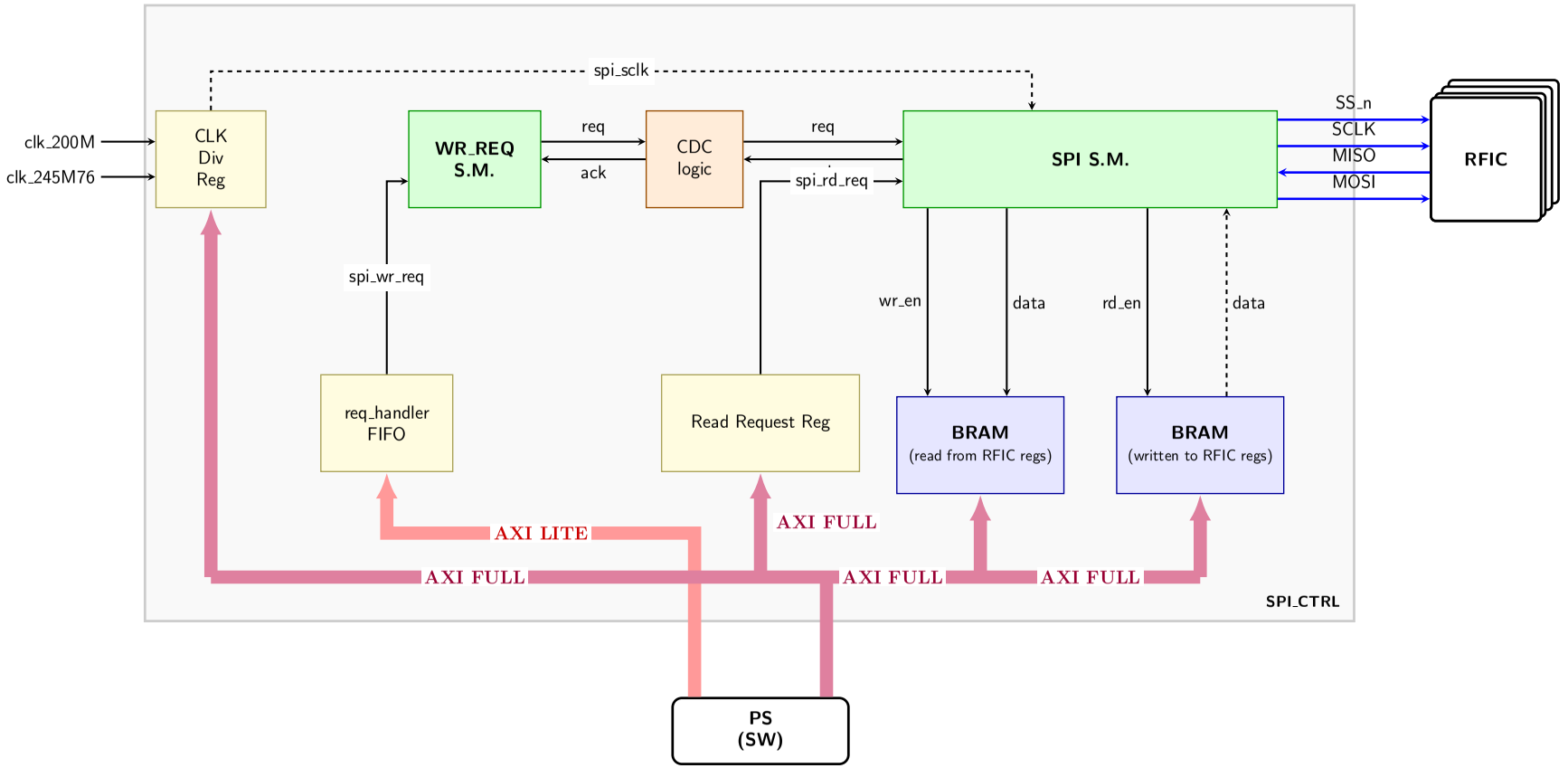


Figure 3.1: Microarchitectural block diagram of the `SPI_CTRL` module. The design illustrates a strictly decoupled architecture: the AXI4-Lite bus handles execution triggers via the request FIFO, while the AXI4-Full interface sustains high-bandwidth payload loading into the dual-BRAMs. Critical control signals traverse asynchronous boundaries via dedicated CDC logic before triggering the core SPI state machine to drive the analog RFICs.

3.1.2 Multi-Clock Infrastructure and Metastability Mitigation

A fundamental complexity of this design is its reliance on three distinct, asynchronous clock domains to manage the chronological execution of SPI transactions:

1. **System Interconnect Domain:** Governs the memory-mapped bus interfaces, the request buffer, and the primary scheduling state machine.
2. **Serial Execution Domain:** Drives the core physical state machine, manages the shift registers, and generates the physical SPI waveforms.
3. **Real-Time Reference Domain:** A dedicated high-frequency global timebase. This domain continuously increments the system's absolute frame and timestamp counters, providing a unified chronological reference for hardware triggers.

Because critical control signals such as execution triggers and completion acknowledgments must traverse these asynchronous boundaries, the hardware extensively employs multi-stage synchronization primitives (cascade flip-flops) to mitigate metastability **[5]**. While these CDC mechanisms ensure electrical stability, they inherently introduce non-deterministic, multi-cycle propagation latency into the control path, presenting a significant timing verification challenge.

3.2 Asynchronous Memory Architecture and Sequence Mapping

To sustain continuous, high-speed SPI transmissions across multiple parallel channels without stalling the SoC interconnect, the hardware architecture physically decouples the software data-loading phase from the hardware execution phase.

3.2.1 Asymmetric Dual-Port RAM Allocation

The core of the data path relies on an embedded Asynchronous Dual-Port RAM (DPRAM). This memory structure is inherently asymmetric to optimize throughput across different clock domains:

- **The System-Facing Port:** Operates within the System Interconnect Domain. It facilitates wide-bus burst writes, allowing the software driver to rapidly populate the memory with extensive RFIC configuration payloads.
- **The Hardware-Facing Port:** Operates within the Serial Execution Domain. It acts as a high-speed, read-only interface, fetching the pre-loaded instructions exclusively when the execution state machine is triggered.

To efficiently manage diverse RFIC configurations, the internal memory space is logically partitioned into discrete sequence banks. Each bank is allocated a specific address offset. This banked architecture enables the software to stage multiple, distinct configuration profiles simultaneously. Consequently, the hardware can seamlessly switch between these profiles during runtime by simply decoding the target bank offset provided in the control trigger.

3.2.2 Programmable Baud Rate Generation

In addition to payload storage, the system interconnect maps critical configuration registers. Notably, a dedicated clock division register dynamically dictates the operational baud rate of the physical SPI transmission.

The hardware derives the final physical serial clock frequency by dividing the core execution clock by this programmable threshold. The architectural separation of data loading and data execution means the hardware buffers configuration payloads indefinitely. Therefore, the physical transmission speed is evaluated dynamically at the exact moment of execution, based on the latest value stored in this division register.

3.3 RTC-Triggered Scheduling Finite State Machine

The defining architectural innovation of the SPI controller is its deterministic, absolute-time execution mechanism. Unlike conventional peripherals that execute commands immediately upon a software register write, this module decouples the issuance of a command from its physical execution, relying instead on a precise Real-Time Clock (RTC) trigger methodology.

3.3.1 Request Frame Parsing and Queue Management

The software driver interfaces with the control plane by transmitting a highly structured, 3-word trigger command over the memory-mapped interconnect. To prevent command loss while the hardware is awaiting a future timestamp, these incoming requests are buffered within an internal asynchronous FIFO structure.

Crucially, this buffer is intentionally designed with a shallow depth limit. This constraint serves as an architectural backpressure mechanism. It ensures that the system interconnect asserts a “not ready” stall signal if the software attempts to over-saturate the queue with pending future transmissions before previous chronological triggers have elapsed.

The 3-word payload is systematically decoded by the hardware into discrete operational parameters:

- **Command Identifier:** Extracts the transaction type and operational flags.
- **Memory Targeting:** Determines the designated sequence bank and the target global frame number.
- **Absolute Timestamp:** Extracts the 32-bit absolute sampling threshold necessary for execution.

3.3.2 The Stalling Evaluation Mechanism

The core scheduling logic is governed by a dedicated Finite State Machine (FSM) operating within the System Interconnect Domain. This FSM manages the lifecycle of a trigger request through a sequence of critical transitions:

1. **Request Ingestion:** Upon detecting a valid queued command, the FSM transitions from an idle state to sequentially fetch the 3-word payload, latching the extracted parameters into internal evaluation registers.

2. **The Chronological Stall:** This is the primary operational state of the control plane. The FSM deliberately suspends execution and enters a continuous polling loop. It performs a strict mathematical equivalency check, comparing the requested target frame and timestamp against the live, incrementing global counters. These counters are continuously bridged from the Real-Time Reference Domain via multi-stage synchronization primitives. The FSM remains locked in this stalling phase until absolute chronological parity is achieved between the requested time and the live system time.

3. **Execution Triggering:** At the precise hardware tick where chronological parity is confirmed, the FSM breaks the stall. It asserts a cross-clock-domain start signal to awaken the physical transmission engine and waits for a synchronized completion acknowledgment.

4. **Response Generation:** Upon physical completion, the scheduling FSM formulates a response packet back to the system interface, signaling the successful execution of the temporal trigger.

3.4 Core Execution Engine and Physical Layer Timing

Upon receiving the synchronized start trigger from the scheduling domain, the operational focus shifts to the serial execution engine. This secondary FSM, operating exclusively within the Serial Execution Domain, is responsible for fetching pre-loaded sequence data from the DPRAM, serializing it, and driving the physical multi-channel SPI lanes in strict adherence to the configured baud rate and temporal gaps.

3.4.1 Sequence Extraction and Temporal Gap Enforcement

The execution engine orchestrates the multi-step process of SPI transmission through a highly optimized state progression. Focusing strictly on the Write data path, the operational flow is defined by the following states:

1. **Instruction Fetching:** The FSM accesses the base address of the designated memory bank. It extracts the sequence length header, which dictates the number of consecutive 16-bit instructions to be transmitted, and dynamically activates the target physical RFIC channels based on embedded mask bits.

2. **Setup Phase (Pre-Gap):** Before shifting the first data bit, the physical Chip Select lines for the active channels are asserted. Crucially, the FSM then stalls for a programmable number of clock cycles. This intentional hardware delay is engineered to satisfy the rigorous Setup Time requirements of the downstream analog RFIC components before any clock edge transitions occur.

3. **Serialization Phase:** This is the core data-shifting state. The FSM remains engaged in this phase until the entire 16-bit payload has been successfully shifted across the active Data Out physical lanes. The specific bit-level timing generated during this state is analyzed in the subsequent section.

4. **Hold Phase (Post-Gap):** Once the transmission concludes, the FSM enters a secondary stall state to satisfy the Hold Time requirements of the RFIC. If subsequent instructions remain in the sequence, the FSM loops back to fetch the next word; otherwise, it deasserts the Chip Select lines, issues a completion acknowledgment to the scheduling domain, and returns to an idle state.

The complete FSM topology governing this chronological sequence—from initialization and payload extraction to physical execution and hold gap fulfillment—is illustrated in Figure 3.2.

3.4.2 Physical Protocol Compliance and Serialization Algorithm

The precise serialization of data onto the physical output lane and the toggling of the Serial Clock lane are governed by an internal baud rate divider counter within the Serialization Phase. This counter increments continuously up to the programmable division threshold, effectively synthesizing the target transmission frequency.

An architectural analysis of the register trasfer level (RTL) counting algorithm inside the `exec_instr` state reveals the exact phase and polarity characteristics of the generated SPI waveform:

- **Data Setup (MOSI Update):** At the absolute zero index of the divider counter, the FSM immediately updates the physical Data Out lane with the next bit from the 16-bit shift register.

- **Leading Edge Generation (Sampling Edge):** When the counter reaches its calculated midpoint, the physical Serial Clock line is driven to a logic-high state (Rising Edge). Because the data was updated at index 0, it is guaranteed to be highly stable across this leading edge, perfectly satisfying the setup-time requirements for the RFIC to sample.

- **Trailing Edge Generation:** When the counter reaches its terminal value, the Serial Clock line is driven back to a logic-low state (Falling Edge), completing one full physical bit-period.

This specific sequence of hardware events where the clock idles low (CPOL = 0), and the data payload is updated prior to the leading clock edge to be sampled on the rising edge confirms that the SPI controller's write data path operates strictly in standard **SPI Mode 0 (Clock Polarity = 0, Clock Phase = 0)**.

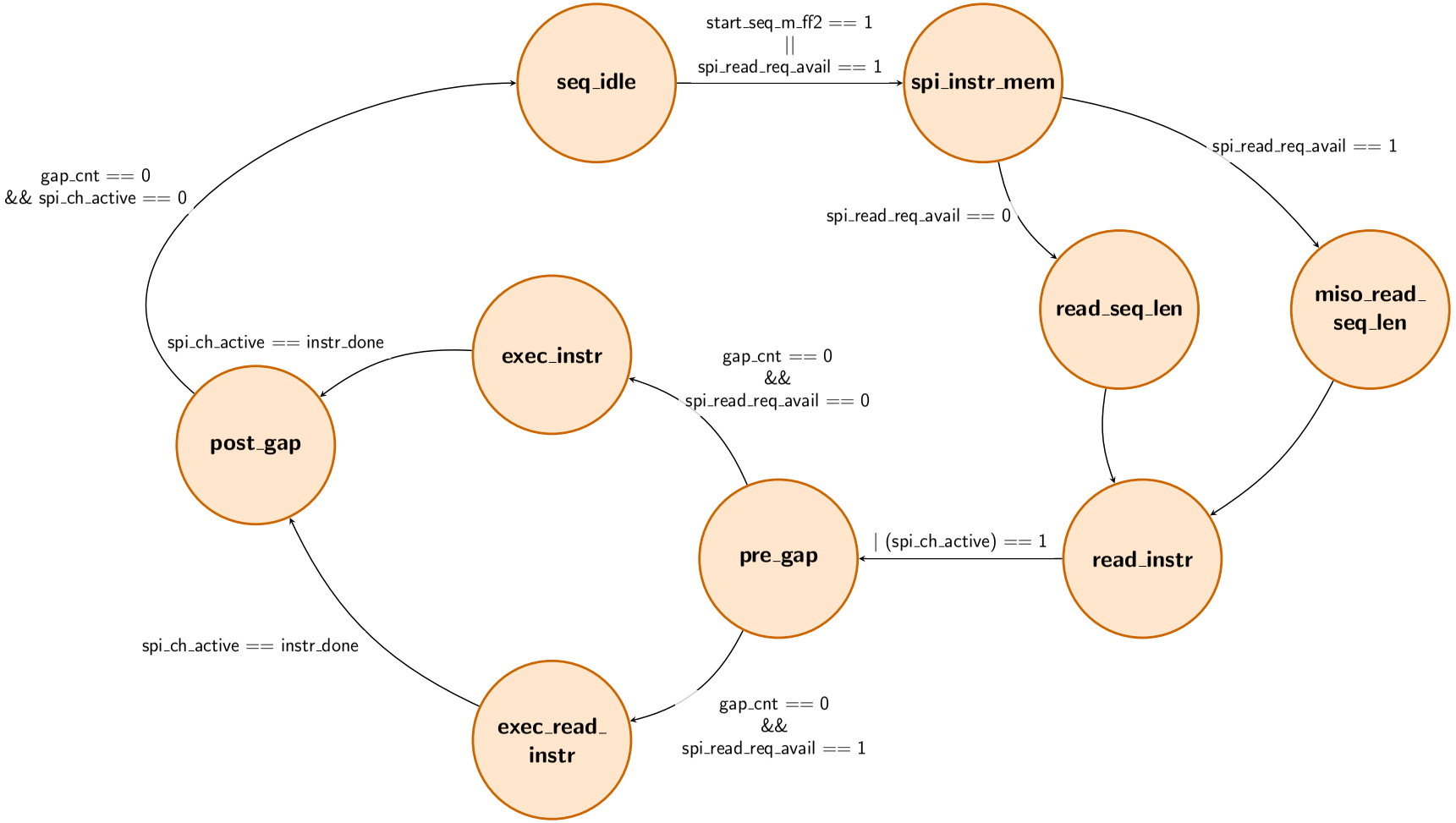


Figure 3.2: FSM topology governing the multi-channel SPI transmissions. The diagram illustrates the complete lifecycle from sequence initialization (`seq_idle`), payload extraction (`read_instr`), physical execution (`exec_instr`), to temporal hold gap fulfillment (`post_gap`).

3.5 Extraction of Core Verification Features

The preceding architectural analysis of the SPI Write data path and its control plane reveals several highly complex hardware behaviors. To achieve exhaustive functional coverage and ensure the robustness of the design, the verification environment must be carefully engineered to address these specific architectural traits. Consequently, the following core verification features have been extracted, which directly influence UVM topology proposed in Chapter 4.

3.5.1 Temporal Decoupling and Shadow Memory Requisite

The design fundamentally decouples the data ingestion phase from the physical execution phase. Because configuration payloads are buffered indefinitely within the internal asynchronous dual-port memory until an absolute-time trigger is issued, the verification environment cannot rely on instantaneous input-to-output prediction.

- **Verification Strategy:** The UVM Reference Model must instantiate a centralized, dynamically updatable "Shadow Memory." This software data structure must precisely mirror the hardware's internal addressing scheme, allowing the prediction engine to independently fetch and construct the expected serial payloads at the exact moment a trigger is detected on the system bus.

3.5.2 Deterministic Triggers with Non-Deterministic Latency

While the scheduling engine initiates transactions based on an absolute chronological target (Real-Time Clock matching), the actual physical emergence of the serial data is subjected to multiple layers of hardware latency. These include multi-stage CDC synchronization cycles, programmable prescaler division rates, and protocol-specific physical setup/hold stalls (gap intervals).

- **Verification Strategy:** A traditional cycle-accurate, delay-based Scoreboard is fundamentally inadequate for this architecture. The verification checking mechanism must implement a mathematically calculated dynamic latency window. Furthermore, it must possess the algorithmic capability to seamlessly mitigate boundary conditions, such as the inherent wrap-around (rollover) characteristic of the hardware frame counter.

3.5.3 Multi-Channel Concurrency and Asynchronous Demultiplexing

The execution engine is capable of driving multiple independent physical SPI lanes concurrently, with each channel potentially transmitting payloads of varying lengths.

- **Verification Strategy:** The UVM Scoreboard must abandon the conventional single-queue architecture. Instead, it requires a robust asynchronous demultiplexing strategy, utilizing an array of independent Transaction-Level Modeling (TLM) FIFOs. This ensures that the expected data packets generated by the Reference Model are strictly isolated by their target channel, preventing false data-mismatch errors during concurrent, multi-lane transmissions.

3.5.4 Command Queue Saturation and Backpressure Evaluation

To prevent the loss of critical scheduling triggers, the hardware implements a shallow request buffer. This architectural choice necessitates that the system bus gracefully handles backpressure when the queue reaches its saturation threshold.

- **Verification Strategy:** The stimulus generation framework must explicitly include aggressive virtual sequences (e.g., rapid, multi-burst trigger injections). These stress scenarios are critical to validating the hardware's ability to assert bus stall signals without corrupting the queued commands or stalling the system interconnect indefinitely.

3.6 Chapter Summary

This chapter has provided a comprehensive microarchitectural dissection of the SPI_CTRL write data path and its associated control plane. By abstracting the raw RTL implementation into high-level state machine theories and physical layer timing protocols, the intrinsic functional complexities of the DUT were systematically elucidated.

The analysis highlighted the module's highly decoupled architecture, wherein the ingestion of massive configuration payloads via the high-bandwidth system interconnect is completely isolated from the physical serial execution. At the core of the control plane, the investigation revealed an innovative, absolute-time scheduling mechanism. Instead of immediate execution, the hardware implements a rigorous chronological stall, utilizing a shallow request buffer to inherently manage backpressure while continuously evaluating RTC synchronizations across multiple asynchronous clock domains.

Furthermore, the dissection of the serial execution engine confirmed its strict adherence to SPI Mode 1 protocols, while exposing the critical, programmable setup and hold phases (temporal gaps) engineered to satisfy downstream RFIC analog constraints. While the multi-stage CDC primitives ensure electrical metastability mitigation, the analysis demonstrated that they concurrently inject non-deterministic propagation latency into the execution pipeline.

Crucially, understanding these hardware behaviors is a fundamental prerequisite for constructing a robust verification environment. The unique architectural traits uncovered in this chapter, specifically the temporal decoupling, the multi-channel concurrency, the non-deterministic chronological latency, and the queue saturation limits, directly translate into the core verification features extracted in Section 3.5. Consequently, these extracted features serve as the definitive blueprint for the UVM topology. They dictate the necessity for the dynamic Shadow Memory, the asynchronous multi-FIFO routing, and the algorithm-driven Absolute

Latency checking mechanisms that will be detailed in the architectural design of the verification platform in Chapter 4.

Chapter 4

UVM Verification Platform Architecture Design

4.1 Architectural Specification of the SPI Controller UVM Verification Environment

4.1.1 Introduction and Architectural Philosophy

The verification environment for the SPI Controller is built upon a highly scalable, robust, and modular UVM framework. The primary objective of this architecture is to ensure exhaustive functional and timing verification of the DUT while maintaining a strict separation of concerns among stimulus generation, bus monitoring, dynamic result prediction, and automated checking.

To achieve this, the architecture integrates industry-standard Cadence Denali Verification IP (VIP) for driving standard AMBA AXI4 and AXI4-Lite interfaces, alongside custom-developed UVM agents dedicated to the proprietary, multi-channel SPI output interfaces. This hybrid approach guarantees protocol compliance on the system bus while providing the necessary flexibility to verify the complex, custom behaviors of the SPI data streams, including dynamic latency, frame boundary crossovers, and high-speed multi-channel synchronization.

The top-level organization and hierarchical topology of the developed UVM verification environment, showcasing the interconnections between active/passive agents and the centralized scoreboard, are illustrated in Figure 4.1.

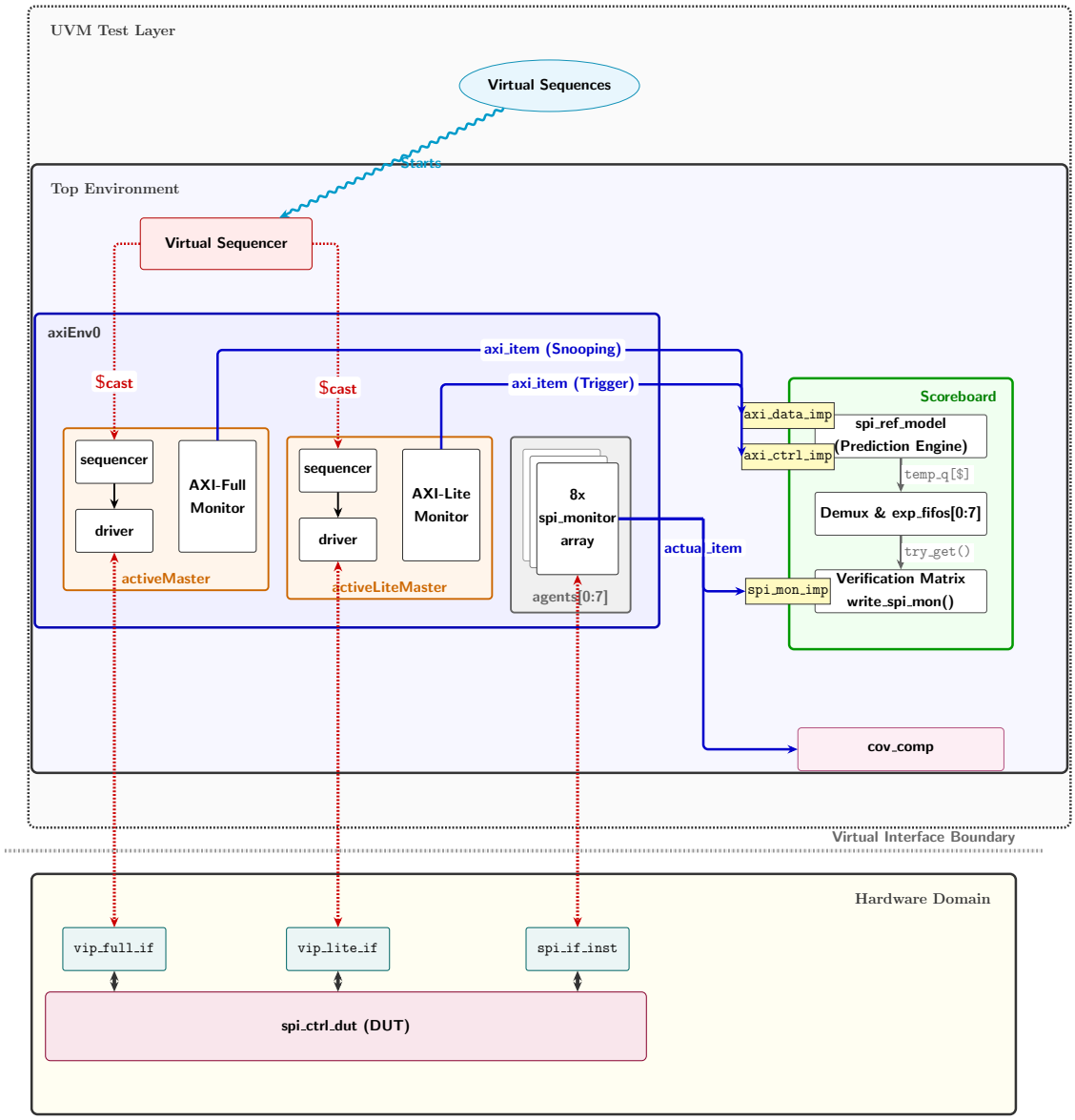


Figure 4.1: Organization of Verification Environment

4.1.2 Static Testbench Topology (spi_ctrl_tb)

The static top-level testbench (`spi_ctrl_tb`) establishes the physical foundation of the simulation. It handles clocking, resets, interface instantiation, and the critical integration between the hardware DUT and the software-driven UVM verification components.

Multi-Domain Clock and Reset Generation

The DUT operates across multiple asynchronous clock domains, which the testbench replicates to mirror real-world system conditions.

- **AXI Clock (`clk_axi`):** Drives the memory-mapped interfaces and system configuration registers.
- **SPI Clock (`clk_spi`):** Drives the actual serial data transmission.
- **Base System Clock (`clk_122P88`):** A high-frequency 122.88 MHz clock utilized as the fundamental timebase for the system's global counters. Independent active-low resets (`rst_axi_n`, `rst_spi_n`) are generated and sequenced to ensure proper initialization of all domains before the injection of UVM stimulus.

Global Timing and Frame Emulation

A fundamental requirement for verifying this SPI controller is highly accurate chronological tracking. The testbench implements a dedicated hardware counter block driven by `clk_122P88`. This block continuously increments a `current_time` variable (up to 1000 ticks) and a `current_frame` variable. These signals are routed into the custom SPI interfaces, serving as an absolute time reference for the UVM monitors to timestamp outgoing serial payloads accurately.

Interface Arrays and Configuration Database Registration

The design features 8 parallel SPI output channels. To avoid repetitive code and ensure scalability, the testbench employs a SystemVerilog generate block to instantiate an array of 8 virtual interfaces (`spi_if`). Through the `uvm_config_db`, each specific interface instance is registered into the UVM configuration space with a unique path (e.g., `*.spi_agent_0*`). This mechanism allows the dynamically constructed UVM agents to retrieve their specific physical connections seamlessly during the build phase. Furthermore, the Cadence AXI VIP virtual interfaces are bridged directly to the DUT's physical AXI ports, ensuring seamless translation from UVM sequence items to pin-level toggles.

4.1.3 UVM Component Hierarchy

The dynamic verification environment is structured hierarchically, ensuring that configuration, stimulus routing, and monitoring are logically isolated.

System Verification Environment (axi4UvmUserSve)

Serving as the highest-level UVM logical container, the System Verification Environment (SVE) encapsulates the entire functional verification apparatus. During its `build_phase`, it constructs the core sub-environment, the centralized Virtual Sequencer, the Scoreboard, and the Coverage collector. The SVE's `connect_phase` acts as the primary switchboard for the environment's TLM network. It systematically maps the inner sequences to the Virtual Sequencer, routes the AXI monitor callbacks to the scoreboard's data and control ports, and establishes a parallel fan-out of the 8 SPI agent monitors to both the scoreboard (for correctness checking) and the coverage collector (for metric aggregation).

The Sub-Environment (axi4UvmUserEnv)

The Env layer is dedicated to configuring and housing the various agents.

- **AXI VIP Configuration:** It instantiates the `axi4UvmUserActiveMaster Agent` (for high-bandwidth BRAM data loading via AXI-Full) and the `axi4Lite UvmUserActiveMasterAgent` (for register configuration and hardware triggering). Specialized configuration objects define memory segments (e.g., mapping `64'h0` to `64'hFFFF_FFFF` as non-shareable domains) and configure AXI burst behaviors to match system specifications.

- **SPI Agent Deployment:** Using a `for` loop, the environment dynamically creates 8 instances of the custom `spi_agent`. Crucially, it leverages the `uvm_config_db` to assign a unique integer `channel_id` (0 through 7) to each agent's internal monitor. This identifier is vital for the scoreboard to demultiplex the asynchronous data streams later in the pipeline.

4.1.4 Custom SPI Protocol Agents (spi_agent & spi_monitor)

Unlike standard active agents that drive stimulus, the `spi_agent` array is explicitly configured with `is_active = UVM_PASSIVE`. Their sole responsibility is to silently observe the DUT's output pins without interfering.

State-Machine Driven Monitoring

The `spi_monitor` contains a robust, phase-accurate state machine running in the `run_phase` to decode the serial protocol:

1. **Synchronization:** The monitor blocks execution until it detects a falling edge on the Chip Select (`cs_n`), signaling the start of a valid SPI frame.

2. **Timestamping:** Immediately upon assertion of `cs_n`, it captures the hardware `current_frame` and `current_time` from the virtual interface, tagging the beginning of the transaction.

3. **Data Shifting:** Operating in SPI Mode 0, it enters a `while(cs_n == 0)` loop, sampling the `mosi` line on every rising edge of the Serial Clock (`sck`). It shifts these bits into a 16-bit register (MSB first).

4. **Transaction Packaging:** Once 16 bits are accumulated, the monitor constructs a `spi_seq_item`, populates the payload data, attaches the unique `channel_id`, stamps it with the captured frame/time, and broadcasts the object through its `item_collected_port` to the broader UVM environment.

4.1.5 The Reference Model: Dynamic Prediction Engine (spi_ref_model)

The `spi_ref_model` is a `uvm_object` that functions as the primary prediction engine of the verification environment. It perfectly mimics the internal memory and processing logic of the DUT, predicting the exact expected SPI output independently of the RTL execution.

Shadow Memory Architecture

To predict output data, the reference model must know what data resides in the DUT. It implements an associative array (`logic [7:0] bram_mem [bit [31:0]]`) representing a shadow copy of the hardware Block RAM. Through a TLM connection from the AXI-Full monitor, every byte written to the DUT's memory by the VIP is simultaneously mapped into this shadow memory. A dedicated `read_word_from_mem` function reconstructs 16-bit little-endian words on demand, accurately mirroring the physical hardware's memory retrieval characteristics.

Algorithmic Trigger Processing

The prediction mechanism is activated via the `process_trigger` function, invoked when the AXI-Lite bus issues a specific 3-word control sequence. The model decodes this trigger to extract the target Bank Address, the expected Frame number, and the trigger Timestamp.

Once decoded, the model simulates the DUT's multi-channel burst engine:

- It iterates over all `NUM_CHANNELS` (8 channels).
- For each channel, it reads a sequence length from the base address. If the length is zero, the channel is skipped, saving computational overhead.
- If active, it executes a secondary loop, fetching payload data from the shadow memory based on a mathematically derived address formula: `Base + (Step * ROW_STRIDE) + (Channel Offset)`.
- **Advanced Timing Flagging:** A critical architectural feature is how the reference model tags the generated expected items. It assigns `check_timing_strict = 1` solely to the *first* packet of a burst sequence. All subsequent packets in that burst are assigned `check_timing_strict = 0`. This design acknowledges that only the initiation of a burst has a deterministic hardware latency from the trigger; subsequent packets are bound only by physical SPI clock constraints and sequence causality.

4.1.6 The Scoreboard: Multi-Dimensional Verification (spi_scoreboard)

The `spi_scoreboard` acts as the final arbiter of correctness. It is a highly complex component designed to ingest asynchronous streams of data from three distinct sources, synchronize them, and apply rigorous checks regarding data integrity and temporal accuracy.

TLM Architecture and Asynchronous Decoupling

The scoreboard defines three `uvm_analysis_imp` ports:

1. **`axi_data_imp`:** Intercepts AXI-Full writes to update the reference model's shadow BRAM. It also acts as a snooper, detecting dynamic configuration writes to address `0xA0A0_4000` to automatically update the internal `EXP_LATENCY` parameter.
2. **`axi_ctrl_imp`:** Buffers AXI-Lite writes into a `req_buffer`. Once a complete 3-word command is accumulated, it invokes the reference model's prediction engine.
3. **`spi_mon_imp`:** Receives the actual output packets generated by the DUT.

To handle the parallel, asynchronous nature of the 8 SPI channels, the scoreboard implements an array of 8 `uvm_tlm_analysis_fifos`. When the reference model predicts a burst of items for various channels, the scoreboard intelligently demultiplexes these expected items, pushing them into their respective channel's dedicated FIFO. When a physical packet arrives from the SPI monitor, the scoreboard pops the expected item from the matching FIFO. This multi-FIFO architecture guarantees thread safety and prevents out-of-order false failures that would occur in a single-queue design.

Data Integrity and Temporal Verification

Upon matching an expected item with a captured item, the scoreboard executes a multi-tiered verification matrix:

- **Payload Verification:** A direct bitwise comparison of the 16-bit data payloads.
- **Strict Absolute Latency Checking:** If the expected item is flagged with `check_timing_strict = 1`, the scoreboard translates both the expected and captured frame/timestamp vectors into a continuous, linear "Absolute Time" value (`Frame * TICKS_PER_FRAME + Timestamp`). It then calculates the `latency_diff`. The test passes only if this difference falls strictly within the dynamically calculated `[EXP_LATENCY - TOLERANCE, EXP_LATENCY + TOLERANCE]` window.
- **Frame Rollover Mitigation:** The 8-bit frame counter inherently resets from 255 back to 0. If a trigger occurs at Frame 255 but the hardware latency pushes the output capture to Frame 0, a naive mathematical comparison would fail. The scoreboard's timing engine mathematically detects this boundary rollover and dynamically injects a full 256-frame cycle into the calculation, ensuring robust verification across counter resets.

- **Causality Verification:** For packets flagged with `check_timing_strict = 0`, the scoreboard drops the rigid latency window and enforces a fundamental sanity check: The absolute capture time must be strictly greater than the absolute expected trigger time. This catches severe chronological hardware bugs, such as causality violations or data buffering corruption.

End-of-Test Phase Checking

To ensure complete transmission and zero data loss, the scoreboard leverages the UVM `check_phase`. At the conclusion of the simulation, it iterates through all 8 expected FIFOs. If any FIFO is not completely empty, it issues a `UVM_ERROR`. This guarantees that every transaction requested by the AXI trigger was successfully transmitted by the SPI interface and captured by the monitors.

4.2 Test Sequences and Scenarios Specification

4.2.1 Documentation Template Definition

To maintain consistency across the verification environment and enhance thesis readability, all stimulus sequences and items will be documented using the following standardized template:

- **Sequence Name:** The exact class name of the UVM sequence.
- **Sequence Type:** Base Sequence / Virtual Sequence / Data Item.
- **Objective:** A high-level description of the sequence's purpose and the specific DUT features it targets.
- **Knobs, Constraints & Randomization:** Key randomized variables, configurable parameters, and constraint blocks that shape the stimulus.
- **Execution Flow (Body Task):** A step-by-step breakdown of the logical operations executed within the sequence's `body()` task.
- **Expected System Response:** The anticipated hardware behavior and the corresponding scoreboard/reference model checks triggered by this sequence.

4.2.2 Stimulus Data Objects

Sequence Name: `spi_load_packet`

- **Sequence Type:** UVM Data Object (`uvm_object`)
- **Objective:** To generate a highly constrained, multi-row randomized data payload that will eventually be written into the SPI controller's BRAM. It ensures the generated data adheres to strict byte-level relationship rules required for hardware parsing.

- **Knobs, Constraints & Randomization:**
  - `load_data[]`: A dynamic array of 32-bit words representing the BRAM payload.
  - `num_rows`: An unsigned integer controlling the burst depth.
  - `c_structure`: Constrains `num_rows` to a valid operational range (1 to 64) and forces the data array size to be a multiple of 8 columns (`num_rows * 8`).
  - `c_zero_padding`: Enforces that the upper 16 bits of every 32-bit word are strictly zero-padded (`16'h0000`), matching the 16-bit physical SPI data width.
  - `c_max_relationship`: A complex constraint guaranteeing that no lower byte in the first row equals or exceeds `num_rows`, while ensuring at least one byte exactly equals `num_rows - 1`.
- **Execution Flow:** As a data object, it does not possess a `body()` task. It relies on the UVM `randomize()` method and features a custom `display()` function to print the generated 8-column hex payload for debugging purposes. We define such constraints to compose the input at will by overriding constraints in upper sequences.

4.2.3 Base Sequences

Sequence Name: `spi_data_load_seq`

- **Sequence Type:** Base Sequence (`userSimpleSeq -> cdnAxiUvmSequence`)
- **Objective:** To write the massive data payloads generated by `spi_load_packet` into the DUT's BRAM over the AMBA AXI-Full interface. It handles automatic data segmentation to comply with AXI protocol limits.
- **Knobs, Constraints & Randomization:**
  - `data_payload[]`: The raw 32-bit data array to be transmitted.
  - `target_addr`: The AXI starting address. Constraint `c_addr_valid` ensures the address is word-aligned (`[1:0] == 2'b00`) and softly defaults to the BRAM base address `32'hA0A0_0000`.
- **Execution Flow (Body Task):**
  1. **Safety Check:** Verifies the `data_payload` size; if empty, it allocates a default 4-word array.
  2. **Burst Segmentation:** Enters a `while` loop to transmit the entire payload. It calculates the `burst_len` dynamically, ensuring no single AXI transaction exceeds the protocol maximum of 256 beats.
  3. **Transaction Randomization:** Creates and randomizes an AXI Write transaction (`DENALI_CDN_AXI_TR_Write`) of type INCR, sizing it to the calculated `burst_len`.
  4. **Endianness Packing:** Manually maps the 32-bit randomized words into the byte-aligned `trans.Data` queue using standard Little-Endian formatting.

5. **Transmission & Iteration:** Sends the transaction to the driver, waits for the response, and increments both the `words_sent` counter and the `current_addr` pointer until the entire payload is successfully written.

- **Expected System Response:** The AXI VIP drives the physical bus. The scoreboard intercepts these writes to update the reference model's shadow memory.

Sequence Name: `simple_lite_worker_seq`

- **Sequence Type:** Base Sequence (`cdnAxiUvmSequence`)
- **Objective:** To generate background AXI-Lite traffic (both reads and writes) to verify the robustness of the DUT's control plane under basic operational stress.
- **Knobs, Constraints & Randomization:**
  - Randomizes `denaliCdn_axiTransaction` objects.
  - Constrains transactions strictly to AXI-Lite requirements: Length == 1, Size == WORD (4 bytes), and BurstKind == INCR.
- **Execution Flow (Body Task):**
  1. Executes 40 randomized Read/Write single-beat transactions across the general address space.
  2. Executes 10 targeted Write transactions specifically hitting the address range between `32'h1000` and `32'h1FFF`.

Sequence Name: `spi_trigger_seq`

- **Sequence Type:** Virtual Sub-Sequence (`axi4UvmVirtualSequence`)
- **Objective:** To formulate and dispatch the critical 3-word control sequence over the AXI-Lite interface that commands the DUT hardware to commence SPI transmission.
- **Knobs, Constraints & Randomization:**
  - `ctrl_config`: The primary control register configuration (e.g., channel enables, mode settings).
  - `bank_addr`: The target memory bank offset.
  - `frame_info`: The specific frame boundary target.
  - `trigger_time`: The absolute time target for execution.
- **Execution Flow (Body Task):**
  1. Packs the randomized variables into three distinct 32-bit command words (`w0, w1, w2`).
  2. Utilizes the `p_sequencer.litemasterSeqr` to execute three consecutive AXI-Lite writes to address `32'h0`.

- **Expected System Response:** The DUT absorbs these three words and triggers the physical SPI state machine. Simultaneously, the UVM Scoreboard buffers these writes and triggers the Reference Model's prediction algorithm.

4.2.4 System-Level Virtual Sequences

Sequence Name: `stress_test_vseq`

- **Sequence Type:** Top-Level Virtual Sequence (`axi4UvmVirtualSequence`)
- **Objective:** The primary orchestration sequence for system-level stress testing. It coordinates data generation, memory loading, hardware configuration, execution triggering, and dynamic simulation delay calculation.
- **Knobs, Constraints & Randomization:**
  - `clk_div`: Configures the SPI clock divider (constrained to positive values, softly defaults to 4).
  - `rand_frame_num` & `rand_trigger_time`: Defines the temporal execution target. Constrained to legal operational ranges (frames 3-255, time 1-999).
  - `bank_offset`: Points to the memory bank, constrained to a maximum of `8'h38` and strict 8-byte alignment (`[2:0] == 3'b000`).
- **Execution Flow (Body Task):**
  1. **Payload Generation:** Instantiates and randomizes the `spi_load_packet` to generate legal BRAM data.
  2. **Memory Load:** Dispatches `spi_data_load_seq` on the AXI-Full sequencer to write the generated payload to the calculated hardware BRAM address (`32'hA0A0_0000 + bank_offset`).
  3. **Hardware Configuration:** Reuses `spi_data_load_seq` to write the randomized `clk_div` value to the configuration register at `32'hA0A0_4000`.
  4. **Hardware Trigger:** Dispatches the `spi_trigger_seq` on the AXI-Lite sequencer, passing down the randomized frame, time, and bank constraints to command the DUT to start.
  5. **Dynamic Wait Calculation:** Instead of utilizing an arbitrary hardcoded delay, the sequence mathematically computes the expected hardware latency: `expected_cycles = rand_frame_num * 1001 + rand_trigger_time`.
  6. **Simulation Suspend:** Suspends the sequence thread for the calculated duration plus a safety margin (`expected_cycles * 15ns + 32 * clk_div * 15ns + 150us`) to allow the physical SPI interfaces and the UVM monitors to complete transaction processing.

- **Expected System Response:** The entire environment engages. The reference model predicts the multi-channel bursts, the DUT drives the physical SPI lines, the passive monitors capture the output, and the scoreboard validates the payload integrity and absolute latency dynamically.

Sequence Name: `trishot_vseq`

- **Sequence Type:** Top-Level Virtual Sequence (`axi4UvmVirtualSequence`)
- **Objective:** To verify the DUT's ability to handle multiple, rapid-succession hardware triggers using pre-loaded data across different memory banks. It stresses the SPI controller's state machine and command queue by issuing back-to-back transmissions with minimal idle time.
- **Knobs, Constraints & Randomization:**
  - `clk_div`: Configures the SPI clock divider (constrained to even values, strictly positive, softly defaults to 16).
  - Instantiates two separate `spi_load_packet` objects (`pkt1`, `pkt2`) to generate two distinct data payloads.
- **Execution Flow (Body Task):**
  1. **Pre-loading:** Randomizes both packets and utilizes the AXI-Full master to sequentially load `pkt1` into Bank 0 (`32'hA0A0_0000`) and `pkt2` into Bank 1 (`32'hA0A0_0800`).
  2. **Configuration:** Sets the `SPI_CLK_DIV` register.
  3. **Multi-Trigger Execution:** Dispatches four discrete `spi_trigger_seq` commands targeting Bank 0 with varying frames (`0x04`, `0x20`, `0x2F`, `0x31`) and timestamps.
  4. **Stressed Spacing:** Uses extremely tight simulation delays (as low as 1us) between the first three triggers to test the controller's burst readiness, followed by a longer 100us and 500us delay to conclude the test.
- **Expected System Response:** The DUT must accurately fetch data from Bank 0 and execute all four SPI bursts at their respective target frames/times without dropping commands or corrupting the payload due to the tight execution window.

Sequence Name: `rewrite_vseq`

- **Sequence Type:** Top-Level Virtual Sequence (`axi4UvmVirtualSequence`)
- **Objective:** To validate the dynamic memory overwriting capabilities of the system. It ensures that the controller can safely accept new BRAM data payloads from the AXI-Full interface between active SPI transmissions and correctly output the updated data on subsequent triggers.

- **Knobs, Constraints & Randomization:**
  - Similar clock divider constraints (`clk_div[0] == 1'b0`) to ensure stable clock generation.
  - Utilizes two separate payloads (`pkt1`, `pkt2`) to simulate data refreshment.
- **Execution Flow (Body Task):**
  1. **Initial Load & Execution:** Loads `pkt1` into Bank 0, configures the clock, and triggers the first SPI transmission targeting Frame `0x04`.
  2. **In-flight Overwrite:** Waits for 150us (allowing the first transmission to clear), then re-engages the `fullWriteSeq` to strictly overwrite the exact same Bank 0 base address (`32'hA0A0_0000`) with the new `pkt2` payload.
  3. **Secondary Execution:** Dispatches a second trigger for Frame `0x2F`.
- **Expected System Response:** The Reference Model's shadow memory must update mid-simulation to reflect the AXI writes. The Scoreboard will verify that the first SPI burst matches `pkt1`, and the second burst accurately reflects the dynamically overwritten `pkt2` data.

Sequence Name: `mult_bank_vseq`

- **Sequence Type:** Top-Level Virtual Sequence (`axi4UvmVirtualSequence`)
- **Objective:** To rigorously test the address decoding and bank offset logic of the SPI controller. It verifies that the hardware can dynamically switch its fetch address based on the `bank_addr` parameter provided in the AXI-Lite trigger command.
- **Knobs, Constraints & Randomization:**
  - `bank_offset1` & `bank_offset2`: Two 8-bit random variables determining the memory offsets.
  - `correct_bank` Constraint: Forces both offsets to be valid (`<= 8'h38`), strictly 8-byte aligned (`[2:0] == 3'b000`), and explicitly unequal (`bank_offset1 != bank_offset2`) to guarantee collision-free memory allocation.
- **Execution Flow (Body Task):**
  1. **Dynamic Address Loading:** Loads `pkt1` to the dynamically calculated Bank 0 address (`32'hA0A0_0000 + bank_offset1`) and `pkt2` to the Bank 1 address (`32'hA0A0_0800 + bank_offset2`).
  2. **Bank 0 Trigger:** Sends an AXI-Lite trigger command explicitly targeting `bank_offset1`.
  3. **Bank Switching:** Waits 300us, then sends a second trigger command explicitly targeting `bank_offset2` (with Bank 1 configuration logic applied via `ctrl_config`).

- **Expected System Response:** The hardware must seamlessly transition its fetch pointers from the first random offset to the second random offset without cross-contamination of data. The Scoreboard ensures that packets collected from the respective triggers perfectly match `pkt1` and `pkt2`.

4.3 Simulation Automation and Regression Execution

Overview of the Automation Script (run_sim.py)

To streamline the verification process and enable Continuous Integration (CI), the environment utilizes a custom Python-based execution wrapper (`run_sim.py`). This script automates the compilation, elaboration, and execution of Cadence Xcelium (`xrun`) simulations, abstracting away complex tool arguments and environment configurations.

Key Features and Functionalities

- **Automated Environment Setup:** Dynamically configures critical environment variables required for Cadence tools and VIPs, making the script standalone and agnostic of the user's base shell configuration.
- **Three-Step Execution Model:** Logically separates the simulation process into Compile (`-compile`), Elaborate (`-elabonly`), and Run phases. This highly optimizes regression testing, as compilation and elaboration are performed only once, regardless of how many test iterations are run.
- **Regression & Randomization:** Features a `-repeat N` flag to execute multi-iteration regression tests automatically. For each iteration, it dynamically generates and injects a random SystemVerilog seed (`-svseed`) to maximize state-space coverage across runs.
- **Smart Log Inspection:** Natively parses simulation log files to actively hunt for `UVM_ERROR`, `UVM_FATAL`, and Cadence internal tool errors (`*E`). This is a critical fail-safe, ensuring that tests are marked as "FAILED" even if the simulator unexpectedly returns a success (0) exit code despite UVM errors.
- **Automated Coverage Merging:** Automatically detects multi-run regressions and utilizes Cadence IMC to merge all generated code and functional coverage databases into a single, unified report upon completion.
- **Workspace Management:** Provides robust `-clean` and `-clean-cov` arguments to safely purge old compilation databases, waveforms (`.shm`), and logs, ensuring a pristine state for new simulation runs.

4.4 Chapter Summary

This chapter has detailed the ground-up development and architectural implementation of a robust, highly scalable UVM environment dedicated to the SPI_CTRL module. The primary

objective of this platform was to systematically address and resolve the unique hardware complexities, specifically the temporal decoupling, multi-clock domain crossings, and RTC scheduling identified during the architectural analysis in Chapter 3.

At the structural level, the environment successfully integrates industry-standard AMBA VIPs with custom-developed, passive SPI protocol agents. By implementing an intelligent Reference Model equipped with a Shadow Memory and an algorithmic trigger decoder, the platform achieves independent, cycle-accurate prediction of the hardware's internal state. Furthermore, the Scoreboard's innovative multi-FIFO topology and dynamic Absolute Latency checking mechanisms effectively resolve the non-deterministic delays and frame rollover ambiguities inherent to the multi-clock design, ensuring rigorous chronological validation without false mismatch errors.

At the stimulus level, a comprehensive suite of constrained-random virtual sequences was engineered. These sequences establish a rigorous test plan ranging from fundamental payload integrity validations (`stress_test_vseq`) to aggressive corner-case explorations, including queue saturation/backpressure evaluation (`trishot_vseq`), dynamic memory overwriting (`rewrite_vseq`), and boundary address decoding (`mult_bank_vseq`).

Finally, the integration of a custom Python-based automation framework (`run_sim.py`) elevates the verification platform to industry-standard Continuous Integration (CI) readiness. By automating compilation, dynamic seed randomization, rigorous log parsing, and unified coverage merging, the infrastructure ensures highly efficient regression execution.

Chapter 5

Design Flaw and Coverage Analysis

5.1 Efficacy of the UVM Architecture: Uncovered Design Flaws

The primary metric of success for any functional verification environment is its proactive capability to expose deeply embedded microarchitectural flaws prior to silicon fabrication. The architecture detailed in Chapter 4 was engineered to apply maximal stress to the SPI_CTRL module's operational boundaries.

The successful identification of the following defects demonstrates the effectiveness of the UVM platform. Specifically, the sophisticated Shadow Memory prediction model, the dynamic Absolute Latency checking algorithms within the Scoreboard, and the deployment of extreme constrained-random virtual sequences (such as `trishot_vseq` and boundary injections) collaboratively orchestrated scenarios that a simplistic, directed-test approach would have inherently missed.

By pushing the hardware into states of queue saturation, evaluating extreme baud rate divisions, and injecting uninitialized memory pointers, the verification environment exposed critical edge-case failures. The most significant uncovered bugs, their manifestations within the simulation, and their corresponding microarchitectural root cause analyses are documented below.

5.2 Analysis of Uncovered Design Flaws

5.2.1 Temporal Scheduling Queue Starvation (Bug ID: 02)

Detection Context: This critical failure was exposed by the `trishot_vseq` virtual sequence, which was specifically engineered to stress the hardware's shallow request buffer by injecting multiple, rapid-succession execution triggers.

Failure Manifestation: When three consecutive requests were dispatched to the control plane, with their absolute execution timestamps configured strictly in the future, the DUT

correctly executed the first chronological request. However, it systematically ignored the second request, failing to process it despite it being mathematically valid.

Root Cause Analysis: The defect originated in the state transition logic of the chronological scheduling engine. While the hardware was suspended in the stalling phase awaiting the first timestamp, the internal FIFO handshake logic failed to correctly latch or backpressure the subsequent incoming requests. Consequently, intermediate requests were silently dropped, leading to a loss of control plane commands and exposing a fundamental flaw in the multi-request queuing mechanism.

5.2.2 Pipeline Latency Violation at High Baud Rates (Bug ID: 04)

Detection Context: This flaw was isolated during stress testing when the programmable baud rate divider (`SPI_CLK_DIV`) was constrained to extreme minimum values (e.g., `clk_div = 2` and `clk_div = 4`).

Failure Manifestation: At `clk_div = 4`, the UVM Scoreboard flagged a critical payload mismatch: the hardware erroneously serialized and transmitted the internal length-information header onto the physical Data Out lane, instead of the actual configuration payload. At `clk_div = 2`, the physical Chip Select and Serial Clock signals collapsed entirely, failing to toggle.

Root Cause Analysis: This behavior highlighted a classic pipeline latency violation. The execution engine's state machine assumes that the data fetched from the internal Block RAM will be stable before the serialization phase begins. However, at exceedingly low clock division ratios, the state machine transitioned through the setup phase (`pre_gap`) faster than the inherent read latency of the synchronous memory. This misalignment caused the FSM to sample the data bus prematurely, resulting in either the transmission of erroneous header data or the complete collapse of the physical layer timing generator.

5.2.3 Zero-Length and Boundary FSM Deadlock (Bug ID: 01 & 03)

Detection Context: These deadlocks were identified when the environment injected boundary constraints, specifically pointing the execution trigger to an uninitialized memory bank (all-zero data) or injecting payloads exceeding the maximum bank depth (>64 rows).

Failure Manifestation: If the sequence length header was read as `0x0000` (indicating zero active channels or zero length), the execution engine permanently deadlocked, stalling all subsequent operations. Conversely, if the sequence length exceeded the physical bank boundary, the FSM entered an infinite loop, continuously re-transmitting out-of-bounds data.

Root Cause Analysis: Both manifestations share a common architectural oversight: the lack of boundary-condition safeguards within the execution FSM. In the zero-length scenario, the FSM lacked a conditional bypass path, causing it to wait indefinitely for a transmission completion flag that would never assert. In the overflow scenario, the address pointer logic lacked saturation protection, allowing the FSM to wrap around or blindly increment beyond the allocated memory segment.

5.3 Construction of the Spatio-Temporal Coverage Model

While automated tools ensure high Code Coverage (Line, Branch, and FSM), relying solely on structural metrics is insufficient for validating a highly decoupled, absolute-time-triggered architecture. To achieve rigorous functional closure, the verification environment implements a dedicated `spi_coverage` component. Derived from `uvm_subscriber`, this component passively ingests the physical output transactions (`spi_seq_item`) captured by the monitors.

Rather than executing a computationally expensive, brute-force enumeration of all 16-bit payloads, the covergroup (`spi_trans_cg`) was strategically engineered to target the absolute operational extremes of the physical layer and the chronological scheduling engine. This Targeted Coverage Strategy is categorized into three critical evaluation dimensions.

5.3.1 Temporal Boundaries and Rollover Dynamics (cp_time & cp_frame)

As detailed in Chapter 3, the SPI_CTRL module relies on a continuous 1000-tick RTC and an 8-bit frame counter. Validating the execution engine's behavior at the absolute extremities of these counters is paramount.

The coverage model implements highly specific bins for the physical capture timestamp (`cap_ts`):

- **`start_of_frame` and `end_of_frame`:** By isolating the boundaries (ticks 0-20 and 981-1000), the model guarantees that the FSM's trigger logic does not suffer from off-by-one latency violations when executing transactions immediately following or immediately preceding a clock reset cycle.

Concurrently, the `cp_frame` coverpoint explicitly isolates the global frame counter's extremes:

- **`frame_zero` (0) and `frame_max` (255):** These bins confirm that the UVM environment has successfully stressed the hardware across its critical rollover boundary. Hitting these bins ensures that neither the hardware comparator nor the Scoreboard's dynamic latency algorithm fails during the 255-to-0 chronological transition.

5.3.2 Payload Integrity and Physical Toggle Stress (cp_data)

To validate the robustness of the physical serialization phase (MOSI lane), the `cp_data` coverpoint targets fundamental hardware stress patterns rather than random payload permutations.

- **`all_zeros` and `all_ones`:** These bins ensure the physical lines are immune to "stuck-at" faults and verify that the Shift Register accurately holds prolonged logical states without drifting.

- **`toggle_55` (0x5555) and `toggle_AA` (0xAAAA):** These are standard VLSI maximum-toggle patterns (alternating 0s and 1s). Hitting these bins mathematically proves that the SPI_CTRL module can sustain maximum dynamic switching frequencies across the AXI-Full datapath, the internal DPRAM, and the physical output pins without violating setup or hold constraints.

5.3.3 Spatio-Temporal Cross-Correlation (cross_ch_time)

The most complex architectural feature of the design is its ability to drive 8 independent physical channels concurrently. To prove that no single antenna channel suffers from priority starvation or chronological locking, the model implements a cross-coverage matrix: `cross_ch_time`.

This matrix intersects the physical channel identifier (`cp_channel`) with the temporal execution windows (`cp_time`). Achieving high saturation in this cross-coverage matrix serves as definitive proof that the hardware's internal multi-channel execution engine is fully decoupled. It proves that any of the 8 physical lanes can be successfully triggered and serialized at any arbitrary phase (Start, Mid, or End) of the RTC frame.

5.4 Coverage Convergence and Advanced Constraint Engineering

The ultimate objective of the automated regression suite, orchestrated by the custom `run_sim.py` framework, was to drive the verification environment toward absolute coverage closure. The collected data was merged and analyzed using the Cadence Integrated Metrics Center (IMC) tool **[6]**to evaluate both the structural integrity of the RTL and the functional rigor of the UVM stimulus.

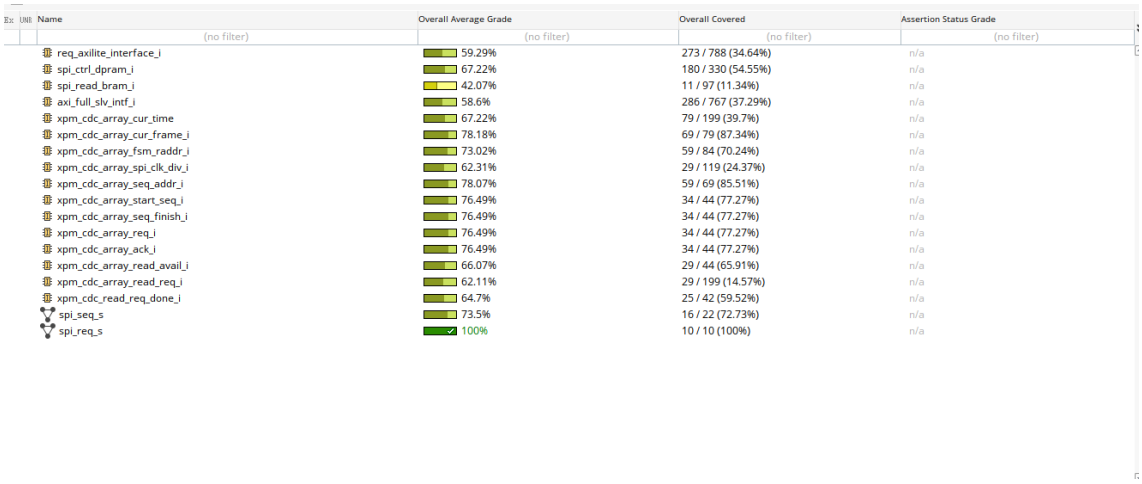
5.4.1 Structural Coverage and Architectural Waivers

The structural coverage data accurately reflects the strictly scoped verification strategy established in Chapter 3. Rather than chasing a superficial 100% across all instantiated sub-modules, the coverage analysis focused exclusively on the customized control and write-datapath logic.

As illustrated in Figure 5.1, the system's primary scheduling brain, the chronological request state machine (`spi_req_s`) achieved a perfect **100% saturation** (10/10 transitions). This confirms that all boundary conditions regarding absolute-time stalling, trigger generation, and hardware backpressure were fully exercised.

The secondary execution state machine (`spi_seq_s`) achieved a highly deterministic **73.5% saturation** (16 out of 22 transitions). This result perfectly aligns with the targeted scope of this thesis: the 16 hit transitions comprehensively encompass the SPI Write datapath, including the setup/hold gap delays and the serialization phases. The remaining unhit transitions are explicitly associated with the SPI Read-Back capability, which are mathematically isolated and intentionally excluded from the current test plan.

Furthermore, lower structural coverage metrics were observed in standard peripheral instances, such as the `axi_full_slv_intf` and the Xilinx-provided `xpm_cdc_array` synchronization primitives. In an industry-standard sign-off process, these components are subject to **Coverage Waivers**. Since they are pre-verified third-party intellectual properties (IPs) or utilized strictly for narrow subset protocols (e.g., standard AXI INCR bursts), exhaustive toggle and branch coverage of their unused legacy features is computationally redundant and architecturally unnecessary for validating the SPI_CTRL module. Report from IMC is illustrated at Figure 5.1.



Name	Overall Average Grade	Overall Covered	Assertion Status Grade
(no filter)	(no filter)	(no filter)	(no filter)
req_axilite_interface_i	59.29%	273 / 788 (34.64%)	n/a
spi_ctrl_dpram_i	67.22%	180 / 330 (54.55%)	n/a
spi_read_bram_i	42.07%	11 / 97 (11.34%)	n/a
axi_full_slv_intf_i	58.6%	286 / 767 (37.29%)	n/a
xpm_cdc_array_cur_time	67.22%	79 / 199 (39.7%)	n/a
xpm_cdc_array_cur_frame_i	78.18%	69 / 79 (87.34%)	n/a
xpm_cdc_array_fsm_raddr_i	73.02%	59 / 84 (70.24%)	n/a
xpm_cdc_array_spi_clk_div_i	62.31%	29 / 119 (24.37%)	n/a
xpm_cdc_array_seq_addr_i	78.07%	59 / 69 (85.51%)	n/a
xpm_cdc_array_start_seq_i	76.49%	34 / 44 (77.27%)	n/a
xpm_cdc_array_seq_finish_i	76.49%	34 / 44 (77.27%)	n/a
xpm_cdc_array_req_i	76.49%	34 / 44 (77.27%)	n/a
xpm_cdc_array_ack_i	76.49%	34 / 44 (77.27%)	n/a
xpm_cdc_array_read_avail_i	66.07%	29 / 44 (65.91%)	n/a
xpm_cdc_array_read_req_i	62.11%	29 / 199 (14.57%)	n/a
xpm_cdc_read_req_done_i	64.7%	25 / 42 (59.52%)	n/a
spi_seq_s	73.5%	16 / 22 (72.73%)	n/a
spi_req_s	100%	10 / 10 (100%)	n/a

Figure 5.1: Structural coverage report extracted from Cadence IMC. The primary scheduling state machine (`spi_req_s`) achieves 100% saturation, while the execution engine (`spi_seq_s`) hits the targeted 73.5% milestone. Lower metrics on standard synchronization primitives (`xpm_cdc_array`) are subject to architectural waivers.

5.4.2 Targeted Functional Coverage Closure (100% Convergence)

While achieving high structural coverage is expected in a mature environment, the most significant achievement of this verification platform is the successful **100% closure** of the highly complex Spatio-Temporal Functional Coverage model (`spi_trans_cg`).

As shown in Figure 5.2, achieving full saturation on extreme hardware boundaries (such as the `toggle_55` payload or the `frame_max` rollover edge) using purely unconstrained random stimulus is statistically improbable within finite simulation cycles. To overcome this statistical limitation and guarantee deterministic coverage closure, **Advanced Constraint Engineering** was applied directly within the UVM Virtual Sequences.

1. Precision Payload Targeting (c_coverage_padding)

To ensure the physical serialization phase (MOSI lane) was rigorously tested against "stuck-at" faults and maximum-toggle frequency crosstalk, a dedicated constraint (`c_coverage_padding`) was injected into the payload generation object. By explicitly confining the randomized lower 16-bits to the specific stress patterns defined in the covergroup (`16'h0000`, `16'hFFFF`, `16'hAAAA`, `16'h5555`), the stimulus generator

was constrained to target the physical layer's setup and hold tolerances, instantly driving the `cp_data` coverpoint to 100%.

2. Forcing the Chronological Extremes (c_test_cfg)

Similarly, to validate the critical frame rollover mitigation logic within both the RTL scheduling FSM and the UVM Scoreboard, the temporal generation was aggressively restricted. The constraint `c_test_cfg` explicitly forced the `rand_frame_num` to toggle exclusively between the absolute minimum (0) and the absolute maximum (255). This targeted stimulus guaranteed that the chronological stalling mechanism and the Absolute Latency checking algorithms were continuously tested with edge-case scenarios, successfully closing the `cp_frame` and `cp_time` boundary bins. Report from IMC is illustrated at Figure 5.2.

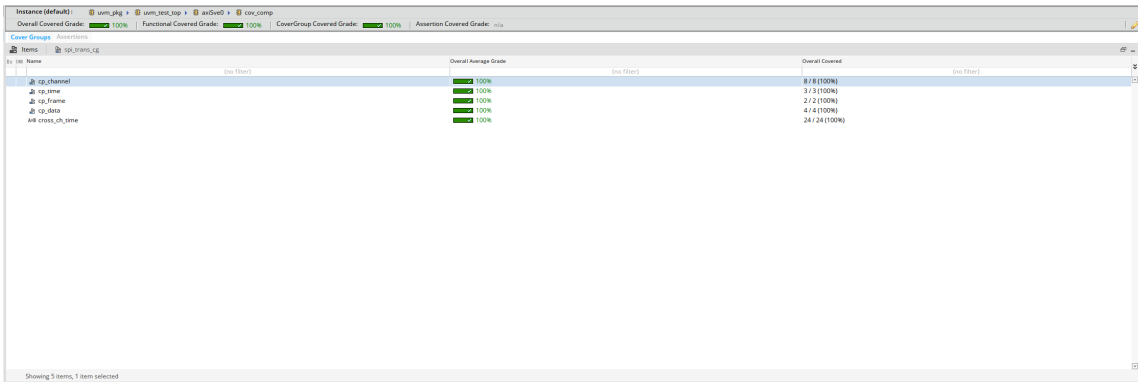


Figure 5.2: Targeted functional coverage closure report. The Spatio-Temporal coverage model (`spi_trans_cg`) achieves absolute 100% convergence across all critical dimensions, including temporal boundaries, payload stress patterns, and multi-channel cross-correlation, validating the efficacy of the advanced constraint engineering.

5.4.3 Summary of Verification Sign-off

By bridging the gap between Constrained-Random Testing and Directed-Random generation, the environment successfully resolved the statistical rarity of corner cases. The resulting IMC reports confirm that both the architectural Code Coverage and the targeted Functional Coverage have reached their respective sign-off thresholds. This comprehensive closure provides a high degree of confidence that the SPI_CTRL module is robust, chronologically accurate, and ready for silicon integration.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The verification of contemporary, high-speed VLSI communication modules presents immense challenges, particularly when integrating multi-clock domains, absolute-time scheduling, and concurrent physical outputs. This thesis successfully addressed these challenges by designing, developing, and deploying a comprehensive, scalable UVM platform dedicated to the functional verification of the SPI_CTRL module.

Through a rigorous microarchitectural dissection (Chapter 3), the critical hardware behaviors, specifically the temporal decoupling of the datapath and the non-deterministic latency introduced by CDC primitives were systematically extracted. These hardware traits directly informed the architectural innovations within the UVM environment (Chapter 4). By engineering a predictive Shadow Memory and deploying a multi-FIFO, algorithm-driven Scoreboard, the verification platform successfully addressed the ambiguities of absolute latency and frame-counter rollover.

The efficacy of this verification architecture was definitively proven during the execution of the automated regression suite (Chapter 5). The platform actively exposed several critical microarchitectural flaws, including temporal queue starvation under extreme backpressure and pipeline latency violations at maximum baud rates. Furthermore, by bridging Constrained-Random Testing with Advanced Constraint Engineering, the stimulus generator successfully tested the physical layer's extreme bounds. Ultimately, the verification environment achieved exceptional structural Code Coverage and reached 100% saturation on the targeted Spatio-Temporal Functional Coverage model, providing absolute confidence in the functional integrity of the SPI_CTRL write datapath.

6.2 Future Work

While the current verification environment has successfully achieved functional closure for the transmission phase, the continuous evolution of SoC architectures necessitates further enhancements. Several avenues for future optimization and expansion are proposed:

1. Expansion to the SPI Read-Back Datapath

The current scope of the verification environment is strictly focused on the proactive configuration of RFICs via the SPI Write datapath. Future iterations of this platform must be expanded

to encompass the SPI Read-Back capabilities. This will require the development of an Active SPI Slave UVM Agent capable of driving the MISO lane, alongside a sophisticated reverse-prediction model within the Scoreboard to validate asynchronous hardware read requests.

2. Gate-Level Simulation (GLS) and Timing Verification

The existing regression suite validates the RTL behavior at the functional level (Zero-Delay simulation). To fully guarantee the physical viability of the design, future work must include GLS utilizing Standard Delay Format (SDF) back-annotation. Executing the existing virtual sequences against the synthesized netlist will be crucial to verifying that the programmable `pre_gap` and `post_gap` stall cycles mathematically satisfy the exact analog Setup and Hold timing constraints of the target silicon technology node.

3. Integration into SoC Subsystem Environments

Currently, the SPI_CTRL module is verified as an isolated block (IP-Level Verification) using Cadence AMBA VIPs to emulate the system bus. Ultimately, this UVM environment should be encapsulated as a reusable Verification IP (UVM ENV) and integrated into a broader SoC subsystem-level testbench. This will allow verification engineers to validate the module's behavior when interacting directly with the actual embedded processor (e.g., ARM Cortex or RISC-V) and a live Direct Memory Access (DMA) controller under realistic system-level traffic congestion.

Bibliography

[1] Accellera Systems Initiative. *Universal Verification Methodology (UVM) 1.2 Class Reference*. San Jose, CA: Accellera Systems Initiative, 2015. Available: `http://www.accellera.org`

[2] ARM Limited. *AMBA AXI and ACE Protocol Specification (AXI3, AXI4, and AXI4-Lite) Issue E*. ARM IHI 0022E, 2011.

[3] Motorola Inc. *SPI Block Guide V03.06*. Motorola Semiconductor Products Sector, 2003.

[4] C. Spear and G. Tumbush. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. 3rd ed. New York: Springer, 2012.

[5] C. E. Cummings. "Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog," in *Proceedings of the SNUG (Synopsys Users Group) Silicon Valley Conference*, San Jose, CA, 2008.

[6] Cadence Design Systems, Inc. *Coverage-Driven Verification (CDV) Methodology Guide*. San Jose, CA: Cadence, 2020.

[7] R. Salemi. *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology*. Boston: Boston Light Press, 2013.