Aligner Verification Plan

Vaaluka Solutions Private Ltd

Table of Contents

1 INTRODUCTION

1.1 DESIGN DESCRIPTION

1.1.1 Module Purpose

The Aligner module serves a critical role in data handling pipelines where alignment of data is essential for ensuring efficient memory access and system performance. In digital systems, particularly those involving memory-mapped peripherals or external memory interfaces, data must often be aligned to specific byte boundaries to meet protocol or hardware constraints. Misaligned accesses can lead to inefficiencies, increased latency, or hardware faults.

This module is designed to accept a stream of incoming unaligned data—data that does not necessarily start or end on standard word boundaries—and reformat it into a properly aligned output stream according to user-specified parameters. These parameters, namely alignment size and offset, are configured through software-accessible registers.

The key goals of the Aligner module include:

- Data Stream Alignment: Ensuring that output data adheres to alignment constraints dictated by the memory system or bus protocol (e.g., 4-byte or 8-byte boundaries).
- Configurability: Allowing flexible adjustment of alignment behavior through runtime control register settings (CTRL.SIZE, CTRL.OFFSET).
- Protocol Compatibility: Supporting integration in systems that use the AMBA 3 APB protocol for control signaling and a custom Memory Data (MD) protocol for data streaming.
- Backpressure Support: Managing flow control using ready/valid handshakes to coordinate with upstream and downstream components.
- Error Detection and Reporting: Identifying and flagging illegal data transactions (e.g., unsupported alignment combinations), and generating interrupt requests for software handling.

By aligning unstructured or partially structured input data to strict output formatting, the Aligner module enables the system to perform optimized memory writes, reduce bus contention, and avoid performance penalties due to unaligned accesses. It is particularly useful in data-processing pipelines, communication interfaces, or DMA subsystems that aggregate or dissect packetized or fragmented data.

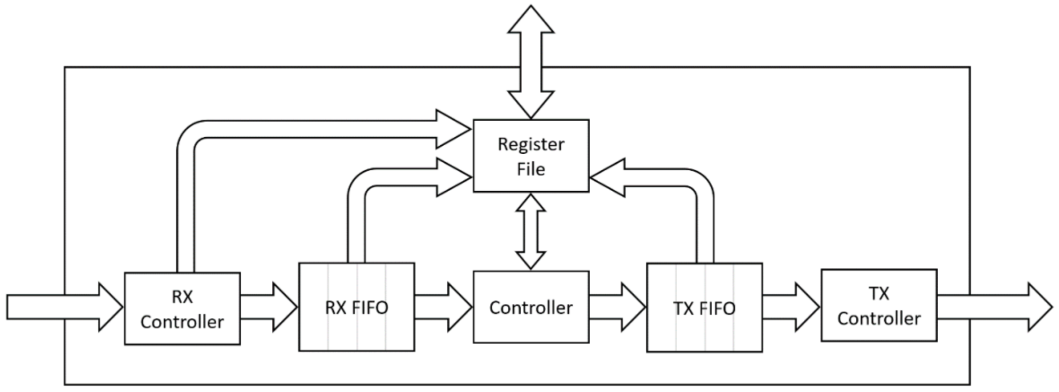
1.1.2 Functional Overview

The Aligner module functions as an intermediate logic block responsible for receiving unaligned data from an input interface, transforming that data according to alignment configuration, and outputting it in a structured and aligned format. It operates in real time and supports both backpressure and error handling mechanisms to ensure robust integration in complex SoC data paths.

At a high level, the module consists of the following functional blocks:

- Input Interface (RX Path): Receives data along with meta-information specifying where valid data starts (OFFSET) and how many bytes are valid (SIZE). These transactions are governed by a custom Memory Data (MD) protocol using a valid/ready handshake scheme.
- Control Logic: Validates input transfers against alignment rules, aligns the data based on configuration settings, and routes it to the output interface. If an input transfer is illegal (e.g., an invalid combination of OFFSET and SIZE), the module flags the error, increments an internal drop counter, and drops the data.
- FIFO Buffers (RX FIFO and TX FIFO): Two FIFO queues decouple the timing between input and output interfaces. The RX FIFO stores incoming data while the TX FIFO holds aligned data awaiting transmission. These FIFOs also serve as flow control buffers and support interrupt generation on full/empty thresholds.
- Output Interface (TX Path): Delivers aligned data using the same MD protocol as the input interface. It waits for the downstream module to assert readiness before transferring data.
- Register Bank (APB Interface): A set of configuration and status registers is exposed via the AMBA 3 APB protocol. These registers allow software to configure the alignment behavior, monitor internal status, clear counters, and enable or acknowledge interrupts.
- Interrupt Handling: The module can generate interrupts for key events such as FIFO full/empty conditions or when the dropped transfer counter reaches its maximum. These events are reflected in a dedicated interrupt status register and a single irq output signal.

The module operates synchronously with a single clock input and supports an active-low reset signal (reset_n). Its behavior is deterministic and configuration-driven, allowing it to be reused across different system architectures with minimal modification.



1.1.3 Interfaces

The Aligner module interfaces with the surrounding system using three key interfaces:

1. A standard AMBA 3 APB (Advanced Peripheral Bus) interface for register configuration and status monitoring
2. An RX (Receive) data interface using a custom Memory Data (MD) protocol to accept unaligned input data
3. A TX (Transmit) data interface, also using the MD protocol, to send out aligned data

Each interface is described below with details about signals, protocols, and functionality.

1.1.3.1 *APB Interface*

The APB interface provides access to the module's configuration and status registers. It uses a subset of AMBA 3 APB signals, following the valid–ready handshake protocol for write and read operations. The address space is word-aligned, with the least significant bits (`paddr[1:0]`) ignored.

Key signals:

Signal	Direction	Description
clk	IN	Clock signal used for synchronous operation
reset_n	IN	Active-low reset signal
psel	IN	APB select – indicates access to the peripheral
penable	IN	APB enable – indicates second phase of the transfer
pwrite	IN	Write control signal (1 for write, 0 for read)
paddr	IN	Address of the register being accessed (word-aligned)
pwdata	IN	Data to be written into the register
prdata	OUT	Data read from the register
pready	OUT	Indicates when the transfer is complete
pslverr	OUT	Indicates an error condition during APB transaction

This interface is mainly used to:
1)Configure alignment behavior (`CTRL` register)
2)Read module status (`STATUS` register)
3)Enable or acknowledge interrupts (`IRQEN`, `IRQ`)
4)Clear internal counters (`CLR` field in `CTRL` register)

1.1.3.2 *MD RX Interface*

The RX interface uses a custom Memory Data (MD) protocol and accepts unaligned input data from the upstream module or bus. The protocol includes valid–ready handshaking and transmits additional information to describe the alignment context of each data word.

Key signals:

Signal	Direction	Description
md_rx_valid	IN	Indicates when the upstream data is valid
md_rx_data	IN	Data word received from the upstream source (width = ALGN_DATA_WIDTH)
md_rx_offset	IN	Byte offset within the word where valid data starts
md_rx_size	IN	Size in bytes of the valid data (must be non-zero)
md_rx_ready	OUT	Indicates the Aligner is ready to accept data
md_rx_err	OUT	Indicates an alignment error (invalid offset/size combination)

Alignment Check Rule:
To be accepted, each transfer must satisfy:
((ALGN_DATA_WIDTH / 8) + OFFSET) % SIZE == 0

If the equation is not satisfied, the transfer is dropped, `md_rx_err` is set to 1, and the internal drop counter (`CNT_DROP`) is incremented.

1.1.3.3 *MD TX Interface*

The TX interface is structurally similar to the RX interface but functions in the opposite direction. It outputs aligned data words to the downstream module or memory subsystem, also using the MD protocol and valid–ready handshaking.

Key signals:

Signal	Direction	Description
md_tx_valid	OUT	Indicates valid aligned data is ready to be transmitted
md_tx_data	OUT	Aligned data word (width = ALGN_DATA_WIDTH)
md_tx_offset	OUT	Byte offset in the data word indicating where valid data starts
md_tx_size	OUT	Size in bytes of the valid data in the word
md_tx_ready	IN	Indicates the receiver is ready to accept data
md_tx_err	IN	Error signal from downstream (used to flag transmission failures)

1.1.3.4 ***Output Interrupt Signal***

irq —> Single-bit interrupt output. Asserted when any enabled interrupt condition is met.
The `irq` signal is asserted based on interrupt conditions such as:
1)RX FIFO full/empty
2)TX FIFO full/empty
3)Drop counter overflow

Each condition is monitored independently and can be enabled or cleared via dedicated registers.

1.1.4 Registers

The Aligner module contains a set of memory-mapped registers accessible through the APB interface. These registers configure alignment behavior, provide status monitoring, and support interrupt handling.

- Control Register (CTRL):

  Used to configure the size and offset of aligned data. Also includes a write-only bit to clear the drop counter (CNT_DROP). Writing illegal size-offset combinations generates an APB error.

- Status Register (STATUS):

  A read-only register that reports internal runtime values such as the number of dropped transfers (CNT_DROP) and the fill levels of RX and TX FIFOs.

- Interrupt Enable Register (IRQEN):

  Allows software to selectively enable interrupt sources related to FIFO states and counter overflows. Each bit corresponds to a specific condition.

- Interrupt Status Register (IRQ):

  Records active interrupt events. Uses Write-1-to-Clear (W1C) behavior and retains the flag until explicitly cleared, even if the condition resolves.

  For detailed description of internal architecture, registers, access types, design behaviour, error and interrupt handling, refer the aligner_datasheet_v_1_0.

1.2 Goals of the verification process

The primary goals of the verification process are:

- Functional correctness: Ensure the Aligner module performs data alignment accurately under all valid combinations of configuration parameters (SIZE, OFFSET) and input conditions.

- Protocol compliance: Verify proper handshake behavior on APB and MD interfaces, including valid/ready signaling, error generation, and alignment condition checking.
- Register verification: Confirm correct read/write behavior of all memory-mapped registers, including error handling for invalid accesses and correct default/reset values.
- Interrupt validation: Ensure that interrupt sources are triggered under the correct conditions and that their enable/clear mechanisms function as expected.
- Boundary and corner cases: Test edge scenarios such as FIFO full/empty conditions, invalid MD transactions, and CNT_DROP counter overflow behavior.
- Robustness: Detect and handle invalid or illegal operations gracefully, without unintended side effects.
- Traceability: Achieve traceability between specification requirements and verification tests, ensuring complete coverage.

2 Verification Environment

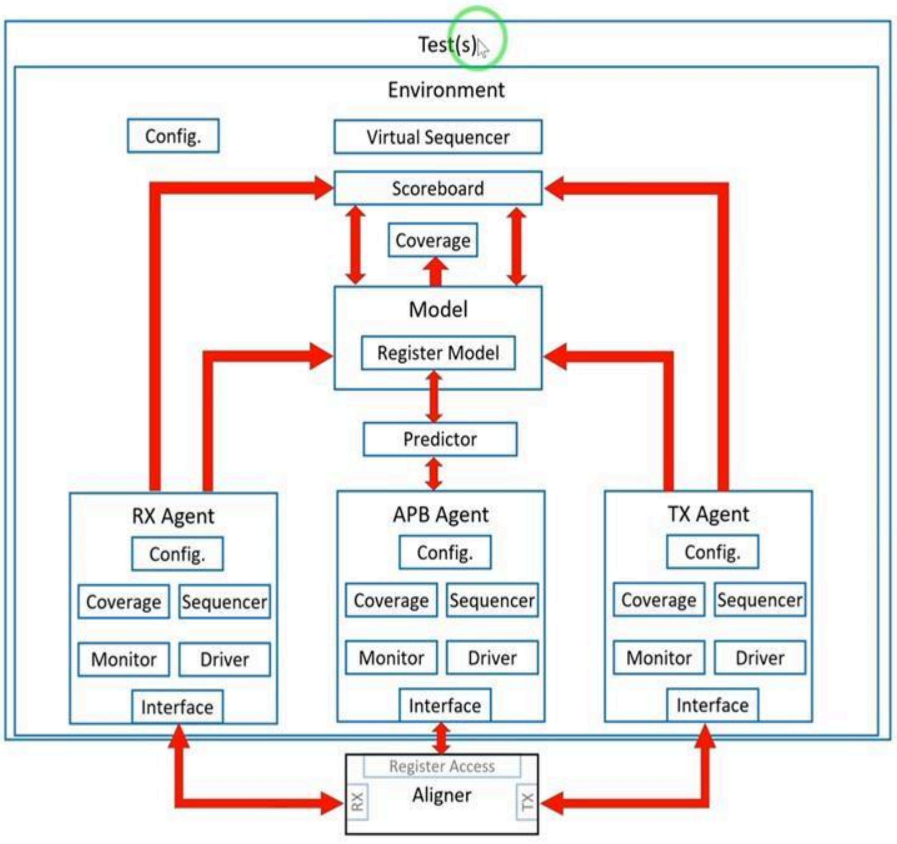
2.1 Simulation Environment

The verification environment for the Aligner module is developed using the Universal Verification Methodology (UVM) in SystemVerilog. The testbench is modular and layered, enabling reuse and scalability. It includes three agents representing each interface of the DUT:

- RX Agent for driving unaligned input data
- TX Agent for monitoring aligned output
- APB Agent for driving register transactions

The simulation is managed by a top-level test class which configures and controls the UVM environment. A virtual sequencer coordinates transaction sequences across all agents. The simulation is executed using an industry-standard simulator (e.g.,Questasim).

2.2 Verification Components



The environment consists of the following key components, as depicted in the diagram:

- **Test(s):** Top-level UVM test classes that instantiate the environment and define scenario-specific configurations, such as register settings, stimulus types, and sequence lengths.

- **Environment:** A UVM container that builds, connects, and manages all agents, models, coverage blocks, and the scoreboard. It also hosts the virtual sequencer to coordinate multiple interface agents.

- **Virtual Sequencer:** Enables synchronization and coordination of sequences across multiple agents (APB, RX, TX), ensuring that complex test scenarios are driven correctly.

- **RX Agent:**
  - Drives randomized or directed unaligned input data to the DUT via the MD RX interface.
  - Includes driver, monitor, sequencer, configuration class, and coverage collector.
- **TX Agent:**
  - Monitors the aligned output data from the DUT via the MD TX interface.
  - Includes monitor, configuration, sequencer (for checking readiness-based handshakes), and coverage.
- **APB Agent:**
  - Handles register-level read/write operations using the APB protocol.
  - Includes a monitor and driver, sequencer for register sequences, and coverage components for register access metrics.
- **Predictor:**
  - Predicts the expected aligned output data based on the received unaligned input and the current register configuration.
  - Acts as the reference model input for the scoreboard and validates DUT functionality.
- **Register Model (RAL):**
  - Mirrors the DUT's internal registers using UVM RAL model for backdoor access, checking, and synchronization.
  - Supports coverage collection and automated register testing.
- **Scoreboard:**
  - Compares actual DUT output (via TX monitor) against expected output from the Predictor.
  - Flags mismatches and functional errors.

- **Coverage Blocks:**
    - Monitor transactions and register accesses to collect functional coverage data.
    - Implemented within agents and at the environment/scoreboard level to ensure full stimulus tracking.

2.3 Simulation flow

The simulation of the Aligner module follows the **standard UVM phase-based flow**. Each phase has a distinct purpose in building, configuring, stimulating, and validating the DUT within a modular and layered testbench architecture.

Build Phase:
In this phase, all the verification components such as agents (RX, TX, APB), drivers, monitors, sequencers, the scoreboard, predictor, and the register model are constructed. Configuration objects are created and passed down to relevant components using the UVM configuration database.

Connect Phase:
Once all components are built, their ports and exports are connected to enable communication. This includes linking monitors to analysis ports of the scoreboard and coverage blocks, and connecting the virtual sequencer to the sequencers of all agents for coordinated stimulus generation.

Elaboration Phase:
This optional phase is used for printing the testbench topology and verifying the factory overrides and configuration settings. It helps in debugging the testbench hierarchy before simulation begins, ensuring that all components are correctly instantiated and connected.

Start of Simulation Phase:
At this stage, simulation logging and test-related information are initialized. Initial register states and

configuration parameters may be printed to the console, providing visibility into how the DUT and testbench are set up before the stimulus is applied.

Run Phase:
This is the main phase where active simulation occurs. The APB agent configures the DUT via register writes, and the RX agent drives randomized or directed unaligned input data. The TX agent monitors the DUT's aligned output, and the predictor generates expected results for comparison. The scoreboard checks functional correctness, while functional and register coverage is collected in parallel.

Extract Phase:
During this phase, data gathered by the monitors, scoreboard, and coverage blocks is extracted for final evaluation. This may include collecting transaction logs, coverage statistics, and predictor outputs to prepare for test status determination.

Check Phase:
In the check phase, final comparisons are made between actual and expected DUT behavior. Any mismatches detected by the scoreboard or assertion violations are reported here, and the integrity of simulation outputs is verified.

Report Phase:
This final phase summarizes the outcome of the simulation. The test status (pass/fail), error/warning counts, and functional coverage results are reported. This information is crucial for determining whether the DUT meets all functional and protocol requirements.

2.4 Reference tables

Valid and Invalid Combinations of SIZE and OFFSET

This table lists all combinations of SIZE and OFFSET for ALGN_DATA_WIDTH = 32 bits (i.e., 4 bytes). Valid combinations satisfy the condition: ((ALGN_DATA_WIDTH + OFFSET) % SIZE == 0). Cells marked in green are valid, while those in red are invalid.

SIZE \ OFFSET	0	1	2	3
1	Valid	Valid	Valid	Valid
2	Valid	Invalid	Valid	Invalid
3	Invalid	Invalid	Invalid	Invalid
4	Valid	Invalid	Invalid	Invalid

Valid Register Addresses

This table lists the valid APB register addresses for the Aligner module, along with their names and descriptions as per the specification. All addresses are word-aligned (paddr[1:0] = 00).

Address	**Register Name**	**Description**
0x0000	CTRL	Control Register
0x000C	STATUS	Status Register
0x00F0	IRQEN	Interrupt Requests Enable Register
0x00F4	IRQ	Interrupt Requests Register

3 Test Plan

3.1 Control path tests

3.1.1 Verification of CLR Bit Functionality - Write 0

Feature to be verified	Verification of design when the CLR bit in the control register is set to 0.
Feature description	To verify that when the CLR bit is set to 0 , the CNT_DROP must not change(must not reset to 0) after reaching 255.
Initial Configuration	1. IRQEN.MAX_DROP should be enabled. Other interrupts can either be enabled or disabled. 2. Set illegal values such that CNT_DROP reaches near max (send 254 times illegal packets continuously). 3. Record TX LVL. 4. Set the CLR bit as 0 in the control register.
Scenario generation	1. IRQEN.MAX_DROP should be enabled. Enable or Disable all other Interrupts. 2. Send the data patterns which are out of legal combinations..something like 3 bytes per cycle, send on any randomly selected lane twice. 3. Initiate APB read to control register.
Expected behaviour	1. CNT_DROP reaches 255 and MAX_DROP interrupt will be set on the 1st illegal packet after initial configuration. 2. CNT_DROP should be still at 255 even after the second illegal transfer. 3. md_rx_err is asserted indicating slave error.
End of test checks	1. CNT_DROP is set to 255(APB Read). 2. irq pin (Design output pin) must be asserted as drop counter reaches

	maximum value (255) and its corresponding interrupt is enabled. 3. TX_LVL must not be changed. 4. Slave error at RX side must be asserted.
How scenario creation is ensured	Functional Coverage Bin/Assertion to confirm that CNT_DROP== 255 and CTRL.CLR==0.
Component responsible for self check mechanism	MD Monitors and scoreboard.

3.1.2 Verification of CLR Bit Functionality - Write 1

Feature to be verified	Verification of design when the CLR bit in the control register is set to 1.
Feature description	Test verifying the clearing of CNT_DROP by asserting CTRL.CLR bit.
Initial Configuration	1. Enable or disable the interrupts. 2. Set the CNT_DROP = 50(which is a non zero value) by attempting to send 50 illegal transfers continuously. 3. Record the drop counter. 4. initiate an APB Read to the STATUS register.
Scenario generation	Set the CTRL.CLR bit = 1.
Expected behaviour	CNT_DROP must reset to 0.
End of test checks	CNT_DROP must reset to 0. This can be checked by initiating an APB Read to the STATUS register.
How scenario creation is ensured	Coverbins on the CTRL.CLR bit write access to confirm that a write of 1 was performed.
Component responsible for self check mechanism	Scoreboard.

3.1.3 Verification of access types of register fields

Feature to be verified	Access types of the register fields.
Feature description	Verify the access type of register fields.
Initial Configuration	1. Cycle reset for 5 clock cycles and release it. 2. Ensure all IRQ sources are disabled initially.
Scenario generation	1. Perform APB writes to IRQEN to set all bits to 1, then read back and check if they are correctly set (RW behavior). 2. Trigger an interrupt (e.g., RX_FIFO_FULL) to set one IRQ bit. 3. Read IRQ to confirm the bit is set. 4. Write 1 to that bit using APB to clear it (W1C behavior). 5. Read back to confirm the bit is cleared. (This scenario is created for verifying the access types of IRQEN and IRQ register fields. Similarly tests must be written to verify the access types of CTRL and STATUS registers.)
Expected behaviour	1. All IRQEN bits must be writable and readable. 2. All IRQ bits must be cleared only when written with 1. 3. Writing 0 to an IRQ bit must not clear it. 4. Readback of IRQEN must reflect last written value. 5. Readback of IRQ must reflect pending interrupts unless cleared via W1C.
End of test checks	1. Confirm IRQEN register reflects all bits set via read after write. 2. Confirm that triggering an interrupt sets the corresponding IRQ bit. 3. Confirm writing 1 to that bit clears it (read returns 0). 4. Confirm writing 0 to IRQ does not clear the bit. 5. Confirm all behavior aligns with RW and W1C access types per specification.

How scenario creation is ensured	Coverbins for each bit of IRQEN and IRQ register to confirm all values (0/1) are written and verified.
Component responsible for self check mechanism	Scoreboard.

3.1.4 Verification of TX_LVL and RX_LVL

Feature to be verified	Verification of TX_LVL and RX_LVL.
Feature description	To verify whether values of TX_LVL and RX_LVL at any point between ongoing transactions are correct or any corruption has taken place.
Initial Configuration	1. Cycle reset for 5 clock cycles and release it. 2. Configure the system such that CTRL.SIZE and SIZE must be equal. Also OFFSET and CTRL.OFFSET must be equal. For example, CTRL.SIZE=4,CTRL.OFFSET=0,SIZE=4,OFFSET=0. 3. Deassert ready at tx side. 4. initiate APB read to status register.
Scenario generation	1. Start driving some finite number of packets.(Any number of packets from 1 to 16). 2. For every clock cycle initiate APB Reads to STATUS register.
Expected behaviour	TX_LVL+RX_LVL must be equal to the number of packets sent at any point of time.
End of test checks	TX_LVL+RX_LVL must be equal to the number of packets sent at any point of time. This can be verified by initiating APB Reads to STATUS register.
How scenario creation is ensured	Assertions to verify the sum of TX_LVL and RX_LVL at any given point of time before this test ends must be equal to the number of packets targeted.

Component responsible for self check mechanism	MD Monitors+Scoreboard.

3.1.5 Verification of RO Mode for reserve fields

Feature to be verified	Verification of Reserved fields access type.
Feature description	To verify the access type of reserved fields in registers is read only.
Initial Configuration	Cycle reset for 5 clock cycles and release it.
Scenario generation	1. Initiate an APB Read to Reserved field of any register in the system. E.g, Bit no.3 to 7 in Control register 2. Initiate an APB write to the same bits of the same register in which previously read operation is done and try to write all those bits as 1. 3. Again initiate an APB read to the same bits of the same register.
Expected behaviour	The data retrieved in both the APB Reads should be the default value of the reserved field provided in the specification i.e, 0. (Indication that write operation is failed)
End of test checks	The data retrieved in both the APB Reads should be the default value of the reserved field provided in the specification i.e, 0.(Indication that write operation is failed)
How scenario creation is ensured	Assertions
Component responsible for self check mechanism	APB Agent.

3.2 Datapath tests

3.2.1 Verification of Data Alignment

Feature to be verified	Data Alignment.
Feature description	1. Verify the design behavior by sending 7 different valid data patterns when CTRL.SIZE = 2 && CTRL.OFFSET = 0 (Output is expected to be 2 bytes per cycle and starting at 0th lane). 2. Verify whether all the combinations of size are being transmitted correctly when corresponding packets are sent continuously (e.g, a stream of packets covering all the combinations of size from input side is being sent).
Initial Configuration	a) 1. Clear the Interrupts (If any set already). 2. Record the Drop Counter. 3. Assert ready at TX side 4. CTRL.SIZE = 2, CTRL.OFFSET = 0. b) 1. Initiate a reset action on the system. 2. Record the Drop Counter. 3. Assert ready at TX side. 4. CTRL.SIZE = 2, CTRL.OFFSET = 0.
Scenario generation	a) 1. Disable all Interrupts. 2. Assert ready at TX side. 3. Send the data patterns a) One byte per cycle, send on any randomly selected lane. b) 2 Bytes per cycle, starting from 0th(send on 0, 1) or 2nd (send on 2nd and 3rd) lane. b) 1. Assert ready at TX side.

	2. Generate streams of sequence from Rx side inorder to cover all the valid combinations of size(e.g,SIZE=1-2-4,2-1-4...etc).
Expected behaviour	1. Design should send the data onto the TX data Aligned data(2 Bytes/cycle; Starting from Lane 0) appears on TX interface. 2. Drop Counter Must not be incremented.
End of test checks	1. No data is expected on 3rd and 4th lanes 2. No Status bits to be set in STATUS register (Read the registers) FrontDoor/BackDoor 3. irq pin (Design output pin) must not be asserted as no interrupt is enabled. 4. TX interface sends aligned data with as per legal patterns
How scenario creation is ensured	Functional Coverage Bin.
Component responsible for self check mechanism	Scoreboard compares expected aligned output with DUT output.

3.2.2 Verification of design behaviour in case of valid and invalid interleaving transactions

Feature to be verified	Valid/Invalid interleaving transactions.
Feature description	Verify the design behavior when legal and illegal RX packet transfers are alternated — ensuring: • Legal packets are processed and forwarded to TX • Illegal packets are dropped and CNT_DROP is incremented
Initial Configuration	1. Set CTRL.SIZE = 2, CTRL.OFFSET = 0 (or other legal values). 2. Ensure IRQEN.MAX_DROP is enabled. 3. Set CNT_DROP = 0 by applying CLR.
Scenario generation	1. Send one legal MD packet that satisfies the alignment condition:

	((ALGN_DATA_WIDTH / 8) + offset) % size == 0 2. Immediately follow it with one illegal packet that violates the condition. 3. Alternate legal and illegal packets for several cycles (e.g., 10 total transactions). (Constrained random sequences alternating legal/illegal SIZE-OFFSET combinations.)
Expected behaviour	1. Only legal packets should reach the TX side. 2. Illegal packets should be dropped. 3. CNT_DROP should be incremented for each illegal packet. 4. No packet corruption or misalignment in TX output. 5. If CNT_DROP reaches 255, IRQ.MAX_DROP and global irq must be asserted.
End of test checks	1. TX output must contain only the expected aligned data from legal inputs. 2. CNT_DROP must equal the number of illegal packets sent. 3. irq output must be asserted if CNT_DROP == 255 and IRQ enabled.
How scenario creation is ensured	1. Coverbins tracking CNT_DROP increments and TX FIFO push events. 2. Assertions verifying: Illegal packet → error is asserted at RX side and no change in TX_LVL. Legal packet → correctly forwarded to TX, i.e, TX_LVL gets incremented.
Component responsible for self check mechanism	Scoreboard.

3.2.3 Verification of design behaviour in case of relieving back pressure

Feature to be verified	Verifying the design behaviour backpressure is relieved for sometime.
Feature description	To verify the design behaviour when both TX FIFO and RX FIFO are full and if backpressure is relieved by TX by asserting ready at TX side.
Initial Configuration	1. IRQEN.RX_FIFO_FULL AND IRQEN.TX_FIFO_FULL should be enabled. 2. Fill RX FIFO by sending randomized legal transactions. 3. Assert md_tx_ready. 4. Initiate APB read to status register.
Scenario generation	1. IRQEN.RX_FIFO_FULL AND IRQEN.TX_FIFO_FULL should be enabled. 2. Assert md_tx_ready. 3. Fill RX FIFO by sending a randomized legal transactions For e.g, SIZE=4 OFFSET=0 and CTRL.SIZE=1 and CTRL.OFFSET=0 makes the RX FIFO full faster. (Firstly TX Fifo gets filled, then deassert md_tx_ready so that TX Fifo wont take any data into it and then gradually Rx fifo gets full) 4. After Rx fifo gets filled,md_rx_ready will be deasserted. Then relieve the backpressure by making md_tx_ready as 1 for sometime.(let us say for one clock cycle).
Expected behaviour	Tx fifo should be able to take data into it after backpressure is relieved.
End of test checks	RX_LVL must be 7 at the time of relieving back pressure and then immediately it should go to 8 again after one cycle. This checking can be done by initiating APB read to the status register.
How scenario creation is ensured	Assertions indicating that after ready at TX side is deasserted , it is asserted for one clock cycle.
Component responsible for self check mechanism	Scoreboard+MD Monitor+APB Monitor.

3.2.4 Verification of data storing capability of the central controller

Feature to be verified	Data storing capability of the central controller.
Feature description	To verify whether the central controller is storing the data in the expected way.
Initial Configuration	Cycle reset for 5 clock cycles and release it.
Scenario generation	1. Configure CTRL.SIZE=4 and CTRL.OFFSET=0. 2. Deassert ready at TX side. 3. Send 4 legal packets with size=1 and offset =0 from Rx side. Every packet must be sent with a finite amount of delay(let us say 5 clock cycles). 4. Initiate APB read to status register.
Expected behaviour	1. Rx level should increase from 0 to 1 for every 5 clock cycles until initial 15 clock cycles and then immediately drop to 0 in the next clock cycles(i.e, 1st,6th,11th,16th clock cycles). 2. Tx level must be 0 until the first 15 clock cycles. At the 16th clock cycle, it must become 1.
End of test checks	1. Rx level should increase from 0 to 1 for every 5 clock cycles until initial 15 clock cycles and then immediately drop to 0 in the next clock cycles(i.e, 1st,6th,11th,16th clock cycles). 2. Tx level must be 0 until the first 15 cl;ock cycles. At the 16th clock cycle, it must become 1. 3. These can be verified by initiating APB Read to the status register.
How scenario creation is ensured	Assertions to be written for this test indicating after every 5 clock cycles data packet of size=1 and offset=0 is being sent.

Component responsible for self check mechanism	Scoreboard.

3.3 Interrupt tests

3.3.1 Verification of triggering of MAX_DROP interrupt

Feature to be verified	MAX_DROP interrupt.
Feature description	Verifying the design behaviour when MAX_DROP interrupt gets triggered.
Initial Configuration	1. IRQEN.MAX_DROP should be enabled. Other interrupts can either be enabled or disabled. 2. Set illegal values such that CNT_DROP reaches near max (send illegal packets 254 times continuously). 3. Record TX_LVL. 4. Initiate APB read to status register.
Scenario generation	1. IRQEN.MAX_DROP should be enabled. 2. Enable or Disable all other Interrupts. 3. Send the data patterns which are out of legal combinations.. something like 3 bytes per cycle, send on any randomly selected lane twice.
Expected behaviour	1. CNT_DROP reaches 255 and MAX_DROP interrupt will be set on the 1st illegal packet after initial configuration. 2. CNT_DROP should be still at 255 even after the second illegal transfer. 3. Slave error at RX side must be asserted.
End of test checks	1. CNT_DROP is set to 255(APB Read). 2. irq pin (Design output pin) must be asserted as drop counter reaches maximum value (255) and its corresponding interrupt is enabled.

	3. The data provided after initial configuration must not appear at TX side. Tx level must not be changed. 4. Slave error at RX side must be asserted.
How scenario creation is ensured	Functional Coverage Bin/Assertion to confirm that CNT_DROP== 255.
Component responsible for self check mechanism	MD Monitors and scoreboard.

3.3.2 Verification of triggering of RX_FIFO_FULL interrupt

Feature to be verified	RX_FIFO_FULL interrupt(Backpressure condition).
Feature description	Verify the design behaviour when RX Fifo gets full.
Initial Configuration	1. IRQEN.RX_FIFO_FULL should be enabled. 2. Fill RX FIFO by sending randomized legal transactions. 3. Assert ready at TX side. 4. Initiate APB read to status register.
Scenario generation	1. IRQEN.RX_FIFO_FULL should be enabled. 2. Assert ready at TX side. 3. Fill RX FIFO by sending randomized legal transactions. For e.g, SIZE=4 OFFSET=0 and CTRL.SIZE=1 and CTRL.OFFSET=0 makes the RX FIFO full faster. (Firstly TX Fifo gets filled, then deassert md_tx_ready so that TX Fifo wont take any data into it and then gradually Rx fifo gets filled)
Expected behaviour	1. RX_FIFO_FULL must be asserted. 2. ready at the RX side must be deasserted. 3. irq output pin must be asserted.
End of test checks	1. IRQ.RX_FIFO_FULL must be asserted. 2. ready at the RX side must be deasserted. 3. irq output pin must be asserted. 4. Initiate APB Read to status register. RX_LVL must be 8.

How scenario creation is ensured	Coverbins
Component responsible for self check mechanism	Scoreboard+MD Monitors+APB Monitor.

3.3.3 Verification of triggering of RX_FIFO_EMPTY interrupt

Feature to be verified	RX_FIFO_EMPTY interrupt.
Feature description	Verify the design behaviour when RX Fifo gets empty.
Initial Configuration	1. IRQEN.RX_FIFO_EMPTY should be enabled. 2. Start with RX FIFO partially filled, let us say RX_LVL =3. 3. Valid at RX Side must be deasserted. 4. Initiate APB read to status register.
Scenario generation	Assert ready at TX side. The packets at the RX side will be consumed by the TX side continuously according to the CTRL.SIZE and CTRL.OFFSET configured. For e.g, Configure CTRL.SIZE=1 and CTRL.OFFSET=2.
Expected behaviour	1. IRQ.RX_FIFO_EMPTY must be asserted. 2. irq output pin must be asserted.
End of test checks	1. IRQ.RX_FIFO_EMPTY must be asserted. 2. ready at RX side must be asserted. 3. irq output pin must be asserted. 4. Initiate APB Read to status register. RX_LVL must be 8.
How scenario creation is ensured	Coverbins
Component responsible for self check mechanism	Scoreboard+MD Monitors+APB Monitor.

3.3.4 Verification of triggering of TX_FIFO_FULL interrupt

Feature to be verified	TX_FIFO_FULL interrupt.
Feature description	Verify the design behaviour when TX_FIFO_FULL interrupts triggers.
Initial Configuration	1. Record the TX Fifo level(by reading the status register). 2. Deassert ready at TX side. 3. IRQEN.TX_FIFO_FULL should be enabled. 4. Initiate APB read to status register.
Scenario generation	1. IRQEN.TX_FIFO_FULL should be enabled. Remaining interrupt request flags can be either 0 or 1. 2. Record the TX Fifo level(by reading the status register). Let it be x. 3. Deassert ready at TX side. 4. Drive (8-x) valid packets from the RX side continuously. For e.g, SIZE=4,OFFSET=0
Expected behaviour	1. IRQ.TX_FIFO_FULL must be asserted. 2. irq output pin in the design must be asserted.
End of test checks	1. IRQ.TX_FIFO_FULL must be asserted. 2. irq output pin in the design must be asserted. 3. Initiate APB read and read the status of TX_LVL , it must be 8.
How scenario creation is ensured	1. Coverbins to verify that STATUS.TX_LVL == 8. 2. Assertions to ensure md_tx_ready == 0 was held while valid RX data was sent in.
Component responsible for self check mechanism	Scoreboard+MD Monitors+APB Monitor.

3.3.5 Verification of triggering of TX_FIFO_EMPTY interrupt

Feature to be verified	TX_FIFO_EMPTY interrupt.
Feature description	To verify the design behaviour when TX_FIFO_EMPTY gets triggered.
Initial Configuration	1. IRQEN.TX_FIFO_EMPTY should be enabled. 2. Valid at RX side must be deasserted. 3. Initiate APB read to status register.
Scenario generation	Drain all TX data by configuring a valid CTRL.SIZE and CTRL.OFFSET. No new data is passed from the RX side as valid at RX side is deasserted.
Expected behaviour	1. TX_FIFO_EMPTY must be asserted. 2. irq output pin must be asserted.
End of test checks	1. IRQ.TX_FIFO_EMPTY must be asserted. 2. irq output pin must be asserted. 3. Initiate APB Read to status register. TX_LVL must be 0.
How scenario creation is ensured	Coverbins for observing when STATUS.TX_LVL == 0
Component responsible for self check mechanism	Scoreboard+MD Monitors+APB Monitor.

3.3.6 Checking design behaviour in case of multiple interrupt triggering

Feature to be verified	Multiple interrupts triggered simultaneously.
Feature description	Verification of design behaviour when multiple interrupts triggered simultaneously.
Initial Configuration	1. Enable all the IRQEN flags. 2. Set illegal values such that CNT_DROP reaches near max ..send 254 illegal packets continuously. 3. Make the RX Fifo to be almost full. i.e, fill upto 7 levels. 4. Attempt to send one illegal packet.

	5. Send one packet such that RX fifo becomes full.
Scenario generation	1. Enable all Interrupts . 2. Send legal combinations of packets continuously to make the RX Fifo almost full. i.e, fill it up to 7 levels. 3. Send the data patterns which are out of legal combinations.. something like 3 bytes per cycle, send on any randomly selected lane . 4. Initiate APB read to status register.
Expected behaviour	1. IRQ.RX_FIFO_FULL must be asserted. 2. ready at RX side must be deasserted. 3. IRQ.MAX_DROP must be asserted. 4. irq output pin in the design must be asserted.
End of test checks	1. IRQ.RX_FIFO_FULL must be asserted. Initiate an APB Read to the status register inorder to confirm the RX_LVL is reached to 8. 2. ready at RX side must be deasserted. 3. IRQ.MAX_DROP must be asserted. Initiate an APB Read to the status register inorder to confirm the drop count is reached to 255. 4. irq output pin in the design must be asserted.
How scenario creation is ensured	Coverbins/Assertions.
Component responsible for self check mechanism	Scoreboard and MD_RX Monitor.

3.3.7 Verification of sticky behaviour of IRQ Register field

Feature to be verified	Sticky nature of IRQ Register.
Feature description	Verification of sticky nature of IRQ register field.

Initial Configuration	Cycle reset for 5 clock cycles and release it.
Scenario generation	1. Initially after asserting and deasserting the reset, enable all the interrupts -->rx fifo empty interrupt will be triggered and irq bit will be set to 1. 2. Then deassert IRQEN.RX_FIFO_EMPTY. 3. Configure CTRL.SIZE=1 and CTRL.OFFSET=0. 4. Start sending valid packets of size=4 and offset =0.(number of packets must be between 1 to 8) (This scenario is explained by taking RX_FIFO_EMPTY interrupt as an example. Similarly it applies to remaining interrupts in the IRQ register i.e, RX_FIFO_FULL, TX_FIFO_EMPTY, TX_FIFO_FULL, MAX_DROP) 5. Initiate APB read to status register and irq register.
Expected behaviour	1. IRQ.RX_FIFO_EMPTY must be asserted even after disabling IRQEN.RX_FIFO_EMPTY. 2. irq must be deasserted after disabling IRQEN.RX_FIFO_EMPTY. 3. Rx Level should not be equal to 0.
End of test checks	1. IRQ.RX_FIFO_EMPTY must be asserted even after disabling IRQEN.RX_FIFO_EMPTY. This can be done by initiating an APB read to the IRQ register. 2. irq must be deasserted even after disabling IRQEN.RX_FIFO_EMPTY.
How scenario creation is ensured	Functional Coverage Bin.
Component responsible for self check mechanism	APB Monitor.

3.3.8 Verification of re-triggering of IRQ bits only when the corresponding event reoccurs, regardless of already generated interrupt

Feature to be verified	Re-triggering of IRQ bits.
Feature description	Verification of re-triggering of IRQ bits only when the corresponding event reoccurs, regardless of already generated interrupt.
Initial Configuration	Enable all the interrupts.
Scenario generation	1. Set CNT_DROP=255 by sending 255 illegal packets continuosly. 2. After the IRQ.MAX_DROP bit gets set, deassert it using APB Write to IRQ register. 3. Make CNT_DROP=0 by writing CLR bit as 1 in the CTRL register. 4. Again send 255 illegal packets. (This scenario is explained by taking MAX_DROP interrupt as an example. Similarly it applies to remaining interrupts in the IRQ register i.e, RX_FIFO_FULL, TX_FIFO_EMPTY, TX_FIFO_FULL, RX_FIFO_EMPTY) 5. Initiate APB read to IRQ Register.
Expected behaviour	IRQ.MAX_DROP must not be reasserted even after disabling IRQ.MAX_DROP and still CNT_DROP=255. It must be reasserted only when again CNT_DROP crosses 254 and reaches 255.
End of test checks	IRQ.MAX_DROP must not be reasserted even after disabling IRQ.MAX_DROP and still CNT_DROP=255. It must be reasserted only when again CNT_DROP crosses 254 and reaches 255. This can be verified by initiating an APB read to the IRQ register.
How scenario creation is ensured	Functional Coverage Bin.
Component responsible for self check mechanism	APB Monitor.

3.4 Error injection tests

3.4.1 Verification of design behaviour during attempt to transfer illegal packet

Feature to be verified	Illegal Transfer Detection.
Feature description	Verify the design behaviour by sending invalid packets and observing change in the drop counter. DUT should drop/reject these kinds of packets which are out of the legal patterns.e.g., SIZE = 3, OFFSET = 1.
Initial Configuration	1. Interrupts can be disabled or enabled 2. Record the drop counter ; If it is already at maximum value(255), enable the CTRL.CLR bit
Scenario generation	1. Enable or Disable all Interrupts. 2. Send the data patterns which are out of legal combinations.. something like 3 bytes per cycle, send on any randomly selected lane.
Expected behaviour	1. Error at RX side must be asserted. 2. CNT_DROP increments. 3. The data must not appear at TX.
End of test checks	1. CNT_DROP increments. 2. DUT must reject the given data. 3. Error at RX side must be asserted.
How scenario creation is ensured	Functional Coverage Bin.
Component responsible for self check mechanism	MD_RX monitor.

3.4.2 Verification of design behaviour during illegal write scenario - write to RO

Feature to be verified	APB Error on Illegal Write.

Feature description	Test directing a write attempt to Status register which is read only.
Initial Configuration	Cycle reset for 5 clock cycles and release it.
Scenario generation	1)Perform an APB write to write 1's in the STATUS register fields. 2)Perform an APB reaxd to the same register.
Expected behaviour	Slave error must be asserted.
End of test checks	1. Slave error must be asserted. 2. APB read must produce all bits as 0's which is the default value of the status register.
How scenario creation is ensured	Coverbins.
Component responsible for self check mechanism	APB Agent.

3.4.3 Verification of design behaviour during illegal read condition- read to WO

Feature to be verified	APB Error on Illegal Read.
Feature description	Test directing a write attempt to CLR bit in CTRL register which is write only.
Initial Configuration	System out of reset.
Scenario generation	Perform APB read of CLR bit from CTRL register.
Expected behaviour	Slave error must be asserted.
End of test checks	Slave error must be asserted.
How scenario creation is ensured	Coverbins.
Component responsible for self check mechanism	APB Agent.

3.4.4 Verification of design behaviour during write to invalid address

Feature to be verified	APB Error on Invalid Register Address access.
Feature description	This test validates that the Aligner module correctly asserts the Slave error when an invalid (unmapped) address is accessed via the APB interface.
Initial Configuration	System out of reset.
Scenario generation	Send any kind of transaction to inaccessible address; something like a)Read: Send APB read to invalid address (e.g., 0x0004). b)Write:Send APB write to invalid address (e.g., 0x00A0). c)Randomized: Randomly generate multiple APB transactions with addresses outside valid map Access Types:Try both read (pwrite = 0) and write (pwrite = 1) operations. Valid register range: 0x0000, 0x000C, 0x00F0, 0x00F4. Invalid addresses: Any APB address not equal to above (e.g., 0x0004, 0x0010, 0x00A0).
Expected behaviour	Slave error must be asserted.
End of test checks	Slave error must be asserted.
How scenario creation is ensured	Coverage group:Tracks which ranges of invalid addresses were exercised.
Component responsible for self check mechanism	APB Agent.

3.4.5 Verification of design behaviour during configuration of invalid combinations of CTRL.SIZE,CTRL.OFFSET

Feature to be verified	APB Error due to configuration of invalid CTRL.OFFSET and CTRL.SIZE.

Feature description	Verifying the design behaviour on configuring invalid CTRL.OFFSET and CTRL.SIZE.
Initial Configuration	Configure CTRL.SIZE and CTRL.OFFSET with invalid values. For e.g, CTRL.SIZE = 2, CTRL.OFFSET = 3
Scenario generation	Perform a read/write with the given CTRL.SIZE and CTRL.OFFSET values
Expected behaviour	Slave error must be asserted.
End of test checks	Slave error must be asserted.
How scenario creation is ensured	cover bins and assertions.
Component responsible for self check mechanism	APB agent.

3.5 Clock and reset tests

3.5.1 Verification of system behaviour on initiating a reset

Feature to be verified	Effect of unexpected reset on the system.
Feature description	Test to verify whether all the entities in the design retain their default value upon applying reset.
Initial Configuration	1)Enable all the IRQEN flags. 2)Set a valid combination of CTRL.SIZE and CTRL.OFFSET. e.g, CTRL.SIZE=2,CTRL.OFFSET=0.
Scenario generation	1. Apply reset and configure the CTRL register normally via APB. 2. Start a stream of valid RX data (number of packets sent can be between 1 to 8) and observe FIFO level. 3. While RX and TX FIFOs are partially filled, issue a reset.

	4. Cycle reset for 5 clock cycles, then release it.
Expected behaviour	1. RX_LVL, TX_LVL should immediately reset to 0. 2. CNT_DROP must reset to 0. 3. All IRQ flags cleared. 4. FIFO contents Cleared, no residual data. 5. CTRL.SIZE must be 1 and CTRL.OFFSET must be 0. All other registers must also be set to their default state as per design specification.
End of test checks	1. FIFO Empty: Both RX and TX FIFOs are empty after reset (STATUS.RX_LVL = 0, TX_LVL = 0). 2. No TX Output: No output appears after reset until new RX is sent 3. CNT_DROP should be zero 4. All IRQ flags cleared 5. CTRL.SIZE must be 1 and CTRL.OFFSET must be 0. All other registers must also be set to their default state as per design specification.
How scenario creation is ensured	Coverbins/Assertions.
Component responsible for self check mechanism	Scoreboard.

3.5.2 Verification of system behaviour during clock failure

Feature to be verified	Clock Disabling.
Feature description	To verify the design behaviour when the clock gets off.
Initial Configuration	Clock must be gated with and gate with an enable signal.
Scenario generation	Assert the enable on for sometime(let us say for 10 clock cycles) and suddenly deassert it.

Expected behaviour	All the registers,the data present in both the tx and rx fifos and the output pins of the dut must get freezed, i.e, they must hold their last values before the clock is off.
End of test checks	All the registers,the data present in both the tx and rx fifos and the output pins of the dut must get freezed, i.e, they must hold their last values before the clock is off. This can be checked by initiating corresponding APB reads to the registers.
How scenario creation is ensured	Assertions/Coverage bins.
Component responsible for self check mechanism	Testbench.

3.6 PROTOCOL CHECKS

3.6.1 APB Protocol Checks

3.6.1.1 Protocol Compliant Checks

1. **DATA_WIDTH legality**
   This test confirms that the PENABLE signal is asserted exactly one cycle after PSEL, as required by the APB protocol. A valid write transaction is set up with stable address, control, and data signals. In cycle 1, PSEL is asserted while PENABLE remains low. In cycle 2, PENABLE is asserted alongside the pre-set signals. Assertions monitor this sequence and confirm that the timing aligns with protocol requirements. No assertion failures occur, indicating correct behavior.

2. **PENABLE deassertion on transfer completion**
   This test checks that PENABLE is deasserted immediately after PREADY=1, as required by the APB protocol. During a write transaction, once PREADY goes high, PENABLE is cleared in the next cycle. Assertions confirm correct timing, and the sequence aligns with PREADY to ensure protocol compliance. No assertion failures indicate correct behavior.

3. **Stable master signals during transfer**
   This test verifies that PADDR, PWDATA, and PWRITE remain stable throughout the APB transfer. All signals are set before asserting PSEL and held constant until PREADY=1. Assertions confirm

stability during the setup and enable phases. No assertion failures occur, and correct data transfer is observed. The driver locks signal values until the transaction completes.

4. **No undefined values on APB signals**
   This test checks that all APB signals stay in known logic states (0 or 1) during a transfer. The APB interface is initialized properly, and read/write transactions are performed while monitoring for any 'x' or 'z' states. Assertions confirm that all signals remain valid, and the simulation log shows no unknown values. The testbench avoids uninitialized variables to maintain signal integrity.

5. **Bounded Transfer duration**
   This test ensures the APB transfer completes within a limited number of wait states. A write is initiated with PSEL=1, PENABLE=1, and the slave delays PREADY for 3 cycles. The transfer then completes as PREADY is asserted. Assertions confirm the transfer stays within the allowed delay, with PENABLE deasserted correctly.

3.6.1.2 Protocol Violation Checks

1. **PENABLE timing check**
   This negative test verifies that PENABLE must not be asserted in the same cycle as PSEL. A write is initiated with all signals configured, but PENABLE is forcefully asserted alongside PSEL. This violates APB timing rules. The assertion detects the violation and fails as expected, with the error captured in the simulation log. The test sequence deliberately introduces this invalid condition.

2. **PENABLE timing check**
   This negative test verifies that PENABLE must not be asserted in the same cycle as PSEL. A write is initiated with all signals configured, but PENABLE is forcefully asserted alongside PSEL. This violates APB timing rules. The assertion detects the violation and fails as expected, with the error captured in the simulation log. The test sequence deliberately introduces this invalid condition.

3. **Master signals stability check**
   This test verifies that PADDR, PWDATA, and PWRITE stay constant during an APB transfer. A valid write is initiated, but address or data is intentionally changed after PENABLE is asserted. This violates protocol rules. Assertions detect the signal change and fail as expected.

4. **No unknown values on APB signals**
   This test ensures APB signals never take on undefined values (x or z) during a transfer. A driver randomly injects x/z into PADDR, PWDATA, and other lines during transactions. Assertions monitor signal integrity and fail upon detecting any undefined state. This negative scenario validates the design's robustness against signal corruption.

5. **Bounded transfer duration**
   This test verifies that the APB transfer duration does not exceed protocol limits. Unlike the compliant test, PREADY is held low for 8 cycles to simulate excessive wait states. This violates the protocol. An assertion detects the prolonged delay and fails, flagging the issue. The slave model is configured to exceed the allowed wait-state range intentionally.

3.6.2 MD Protocol Checks

3.6.2.1 Protocol Compliant Checks

1. **DATA_WIDTH legality**
   To ensure MD protocol compliance, the legality of ALGN_DATA_WIDTH must be verified by setting it to a valid power of 2, such as 32. The simulation should begin with the system out of reset and the parameter correctly applied in the configuration. No assertion failures or initialization errors should occur, confirming that the width is accepted by the RTL. This check is typically validated using RTL assertions within the initial block that enforce power-of-2 legality.

2. **DATA_WIDTH minimum**
   To ensure MD protocol compliance, ALGN_DATA_WIDTH must be set to a minimum of 8 bits. The test begins with the system out of reset, using values like 8 or 16 to verify interface initialization. The simulation should proceed without assertion failures, confirming proper handling of the minimum allowed width. This is enforced through RTL assertions that validate the parameter meets the minimum bit-width requirement.

3. **md_rx_valid hold until md_rx_ready**
   To comply with MD protocol handshake requirements, md_rx_valid must be held high until md_rx_ready is asserted. In this check, md_rx_valid is kept at 1 while md_rx_ready is initially low, ensuring that valid data is not dropped prematurely. The transfer completes successfully once md_rx_ready goes high, with md_rx_err remaining 0 and no assertions triggered. This

behavior is verified using a UVM monitor along with SystemVerilog Assertions (SVA) to confirm that the valid signal is deasserted only after the handshake completes.

4. **md_rx_data stability while valid**
   To ensure proper MD protocol behavior, md_rx_data must remain stable when md_rx_valid is high and md_rx_ready is low. In this scenario, data is driven and md_rx_valid is asserted, but md_rx_ready is held low to delay the handshake. The driver holds md_rx_data constant during this period to prevent data corruption. Successful transfer is indicated by correct data delivery, no assertion failures, and acceptance by the DUT. This check is monitored using a UVM monitor and SystemVerilog Assertions (SVA) to verify data stability throughout the handshake.

5. **md_rx_data stable till md_rx_ready**
   To meet MD protocol requirements, md_rx_data must remain constant while md_rx_valid is high and md_rx_ready is low. In this check, md_rx_valid is asserted with md_rx_ready deasserted, and the driver holds md_rx_data stable until the handshake completes. This ensures that no protocol errors or data drops occur during the wait. The behavior is validated using a UVM monitor with SystemVerilog Assertions (SVA) to confirm that the data remains unchanged until md_rx_ready is asserted.

6. **md_rx_offset valid with md_rx_valid**
   To ensure MD protocol compliance, md_rx_offset must carry a valid value whenever md_rx_valid is high. In this check, a legal offset such as 0 or 2 is driven along with md_rx_valid = 1, ensuring alignment with the data width. The data is accepted without triggering errors, and the drop counter remains unchanged. The validity of the offset is determined using the formula ((width/8) + offset) % size == 0, and this condition is monitored using a UVM monitor with SystemVerilog Assertions (SVA).

7. **md_rx_offset stable till md_rx_ready**
   To ensure MD protocol compliance, md_rx_offset must remain stable while md_rx_valid is high and md_rx_ready is low. In this scenario, a valid offset (e.g., 0) is driven along with md_rx_valid = 1, and held constant until md_rx_ready is asserted. No errors should be triggered, and the DUT must align the data correctly based on the held offset. This behavior is enforced through controlled stimulus and verified using a UVM monitor with SystemVerilog Assertions (SVA).

8. **md_rx_size valid with md_rx_valid**
   To comply with the MD protocol, a valid md_rx_size must be driven whenever md_rx_valid is asserted. In this check, a legal size value such as 4 is used along with md_rx_valid = 1, ensuring a correct size-offset combination. The data is properly aligned, transmitted, and accepted without any assertion failures or packet drops. This scenario is implemented using a directed sequence with known legal values and is verified using a UVM monitor with SystemVerilog Assertions (SVA).

9. **md_rx_size stable till md_rx_ready**
The md_rx_size signal must remain stable from the moment md_rx_valid is asserted until the handshake completes with md_rx_ready = 1. This ensures protocol compliance, as any premature change in md_rx_size during this phase is considered a violation. The sequence driver is responsible for holding the md_rx_size constant throughout the transaction. On the verification side, the UVM monitor includes a SystemVerilog Assertion (SVA) that checks for stability of md_rx_size during the handshake window. A valid transfer is recognized only when both md_rx_valid and md_rx_ready are high, upon which the data is accepted, and md_rx_err must remain deasserted. This check ensures that size-related control information is not altered mid-transaction, contributing to both data integrity and protocol correctness.

10. **Valid md_rx_size ≠ 0**
The md_rx_size signal must be non-zero during a valid transfer to ensure proper data movement, as a size of zero is considered invalid in the MD protocol. During stimulus generation, the sequence enforces this rule by driving md_rx_size = 2 (or any non-zero value) along with md_rx_valid = 1, indicating the start of a normal data transfer. The UVM monitor uses a SystemVerilog Assertion (SVA) to flag any instance where md_rx_size == 0 while md_rx_valid is asserted, as this represents an invalid setup. Successful completion of the transaction is marked by the handshake (md_rx_valid && md_rx_ready), with no assertion failures, no increment in the drop counter, and md_rx_err remaining low, confirming that the non-zero size requirement was met and the data was accepted correctly.

11. **md_rx_err valid only on handshake**
The md_rx_err signal must be asserted only during a valid handshake, i.e., when both md_rx_valid and md_rx_ready are high, ensuring that errors are flagged precisely at the point of data acceptance. It is invalid to raise md_rx_err before the handshake occurs, even if the size or offset values are incorrect. To validate this behavior, the sequence sets up a scenario with invalid parameters and initiates the handshake to intentionally trigger the error. The UVM monitor includes a SystemVerilog Assertion (SVA) that ensures md_rx_err is asserted only during a valid handshake and not before. If the error is flagged at the correct time due to protocol violation, no assertion failure occurs. This check confirms that error signaling is tightly coupled with data acceptance timing, maintaining the integrity of the error reporting mechanism.

12. **md_rx_err only high on md_rx_valid + md_rx_ready**
The md_rx_err signal must be asserted only during a valid protocol handshake—specifically when both md_rx_valid and md_rx_ready are high—ensuring that error reporting aligns strictly with data acceptance timing. Even in cases where an invalid condition exists (e.g., an incorrect offset), the error must not be raised prematurely. To test this, the sequence drives legal data with an intentionally invalid offset and initiates a proper handshake to trigger the error. The UVM monitor, equipped with a SystemVerilog Assertion (SVA), verifies that md_rx_err is asserted only when both md_rx_valid and md_rx_ready are high. The check passes as long as the error

indication occurs exactly at the handshake and not before, ensuring the protocol's error timing semantics are strictly followed.

13. **md_rx_valid is known**
    The md_rx_valid signal must always hold a known binary value—either 0 or 1—throughout simulation, as undefined (x) or high-impedance (z) states can lead to unpredictable protocol behavior and invalid functional results. To enforce this, all interface signals, including md_rx_valid, are properly initialized and driven with clean values from the beginning of the simulation. The sequence ensures no undefined values are introduced, while the UVM monitor includes a SystemVerilog Assertion (SVA) to continuously check that md_rx_valid remains within legal logic levels. This check helps maintain signal integrity and prevents propagation of indeterminate states, ensuring that no assertion failures occur due to uninitialized or floating signals in the testbench environment.

14. **md_rx_ready valid only when md_rx_valid = 1**
    The md_rx_ready signal must only be asserted when md_rx_valid is high, indicating that valid data is present for transfer. Asserting md_rx_ready when md_rx_valid = 0 would represent an invalid handshake attempt and violates the MD protocol. To validate this, the testbench includes scenarios where md_rx_valid = 0, and the monitor checks that md_rx_ready remains low during such periods. The UVM monitor uses a SystemVerilog Assertion (SVA) to flag any instance where md_rx_ready is high without a corresponding md_rx_valid, ensuring no unintended data transfer occurs. This check confirms that the slave interface responds only to valid requests, maintaining proper handshake semantics and preventing protocol violations.

15. **md_rx_offset + md_rx_size ≤ data width**
    The combination of md_rx_offset and md_rx_size must satisfy the condition (offset + size) ≤ (ALGN_DATA_WIDTH / 8) to ensure that the data transfer remains within the valid byte range of the data bus. For example, with ALGN_DATA_WIDTH = 32, the sum of offset and size must not exceed 4. A valid case like offset = 2 and size = 2 ensures correct alignment and usage of the data bus. The sequence uses constrained randomization to generate only legal combinations, ensuring that the data fits within the permissible range. During such valid transactions, the data is accepted, no md_rx_err is asserted, and normal transmission behavior is observed at the output. The UVM monitor uses a SystemVerilog Assertion (SVA) to continuously check that (md_rx_offset + md_rx_size) ≤ 4, ensuring strict adherence to protocol limits and preventing misaligned or out-of-bound data accesses.

16. **Transfer must not have infinite length**
    Each transfer must complete within a finite and reasonable time to avoid protocol hangs; specifically, md_rx_valid must not remain high indefinitely without a response. To enforce this, the sequence asserts md_rx_valid = 1 and ensures that md_rx_ready is driven high within a defined time window (e.g., within 10 cycles), completing the handshake. If md_rx_ready is not asserted within this window, it indicates a protocol timeout. The driver is responsible for

maintaining this response-time constraint, while the UVM environment includes both a timeout monitor and a SystemVerilog Assertion (SVA) to detect cases where md_rx_valid remains stuck high without a valid handshake. A successful transfer is marked by handshake completion within the allowed time and no timeout events, ensuring robust and deadlock-free protocol behavior.

3.6.2.2 Protocol Violation Checks

1. **DATA_WIDTH legality**
   This test verifies that the ALGN_DATA_WIDTH parameter is restricted to powers of 2. After reset, an illegal value like 20 is passed via parameter override. This should trigger an assertion failure, ensuring the parameter legality is enforced. The testbench applies the override during setup to validate this constraint.

2. **DATA_WIDTH minimum**
   This test ensures that ALGN_DATA_WIDTH is not set below the minimum legal value of 8 bits. After reset, the interface is instantiated with invalid values like 7 or 20. These violations should trigger assertion failures, which are confirmed through logged assertion messages. The testbench applies these overrides to validate minimum parameter enforcement.

3. **md_rx_valid hold till md_rx_ready**
   This test ensures md_rx_valid remains high until md_rx_ready is asserted. After reset, md_rx_valid is forced high and then incorrectly deasserted before md_rx_ready goes high. This violation should trigger md_rx_err = 1 and increment STATUS.CNT_DROP. The test also checks that IRQ.MAX_DROP is asserted, validating protocol enforcement. A directed sequence is used to apply this invalid stimulus.

4. **md_rx_data is valid while md_rx_valid is high**
   This test ensures that md_rx_data remains stable while md_rx_valid is high and md_rx_ready is low. Post reset, md_rx_data is deliberately toggled during this period, violating the MD protocol. The assertion detects this and fails, with a message logged in the simulation. A directed invalid scenario is used to trigger this behavior.

5. **md_rx_data stable till md_rx_ready**
   This test verifies that md_rx_data remains constant from the assertion of md_rx_valid until

md_rx_ready goes high. During the test, with md_rx_valid = 1 and md_rx_ready = 0, the data is incorrectly changed mid-handshake. This triggers an assertion failure, which is logged.

6. **md_rx_offset valid with md_rx_valid**
   This test checks that md_rx_offset remains stable while md_rx_valid is high. During the test, with md_rx_valid = 1, md_rx_offset is deliberately toggled to violate protocol requirements. The assertion detects the instability and fails, with an error message logged. This behavior is driven by an invalid toggle stimulus.

7. **md_rx_offset stable till md_rx_ready**
   This test ensures that md_rx_offset stays constant from md_rx_valid = 1 until md_rx_ready is asserted. In this scenario, md_rx_offset is toggled while md_rx_ready = 0, and then md_rx_ready is asserted. This violates the protocol and triggers an assertion failure, logged for review.

8. **md_rx_size valid with md_rx_valid**
   This test checks that md_rx_size remains stable while md_rx_valid is high. The test sets md_rx_valid = 1 and then toggles md_rx_size, violating the protocol requirement. An assertion catches this change and fails, with a message logged in the simulation output. The scenario is driven by an invalid toggle to confirm enforcement.

9. **md_rx_size stable till md_rx_ready**
   This test ensures that md_rx_size remains constant from the time md_rx_valid = 1 until md_rx_ready is asserted. In the scenario, md_rx_size is toggled while md_rx_ready = 0, followed by asserting md_rx_ready = 1. This violates the MD protocol, triggering an assertion failure.

10. **Invalid md_rx_size = 0**
    This test verifies that sending md_rx_size = 0 during a valid RX transaction is illegal. The RX input is driven with md_rx_valid = 1 and md_rx_size = 0. As per protocol, this triggers md_rx_err = 1, increments STATUS.CNT_DROP, and no data is processed. If enabled, IRQ.MAX_DROP is also asserted.

11. **md_rx_err valid only on md_rx_valid + md_rx_ready**

    This test confirms that md_rx_err is asserted only when both md_rx_valid and md_rx_ready are high. An invalid transfer is initiated using md_rx_offset = 3, md_rx_size = 2 with md_rx_valid = 1. Before the handshake completes (md_rx_ready = 1), md_rx_valid is deasserted. If md_rx_err still asserts, the protocol is violated, triggering an assertion failure. The test ensures that error signaling is restricted to valid transaction windows.

12. **md_rx_err high only on md_rx_valid + md_rx_ready**
    This test verifies that md_rx_err is asserted only when both md_rx_valid and md_rx_ready are high. An invalid RX transfer is driven using md_rx_offset = 3, md_rx_size = 2 while md_rx_valid = 1. Before the DUT can respond with md_rx_ready, md_rx_valid is deasserted. If md_rx_err still asserts outside this valid handshake window, an assertion fails. This confirms that error signaling must only occur during a valid transaction.

13. **md_rx_valid cannot be unknown**
    This test ensures that md_rx_valid never takes on an undefined value (X or Z). The scenario injects md_rx_valid = X during simulation. This violates signal integrity and must trigger an assertion failure. The test confirms protocol robustness and logs the failure, validating the design's handling of undefined control signals.

14. **md_rx_ready valid only when md_rx_valid is high**
    This test ensures that md_rx_ready is only asserted when md_rx_valid is high. The test holds md_rx_valid = 0 for multiple cycles. If md_rx_ready is asserted during this period, it violates the MD handshake protocol. An assertion detects and logs this failure, ensuring no handshake occurs without a data request.

15. **md_rx_offset + md_rx_size must ≤ bus width**
    This test ensures that md_rx_offset + md_rx_size does not exceed (ALGN_DATA_WIDTH / 8). A transaction is initiated with md_rx_offset = 4 and md_rx_size = 6 on a 32-bit bus. This illegal configuration triggers md_rx_err = 1, prevents data transfer, and may raise an APB error. The test confirms that invalid combinations are correctly handled by the design.

16. **MD transfer can not have an infinite length**
    This test ensures that an MD transfer does not remain pending indefinitely. A valid transfer is

initiated with md_rx_valid = 1 and valid data, but md_rx_ready is never asserted by the DUT. If md_rx_valid stays high for more than N cycles without md_rx_ready, it indicates a protocol violation. A monitor tracks the stall duration and triggers a timeout error if the threshold is exceeded, ensuring timely transaction completion.

4 Code/Functional/Assertion Coverage plan

4.1 Description of code coverage metric

4.2 Goal of each coverage metric

4.3 Methodology for measuring code coverage

5 Bug tracking

5.1 Methodology for tracking and reporting bugs

5.2 Reporting format of each bug

5.3 Bug severity levels and priority

6 Sign-off criteria

6.1 Criteria for deciding when verification is completed

6.2 Criteria for deciding if the design is ready for tapeout

6.3 Verification closure process