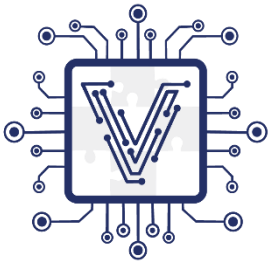


Aligner Verification Plan

Vaaluka Solutions Private Ltd

Table of Contents

1 INTRODUCTION

1.1 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.

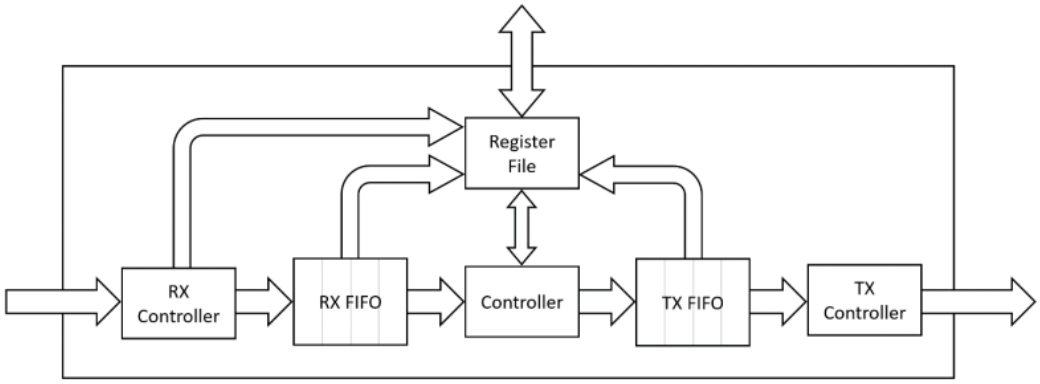
1.2 FUNCTIONAL OVERVIEW

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.



1.3 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.

1.3.1 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.

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).

1.3.2 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.

1.3.3 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.

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)

1.3.4 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.

1.4 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.

1.5 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.

1.6 VERIFICATION ENVIRONMENT

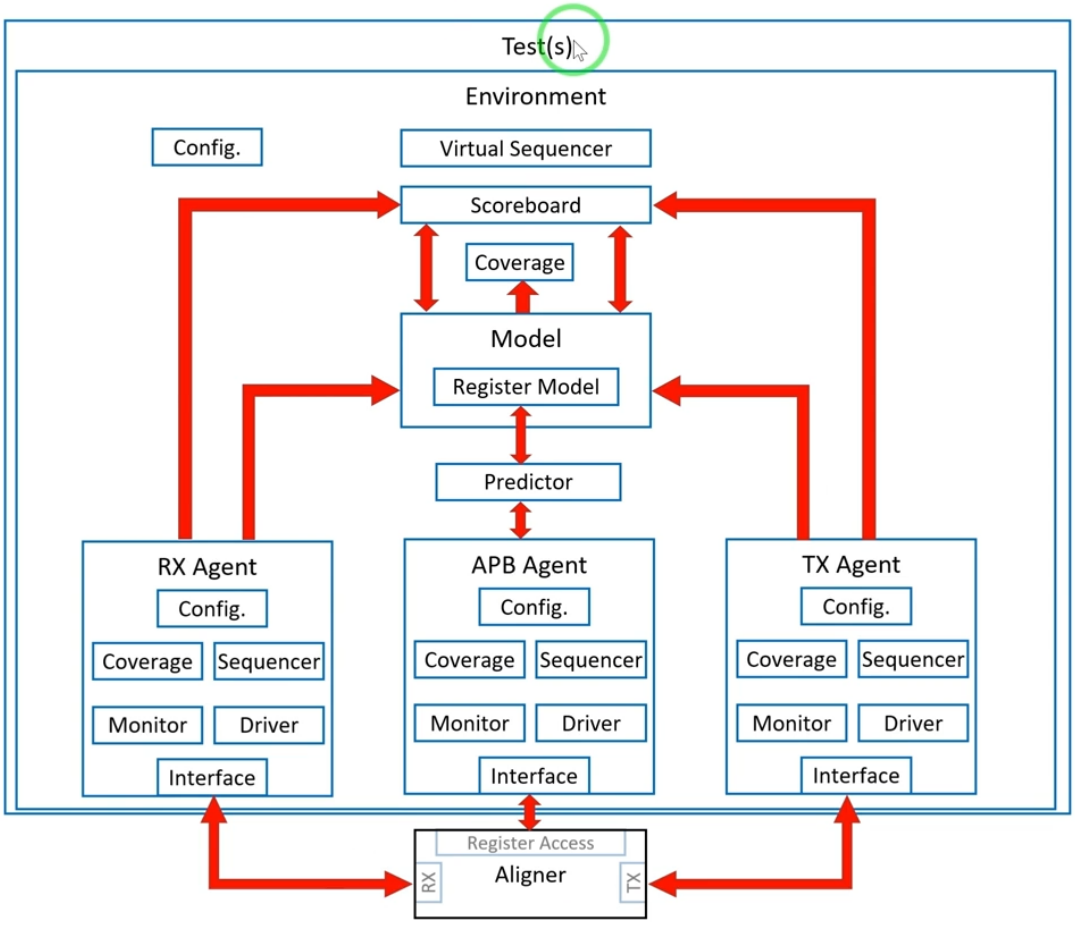
1.6.1 SIMULATION ENVIRONMENT

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).

1.6.2 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.

1.7 General Configurations

1.7.1 Legal Combinations of `CTRL.SIZE` and `CTRL.OFFSET`

- The design supports up to 4-byte lanes (lane 0 to lane 3).
- Legal combinations must be:
  - `SIZE = 1: OFFSET = 0, 1, 2, 3`
  - `SIZE = 2: OFFSET = 0, 2`

- `SIZE = 4: OFFSET = 0`

Each configuration defines how many bytes should be output per cycle and from which lane the data alignment starts.

1.7.2 General Configuration for a Valid Transaction

1. Ensure system reset is properly applied and deasserted to bring DUT into a known good state.
2. Configure DUT registers through legal APB transactions:
    a. Set `CTRL.SIZE` and `CTRL.OFFSET` to a valid combination (see legal combinations above).
    b. Optionally configure `IRQEN` to enable or disable interrupts.
    c. Ensure system clock is stable and running.
    d. Protocol Signal Requirements:
        i. `md_rx_valid` must be asserted when RX data is valid and ready to be sent into the DUT.
        ii. `md_rx_ready` from DUT must be high to accept data.
        iii. Only when both `md_rx_valid = 1` and `md_rx_ready = 1`, data is considered valid for capture.
        iv. Similarly, `md_tx_valid` must be asserted by the DUT when valid TX data is available.
        v. `md_tx_ready` from the sink must be high to accept data from DUT.
        vi. TX data transfer occurs only when both `md_tx_valid = 1` and `md_tx_ready = 1`.
3. Provide RX data aligned according to the legal combinations to continue a valid transaction.
4. The DUT should:
    a. Align and buffer incoming data as per `CTRL.SIZE` and `OFFSET`.
    b. Drive aligned data onto the TX interface.
    c. Maintain accurate FIFO level indicators and status flags in the `STATUS` register.
5. Any violations in protocol (e.g., asserting `md_rx_valid` when `md_rx_ready = 0`) should not cause data corruption.
6. Illegal conditions (e.g., overflows, misaligned transfers) must be handled cleanly by incrementing `STATUS.CNT_DROP` and optionally asserting `md_rx_err`.

Add addresses of the registers present in the DUT.

2 DATA PATH FUNCTIONAL VERIFICATION

2.1 DATA ALIGNMENT

To verify the Aligner IP's ability to properly align input data based on different `CTRL.SIZE` and `CTRL.OFFSET` configurations, ensuring that valid data is correctly transmitted through the TX interface without data corruption, loss, or misalignment.

2.1.1 Initial Configuration Steps

1. Reset the system.
2. Disable all the Interrupts in `IRQEN` register.
3. Record the initial value of the Drop Counter.
4. Program `CTRL.SIZE` and `CTRL.OFFSET` with the values for the current test.

2.1.2 Scenario Generation

Randomly,

1. Initiate and send the applicable configuration data packets to the DUT with `md_rx_valid=1`
    a. Cover all the configurations.
    b. 1 Byte per cycle
    c. 2 Bytes per cycle
    d. 4 Bytes per cycle
2. Modify `CTRL.SIZE` and `CTRL.OFFSET` with all the applicable values and repeat the test.
3. Repeat this test by interleaving different input combinations to emulate realistic traffic conditions.

2.1.3 Expected Behavior

1. TX interface outputs aligned data as per the `CTRL.SIZE` and `CTRL.OFFSET` values.
2. Data appears only on the valid lanes, starting from the specified offset.
3. No assertion of interrupt (`irq`) line, since interrupts are disabled.
4. `STATUS` register remains clear of any error/status flags.
5. Drop Counter (`STATUS.CNT_DROP`) must not increment, indicating no data loss.

2.1.4 End of Test Checks

1. Verify the TX output log for proper alignment and completeness.
2. Read Drop Counter (`STATUS.CNT_DROP`): Must match initial value.
3. Read `STATUS` register via both Front Door or Back Door: Must remain unchanged.
4. Confirm `irq` pin was never asserted during the test.

2.2 ILLEGAL TRANSFER DETECTION

To verify that the Aligner IP correctly detects and handles illegal data transfer conditions, where the incoming data width or alignment violates the current `CTRL.SIZE` and `CTRL.OFFSET` configuration. The IP must not forward misaligned or invalid-width data and must increment the drop counter and set status bits where applicable.

2.2.1 Initial Configuration Steps

1. Reset the system.
2. Disable all the Interrupts in `IRQEN` register.
3. Record the initial value of the Drop Counter, if it is already at maximum value (255), enable the `CTRL.CLR` bit.
4. Program `CTRL.SIZE` and `CTRL.OFFSET` with the applicable values for the current test.

2.2.2 Scenario Generation

1. For each legal configuration in `CTRL` register:
    a. Send **illegal data** where:
        i. The SIZE and OFFSET of the incoming data is out of the applicable configurations.
        ii. Assert `md_rx_valid=1` even for the invalid combinations and drive them to DUT.
    b. Randomly:
        i. Switch between illegal and legal transfers within the same stream.
        ii. Send legal data to ensure that illegal transfers are distinguishable and isolated in behavior.
        iii. Introduce back-to-back illegal patterns to test robustness.
    c. Emulate real-time system behavior by injecting **variable bursts** with mixed legality.
    d. Count the number of incoming packets to the DUT.
2. Make sure to not cross 255 data packets to avoid mismatch between actual number of packets driven and the number of illegal packets count. This is because the maximum value of `STATUS.CNT_DROP` is 255, once reached, it will saturate unless cleared by `CTRL.CLR` bit.

2.2.3 Expected Behavior

1. Invalid data packets must be dropped silently, i.e., no data appears on TX output.
2. `CNT_DROP` counter must increment for each dropped illegal data packet.
3. No interrupt should be raised if IRQs are disabled.
4. The DUT must not enter deadlock or undefined state.
5. TX interface must only show aligned, legal data—never partial or malformed output.

2.2.4 End of Test Checks

1. Read and compare `STATUS.CNT_DROP`: Must have increased from initial value by the number of detected illegal data packets.

2. Signal `md_rx_err` is asserted indicating error.
3. Verify that no TX output occurred for illegal transfers.
4. Confirm that `irq` pin remains de asserted throughout the test.

2.3 VALID / INVALID TRANSACTIONS INTERLEAVED

To verify that the Aligner IP handles a mixed stream of valid and invalid media data (MD) transactions without disruption. Legal transactions must be processed and produce aligned output, while illegal transactions must be:

1. Dropped without affecting ongoing traffic.
2. Detected and signaled via `md_rx_err`.
3. Tracked via incrementing `STATUS.CNT_DROP`.
4. Not allowed to corrupt FIFO levels or alignment logic.

2.3.1 Initial Configuration Steps

1. Reset the system.
2. Program `CTRL.SIZE` and `CTRL.OFFSET` randomly with valid combinations.
3. Enable `IRQEN.MAX_DROP`.
4. Initialize and log `STATUS.CNT_DROP` register.
5. Ensure `md_rx_ready` is set to 1 (ready to receive).

2.3.2 Scenario Generation

1. Legal MD Transfers**:** Send packets with valid SIZE and OFFSET.
2. Illegal MD Transfers: Send packets with invalid SIZE and OFFSET.
3. Inject 10–20 alternating legal and illegal MD transactions.

2.3.3 Expected Behavior

Behavior Element	Legal Transfers	Illegal Transfers
`Md_rx_err`	0	1 (for 1 clock cycle)
TX Stream	Data with valid SIZE and OFFSET appears	No Data with invalid SIZE and OFFSET is transmitted.
`STATUS.CNT_DROP`	Unchanged	Increments for every illegal data packet.
`STATUS.RX_LVL`	Increases and decreases as packet enter and leave	No change
`STATUS.TX_LVL`	Increases and decreases as packet enter and leave	No change
`IRQEN.MAX_DROP`	Doesn't affect	Asserted if `STATUS.CNT_DROP` hits 255

2.3.4 End of Test Checks

1. **TX Output**: Matches expected data only for legal transfers.
2. **CNT_DROP**: Equal to number of illegal transfers.

3. **RX/TX FIFO Levels**: Reflect only legal traffic; no overflow or deadlock.
4. **`md_rx_err`**: Pulsed (1-cycle assertion) only during illegal transactions.
5. **IRQ**: Asserted and latched if `IRQEN.MAX_DROP` is reached and remains until cleared.

2.4 CHANGING LEGAL REGISTER VALUES MID-STREAM

To verify that the Aligner IP correctly handles dynamic changes to `CTRL.SIZE` and `CTRL.OFFSET` during an active RX to TX stream. The design must continue to transmit aligned data seamlessly under the updated configuration without any corruption, overlap, or data loss.

2.4.1 Initial Configuration Steps

1. Apply reset to the DUT.
2. Program `CTRL.SIZE` = 1 and `CTRL.OFFSET` = 0 via APB interface.
3. Ensure `IRQEN` is disabled.
4. Record the initial value of `STATUS.CNT_DROP`.

2.4.2 Scenario Generation

1. Begin transmitting a stream of valid MD packets (md_rx_valid = 1), formatted according to the initial configuration.
2. After 4 valid transfers:
    a. Issue a legal APB write to update CTRL.SIZE and CTRL.OFFSET to a new valid combination (e.g., {2,2}).
3. Continue the RX stream with new packets that match the updated configuration.
4. Monitor the TX interface throughout:
    a. Confirm first 4 outputs match initial config.
    b. Confirm subsequent outputs match updated config.
5. Optionally interleave updates to other valid configurations to evaluate repeated dynamic reconfiguration.

2.4.3 Expected Behavior

1. First 4 TX outputs must be aligned as per the initial configuration (`CTRL.SIZE`=1, `OFFSET`=0).
2. Subsequent TX outputs must reflect the new configuration (`CTRL.SIZE`=2, `OFFSET`=2).
3. TX data must be cleanly aligned—no duplication, data corruption, or misalignment across configuration transition.
4. CNT_DROP must not increment.
5. No assertion of `md_rx_err`.
6. `irq` pin must remain de asserted.
7. FIFO levels must reflect consistent RX and TX flows.

2.4.4 End of Test Checks

1. Analyse TX stream log:
    a. Confirm 4 aligned data cycles from the initial setting.
    b. Confirm aligned output with new setting from 5th cycle onward.

2. Read `STATUS.CNT_DROP`: Must match the initial recorded value.
3. `md_rx_err` signal must remain LOW throughout.
4. Confirm `irq` pin remained inactive.

3 FIFO BEHAVIOR AND STRESS CONDITIONS

3.1 RX FIFO FULL

To verify the behavior of the Aligner IP when the RX FIFO reaches its maximum capacity, ensuring the design halts incoming MD transactions, sets the appropriate `IRQ` flag, and preserves data integrity.

3.1.1 Initial Configuration Steps

1. Apply system reset.
2. Set `md_tx_ready` = 0 to fill the TX FIFO, preventing RX to TX transfer.
3. Enable `IRQEN.RX_FIFO_FULL` interrupt.
4. Confirm initial `STATUS.RX_LVL` and `STATUS.CNT_DROP` values.

3.1.2 Scenario Generation

1. Drive legal MD packets (valid SIZE, OFFSET pairs) into RX interface with `md_rx_valid` = 1.
2. Repeat the data transfers until RX FIFO level reaches full (8 entries).
3. Monitor and log `md_rx_ready` signal.
4. Attempt to push a 9th MD transaction beyond FIFO capacity.
5. Observe `irq` pin and `IRQ.RX_FIFO_FULL` register bit.
6. Record `STATUS.RX_LVL` from status register.

3.1.3 Expected Behavior

1. When RX FIFO is full, `md_rx_ready` should assert LOW (0).
2. No new MD transfers should be accepted.
3. `IRQ.RX_FIFO_FULL` bit must be set in IRQ register.
4. `irq` pin should be asserted HIGH.
5. `STATUS.RX_LVL` register field should read `4'b1000`.
6. FIFO contents should remain consistent and error-free.

3.1.4 End of Test Checks

1. Verify `md_rx_ready` deasserted when FIFO is full.
2. Check that `irq` pin is HIGH and `IRQ.RX_FIFO_FULL` is set.
3. Confirm no overflow or data corruption occurred in FIFO.
4. Validate that `md_rx_ready` reasserts HIGH once FIFO space is available.
5. Confirm `STATUS.RX_LVL` accurately reflects FIFO status throughout.

3.2 RX FIFO EMPTY

To verify the behavior of the Aligner IP when the RX FIFO becomes empty. The design should assert the `IRQ.RX_FIFO_EMPTY` interrupt, maintain correct sticky behavior, and allow smooth resumption of RX stream without data loss or corruption.

3.2.1 Initial Configuration Steps

1. Apply system reset.
2. Partially fill the RX FIFO with a few legal MD packets.
3. Set `md_rx_valid = 0` (i.e., stop feeding RX data).
4. Enable `IRQEN.RX_FIFO_EMPTY` interrupt.

3.2.2 Scenario Generation

1. Set `md_tx_ready = 1` to allow transmission and drain RX FIFO.
2. Allow the FIFO to gradually empty as data flows from RX to TX.
3. Monitor for assertion of `IRQ.RX_FIFO_EMPTY` and the `irq` pin.
4. Confirm that `STATUS.RX_LVL` reduces to `4'b0000`.
5. Optionally, resume RX stream after IRQ is asserted and observe correct behavior.

3.2.3 Expected Behavior

1. Once RX FIFO is completely drained, `IRQ.RX_FIFO_EMPTY` bit is set.
2. `irq` pin is asserted HIGH for 1 clock cycle.
3. `IRQ.RX_FIFO_EMPTY` bit remains set (sticky behavior) until cleared explicitly.
4. `STATUS.RX_LVL` register reads `4'b0000`.
5. FIFO remains valid and intact — no corruption or side effects.

3.2.4 End of Test Checks

1. Verify that `IRQ.RX_FIFO_EMPTY` is set once FIFO is empty.
2. Check that `irq` pin is asserted for 1 clock cycle.
3. Confirm that `STATUS.RX_LVL = 4'b0000`.
4. Ensure system allows new MD data to resume without issue.
5. Optionally, clear `IRQ.RX_FIFO_EMPTY` and verify it is cleared properly.

3.3 TX FIFO FULL

To validate the behavior of the Aligner IP when the TX FIFO becomes full. The design must assert `IRQ.TX_FIFO_FULL`, stop processing aligned RX to TX transfers, and ensure no data loss. The TX stream should resume seamlessly once space becomes available.

3.3.1 Initial Configuration Steps

1. Apply system reset.
2. Set `md_tx_ready = 0` to prevent the TX FIFO from draining.
3. Enable `IRQEN.TX_FIFO_FULL`.
4. Begin sending legal MD packets to fill the TX FIFO.

3.3.2 Scenario Generation

1. Generate and transmit a sequence of legal MD transactions.
2. Monitor FIFO levels as data moves from RX to TX FIFO.
3. Continue transmitting until `STATUS.TX_LVL = 4'b1000` (full).

4. Monitor `irq` pin and `IRQ.TX_FIFO_FULL` register bit.

3.3.3 Expected Behavior

1. No data on TX Interface due to `md_tx_ready = 0`.
2. `IRQ.TX_FIFO_FULL` is asserted.
3. `irq` pin goes high for 1 clock cycle.
4. `STATUS.TX_LVL` shows `4'b1000`.
5. No data corruption or overwrite occurs.
6. When `md_tx_ready` is de asserted (set to 1 again), aligned data flow resumes.

3.3.4 End of Test Checks

1. Confirm `IRQ.TX_FIFO_FULL` is set.
2. Check `irq` pin is HIGH for 1 clock cycle.
3. Verify `STATUS.TX_LVL = 4'b1000`.
4. Resume `md_tx_ready = 1` and verify smooth continuation of TX.

3.4 TX FIFO EMPTY

To verify the behavior of the Aligner IP when the TX FIFO becomes empty. Ensure that `IRQ.TX_FIFO_EMPTY` is correctly asserted and behaves as a sticky interrupt. The design should resume transmission correctly once new data is available.

3.4.1 Initial Configuration Steps

1. Apply system reset.
2. Enable `IRQEN.TX_FIFO_EMPTY`.
3. Start with the RX FIFO partially filled.
4. Keep `md_rx_valid = 0` initially to prevent new data input.

3.4.2 Scenario Generation

1. Set `md_tx_ready = 1` to allow TX FIFO to drain.
2. Wait for TX FIFO to empty as aligned data is transmitted.
3. Monitor `IRQ.TX_FIFO_EMPTY` and `irq` pin status.
4. Observe `STATUS.TX_LVL` register for 0 level.
5. Optionally check that the sticky bit remains set even after the FIFO becomes non-empty again (until explicitly cleared).

3.4.3 Expected Behavior

1. `IRQ.TX_FIFO_EMPTY` is set once TX FIFO is drained.
2. `irq` pin is asserted for 1 clock cycle if IRQ is enabled.
3. `STATUS.TX_LVL = 4'b0000` (FIFO empty).
4. No data corruption or misalignment occurs.

3.4.4 End of Test Checks

1. Confirm `IRQ.TX_FIFO_EMPTY = 1`.
2. `irq` pin is HIGH for 1 clock cycle.
3. Confirm `STATUS.TX_LVL = 4'b0000`
4. Ensure sticky bit retains its state until software clears it.

3.5 CNT_DROP MAX DETECTION

To verify the behavior of the Aligner IP when the `CNT_DROP` counter reaches its maximum value (255). Ensure the counter saturates correctly, does not overflow, and triggers `IRQ.MAX_DROP` if enabled.

3.5.1 Initial Configuration Steps

1. Apply reset.
2. Enable `IRQEN.MAX_DROP` bit.
3. Set up the DUT with legal initial configurations (`CTRL.SIZE, CTRL.OFFSET`).
4. Ensure the RX path is active (`md_rx_ready = 1`).

3.5.2 Scenario Generation

1. Begin sending illegal MD transfers — e.g., invalid offset-size combinations (e.g., `SIZE=3, OFFSET=1`) to trigger drops.
2. Send at least 255 illegal transactions.
3. Optionally, interleave with legal transactions to verify no disruption.
4. Monitor `STATUS.CNT_DROP`, `irq` pin, and `IRQ.MAX_DROP` register bit.

3.5.3 Expected Behavior

1. `STATUS.CNT_DROP` increments with each illegal transfer.
2. When `STATUS.CNT_DROP` reaches 255, it saturates and does not overflow.
3. `IRQ.MAX_DROP` is set.
4. `irq` pin is asserted for 1 clock cycle.
5. Legal transactions continue to be processed correctly.
6. No data is transmitted for illegal MDs.

3.5.4 End of Test Checks

1. `STATUS.CNT_DROP = 8'hFF` (255), and stays in 255 even if more than 255 illegal packets have been sent to the DUT.
2. `IRQ.MAX_DROP = 1`.
3. `irq` pin = HIGH (if IRQ enabled).
4. `md_rx_err` pulses for each illegal transfer.
5. Legal data appears correctly on TX stream.

3.6 VERIFICATION OF TX_LVL AND RX_LVL

To ensure the STATUS.TX_LVL and STATUS.RX_LVL register fields correctly reflect the current levels of the TX and RX FIFOs during ongoing transactions, and that the appropriate interrupts are raised at boundary conditions.

3.6.1 Initial Configuration Steps

1. Apply system reset.
2. Enable FIFO-related interrupts:
    a. IRQEN.RX_FIFO_FULL
    b. IRQEN.TX_FIFO_FULL
    c. IRQEN.RX_FIFO_EMPTY
    d. IRQEN.TX_FIFO_EMPTY
3. Configure CTRL.SIZE and CTRL.OFFSET with legal values.

3.6.2 Scenario Generation

1. Begin sending a finite number of valid MD packets into the RX FIFO.
2. After each packet, read and log STATUS.RX_LVL and STATUS.TX_LVL
3. Allow partial draining by asserting md_tx_ready.
4. Repeat the process with varying SIZE values (1, 2, 4).
5. Trigger full and empty FIFO conditions to verify interrupts.

3.6.3 Expected Behavior

1. At any point in time:

```
total_bytes_in_fifos = (rx_lvl * size_reg) + (tx_lvl * size_reg);
```

2. STATUS.TX_LVL reflects number of packets, not bytes.
3. STATUS.RX_LVL updates with each RX transaction.
4. FIFO levels never exceed FIFO_DEPTH.
5. FIFO levels return to 0 after complete drain.
6. Correct interrupts are asserted at appropriate thresholds.
7. No loss or duplication of data.

3.6.4 End of Test Checks

1. STATUS.RX_LVL and STATUS.TX_LVL match the expected FIFO states at each step.
2. No FIFO overflow occurs.
3. IRQ lines for FIFO full/empty are correctly asserted/deasserted.
4. Data output matches input (alignment intact).
5. Simulation log confirms all levels and interrupts function as intended.

4 REGISTER INTERFACE VERIFICATION

4.1 CLR BIT FUNCTIONALITY

To verify the behavior of the CTRL.CLR bit, which is responsible for resetting the STATUS.CNT_DROP register used to count illegal MD transactions.

4.1.1 Initial Configuration Steps

1. Apply system reset.
2. Enable the IRQEN.MAX_DROP interrupt.
3. Prepare the environment to send illegal MD transactions that would increment STATUS.CNT_DROP.

4.1.2 Scenario Generation

1. Send a sequence of illegal MD transactions to increment the STATUS.CNT_DROP value to a known number (e.g., 10).
2. Write a 1 to CTRL.CLR and verify that STATUS.CNT_DROP resets to 0.
3. Continue sending more illegal MD transactions and verify that STATUS.CNT_DROP increments from 0 again.
4. Special Case:
    a. Allow STATUS.CNT_DROP to increment until it reaches 255.
    b. Write a 1 to CTRL.CLR and confirm the reset.
    c. Keep CTRL.CLR = 1 and send further illegal packets.
    d. Confirm that STATUS.CNT_DROP increments again (since CLR is edge-sensitive).
    e. Ensure that writing 0 to CTRL.CLR has no effect.
    f. Confirm that reading CTRL.CLR always returns 0 (write-only behavior).

4.1.3 Expected Behavior

1. STATUS.CNT_DROP resets to 0 immediately upon writing 1 to CTRL.CLR.
2. Writing 0 to CLR does not change the STATUS.CNT_DROP value.
3. If STATUS.CNT_DROP is 255, the IRQ.MAX_DROP interrupt is triggered and the IRQ pin is asserted.
4. Subsequent illegal transfers after reset continue incrementing STATUS.CNT_DROP properly.
5. CTRL.CLR read returns 0.

4.1.4 End of Test Checks

1. STATUS.CNT_DROP = 0 after writing 1 to CTRL.CLR.
2. IRQ.MAX_DROP asserted when STATUS.CNT_DROP = 255.
3. Illegal MD packets continue to increment STATUS.CNT_DROP even if CLR remains at 1.
4. Data integrity is not affected by CLR operation.

4.2 CHECK THE ACCESS TYPES OF THE REGISTER FIELDS

To verify the access type behavior (RO, WO, RW, W1C, Reserved) of various register fields by attempting valid and invalid operations and ensuring the DUT responds according to specification.

4.2.1 Initial Configuration Steps

1. Apply system reset.
2. No interrupts need to be enabled for this test.
3. Ensure register model is available to compare mirrored values with DUT.

4.2.2 Test Procedure

1. RW Fields (e.g., `CTRL.SIZE, CTRL.OFFSET, IRQEN`)
    a. Write known legal values.
    b. Read back the values.
    c. Compare read values to written values.
2. RO Fields (e.g., `STATUS.RX_LVL, STATUS.TX_LVL, STATUS.CNT_DROP`)
    a. Attempt to write non-zero values.
    b. Expect `pslverr = 1` indicating write failure.
    c. Read back to confirm values remain unchanged.
3. WO Field (`CTRL.CLR`)
    a. Read the field — should always return `0`.
    b. Write `1` — expect the `STATUS.CNT_DROP` register to reset to `0`.
    c. Write `0` — no effect on `STATUS.CNT_DROP`.
    d. Validate `STATUS.CNT_DROP` before and after to confirm behaviour.
4. W1C Fields (`IRQ` bits)
    a. Simulate IRQ conditions (e.g., trigger TX FIFO empty).
    b. Read the `IRQ` register — bits should be set.
    c. Write `0` to set bits — no effect; bits stay high.
    d. Write `1` — bits should clear.
    e. Read back to verify bits are cleared.
5. Reserved or Unused Bits
    a. Attempt writes (if addressable).
    b. Expect field to be read-only.
    c. Read back — should return `0`.

4.2.3 Expected Behavior

1. **RO Fields:** Reject writes; return actual status; `pslverr = 1` on write.
2. **RW Fields:** Accept write and reflect values correctly on read.
3. **WO Fields:** Return `0` on read; perform defined action on valid write (e.g., `CTRL.CLR`).
4. **W1C Fields:** Only clear on `1` write; writing `0` leaves them unchanged.
5. **Reserved Fields:** Not writeable; return 0.

4.2.4 End of Test Checks

1. All **RW fields** correctly reflect updated values.
2. All **RO field** writes are rejected with `pslverr = 1`, values unchanged.
3. **WO fields** read as `0`, write operations have expected side-effects.
4. **W1C fields** clear only on writing `1`.
5. No unintended corruption or side effects in register state.
6. Register model mirror values match DUT for all RW fields.

4.3 APB Error Conditions

To verify that illegal or unsupported APB register accesses are properly handled by the DUT by:

- Asserting `pslverr = 1` for invalid writes.
- Preventing updates to read-only or restricted fields.
- Accesses to unmapped (invalid) APB addresses.
- Ensuring register integrity and correct response signaling.

4.3.1 Initial Configuration Steps

1. System should be out of reset.
2. No interrupts need to be enabled for this test.
3. Valid register addresses include: `0x0000, 0x000C, 0x00F0, 0x00F4`.

4.3.2 Test Procedure

1. Attempt to write to read-only register:
    a. Write a non-zero value to `STATUS` (RO).
    b. Expect `pslverr = 1`, and no change to the `STATUS` register.
2. Attempt to write to W1C register with arbitrary value:
    a. Write `CTRL.SIZE = 0` (invalid size).
    b. Expect `pslverr = 1`, and no update to the `SIZE` field.
3. Write out-of-range values:
    a. Write illegal combinations to `SIZE` and `OFFSET`.
4. Write a valid configuration to confirm normal behavior:
    a. Write `CTRL.SIZE = 1`.
    b. Expect `pslverr = 0`, and the write should succeed.
5. Read from Invalid Address: Send an APB read to address `0x0004`.
6. Write to Invalid Address: Send an APB write to address `0x00A0`.
7. Randomized Invalid Access: Randomly generate APB transactions targeting unmapped addresses.
8. Try Both Read and Write: Perform both `pwrite = 0` and `pwrite = 1` operations to invalid locations.

4.3.3 Expected Results

1. All illegal access attempts should raise `pslverr = 1`.

2. No updates should occur in read-only or W1C fields on illegal writes.
3. For valid writes, `pslverr = 0` and values should be updated correctly.
4. `pready = 1`: Transaction completes.
5. `pslverr = 1`: Error correctly flagged.

4.3.4 End of Test Checks

1. `STATUS` and `IRQ` remain unchanged after illegal writes.
2. `CTRL.SIZE` and `CTRL.OFFSET` do not accept illegal values.
3. Unmapped Address access will generate `pslverr`.
4. Register integrity is preserved.
5. No side-effects or false interrupt triggers due to invalid access.

5 RESET AND LOW POWER CONDITIONS

5.1 EFFECT OF UNEXPECTED RESET ON FIFOS

To verify that a mid-stream reset correctly flushes both RX and TX FIFOs, resets relevant status and control registers, and does not lead to data corruption or undefined behavior upon resuming operation.

5.1.1 Initial Configuration

1. Set `CTRL.SIZE = 2`, `CTRL.OFFSET = 0` or any legal combination.

5.1.2 Test Procedure

1. Configure DUT using legal APB transactions.
2. Send a few RX data words (e.g., 4–5) to partially fill RX FIFO.
3. Let some TX responses propagate and fill the TX FIFO.
4. Assert reset mid-operation.
5. Hold reset for 5–10 cycles.
6. Deassert reset.
7. Read back STATUS.RX_LVL, STATUS.TX_LVL, and STATUS.CNT_DROP.
8. Reconfigure DUT using APB writes.
9. Resume RX and TX data flow.
10. Monitor output and internal state post-reset.

5.1.3 Expected Behavior

1. `STATUS.RX_LVL` and `STATUS.TX_LVL` reset to 0.
2. `STATUS.CNT_DROP` reset to 0.
3. All `IRQ` flags are cleared.
4. `CTRL.SIZE`, `CTRL.OFFSET`, and `CTRL.CLR` reset to default values.
5. `md_rx_ready` and `md_tx_valid` deasserted until valid reconfiguration.
6. No stale data is observed on TX output after reset.
7. DUT should resume normal functionality after reset and reconfiguration.

5.1.4 End-of-Test Checks

1. Confirm both FIFOs are empty (`STATUS.RX_LVL = 0, TX_LVL = 0`).
2. No TX output should appear unless valid new RX is sent.
3. `STATUS.CNT_DROP` is 0.
4. Reconfigurable registers hold default values after reset.
5. DUT accepts and processes new data correctly post-reset.

5.2 CLOCK GATING

To validate that internal gated clock functions correctly, ensuring that functional blocks stop toggling during gating, preserve their state, and resume cleanly without data loss or corruption. The DUT must not trigger false interrupts or alter internal states during clock gating.

5.2.1 Initial Configuration

1. Apply system reset.
2. Configure the `CTRL` register with valid values (e.g., `CTRL.SIZE = 2`, `CTRL.OFFSET = 0`).
3. Enable external gating enable signal.

5.2.2 Test Procedure

1. Apply reset and configure the DUT via APB interface.
2. Send a few valid RX transactions to load RX FIFO.
3. Observe TX activity as normal operation begins.
4. Assert the clock gating enable signal.
5. Hold clock gating for several cycles:
    a. TX logic should stall.
    b. No changes should occur in TX FIFO, RX FIFO, or outputs.
6. Deassert clock gating and allow DUT to resume.
7. Confirm all pending operations continue as expected.
8. Repeat gating during different phases (RX active, TX active) to ensure robustness.

5.2.3 Expected Behavior

1. During Gating:
    a. No toggling in TX/RX logic.
    b. `md_tx_valid` and `md_rx_ready` remain static.
    c. FIFO levels do not change.
    d. No change in internal registers or DUT state.
    e. No IRQs triggered.
2. After Gating:
    a. TX and RX resume correctly.
    b. Aligned data continues from correct point.
    c. No packet drops (`STATUS.CNT_DROP = 0`).
    d. Register values remain intact.

5.2.4 End-of-Test Checks

1. FIFO levels are unchanged during gated period.
2. No changes in register values or side effects observed.
3. `STATUS.CNT_DROP = 0` indicating no unexpected drops.
4. TX output resumes and aligned data flows after removing gate enable.
5. No unexpected interrupts are triggered.
6. Functional behavior remains intact after clock gating.

6 BASIC SANITY TESTS

6.1 SANITY CHECKS

To verify basic functional correctness of the DUT under normal configuration. This includes validating reset behavior, register access, and end-to-end data flow from RX to TX. These checks ensure the design is in a stable and known-good state before advanced verification begins.

6.1.1 Initial Configuration

1. Assert and deassert system reset.
2. Complete system with Default values.
3. Clock and interface signals are active.

6.1.2 Test Procedure

1. Reset Sanity Check**:**
    a. Apply reset (assert then deassert).
    b. Read all registers (`CTRL, STATUS, IRQEN, IRQ`, etc.) to confirm default values.
    c. Check FIFO status (`STATUS.RX_LVL, STATUS.TX_LVL = 0`), `STATUS.CNT_DROP = 0`.
2. APB Register Access Sanity:
    a. Write legal values to `CTRL` and `IRQEN` registers.
    b. Read back and verify mirror matches DUT.
    c. Ensure no `pslverr` during legal access.
3. Data Path Sanity Check:
    a. Configure `CTRL.SIZE = 1, CTRL.OFFSET = 0` or any legal combination.
    b. Generate a burst of 4 RX data transfers with toggling pattern.
    c. Confirm that TX interface generates aligned output.
    d. Monitor FIFO levels, verify expected rise and fall.

6.1.3 Expected Behavior

1. `md_rx_ready`: Asserted unless RX FIFO is full.
2. `md_tx_valid`: Asserted when aligned TX data is available.
3. `md_tx_data`: Matches aligned RX pattern.
4. `STATUS.RX_LVL / TX_LVL`: Increase then decrease during traffic.
5. `IRQ`: No interrupts triggered (`IRQEN = 0`).
6. `STATUS.CNT_DROP = 0.`

6.1.4 End of Test Checks

1. Data integrity confirmed: TX output matches aligned RX input.
2. FIFO levels return to zero (empty).
3. No error signals (e.g., `md_tx_err`) asserted.
4. All registers reflect correct post-operation values.

5. DUT returns to idle/stable state with no hanging valid signals.

6.2 Dynamic Configuration Change Tests

Verify DUT's ability to handle legal mid-transfer updates to control registers without corrupting ongoing transfers.

What happens to the already aligned data, how many clock cycles are stored before the setting has changed? What my understanding is, the previously aligned data comes out and the data which is incoming will start aligning as per the new config.

6.2.1 Initial Configuration

1. Apply system reset.
2. Set `CTRL.SIZE = 1`, `CTRL.OFFSET = 0` or any other legal combination.
3. Prepare a valid RX data stream.

6.2.2 Test Procedure

1. Begin transmission with the above config.
2. Midway, update `CTRL.SIZE = 2`, `CTRL.OFFSET = 2` or any other legal combination other than the above chosen combination.
3. Continue RX data stream.
4. Monitor TX output and alignment.
5. Observe FIFO levels and error flags.

6.2.3 Expected Behavior

1. No `md_rx_err`.
2. Data before config switch is aligned per old setting.
3. Data after switch aligns with new setting.
4. `STATUS.TX_LVL` reflects smooth operation without drop spikes.
5. `STATUS.CNT_DROP` remains unchanged.

6.2.4 End of Test Checks

1. Confirm split in alignment behavior before/after switch.
2. Ensure TX output reflects correct byte alignment from lane 2 post update.
3. No IRQ or errors triggered unexpectedly.

7 ASSERTION BASED VERIFICATION

7.1 APB PROTOCOL VIOLATION CHECKS

7.1.1 Stable Address During Transfer

The objective is to ensure that the APB address (`paddr`) remains stable during an active transfer when `psel` and `penable` are asserted, avoiding any protocol violations. The system should be reset and the APB master initialized with a valid address before starting the test. During the test, an APB transfer is initiated by asserting `psel` and `penable`, and while `pready` is low, the `paddr` signal is deliberately changed to check if the DUT detects the violation. The expected behavior is that `paddr` must remain stable until `pready` is asserted, and any change in the address during this period should trigger an assertion failure.

7.1.2 Too Many Wait States

This check verifies that the APB slave does not hold the master in a wait state longer than 5 clock cycles after `penable` is asserted. The test starts by driving valid APB transfers with `psel` and `penable` both asserted, while forcing `pready` to remain low beyond 5 cycles intentionally. The DUT is expected to assert `pready` within 5 cycles; failure to do so indicates a protocol violation. An error flag or assertion should trigger if the wait state exceeds this limit, ensuring timely handshake completion.

7.1.3 PENABLE Must Not Be Asserted Without PSEL

This check ensures that `penable` is only asserted when `psel` is also high, as per APB protocol rules. A stimulus is applied that incorrectly asserts `penable` while keeping `psel` low to test for violations. The DUT should flag this as a protocol error and ignore the transfer attempt. Assertions monitoring this condition should trigger to confirm compliance. This ensures valid transaction initiation behavior on the APB bus.

7.1.4 PENABLE Timing Check

This check validates that `penable` is asserted only in the second cycle of an APB transaction, following a valid assertion of `psel`. A valid APB transfer begins with `psel = 1` and `penable = 0`, and then `penable` is set high in the next cycle. Any early or delayed assertion of `penable` indicates protocol violation. Assertions should monitor this sequence to ensure correct transaction timing and prevent unintended data operations.

7.1.5 PENABLE Deassertion Check

This check ensures that `penable` is deasserted immediately after `pready` is asserted in a valid APB transaction. Once the slave responds with `pready = 1`, the master must drop `penable` in the following cycle. Holding `penable` high beyond this violates the APB protocol and can result in incorrect transaction interpretation. An assertion should be used to flag if `penable` remains high more than one cycle after `pready`. It is expected that the assertion should fail.

7.1.6 Master Signals Stability Check

This check ensures that all APB master control signals (`paddr, pwrite, psel, penable`) remain stable throughout the transaction. From the assertion of `psel` until the end of the transfer (i.e., after `pready` goes high), none of these signals should toggle. Any change during this window indicates a protocol violation and may result in unpredictable behavior from the slave. Assertions should verify stability using `$stable` for each signal.

7.1.7 No Unknown Values on APB Signals

This check ensures that no APB signal (`psel, penable, paddr, pwrite, pwdata, pready, prdata`) should ever hold an unknown (`X`) value during a transaction. The presence of unknown values on these lines can lead to undefined behavior or silent functional mismatches. Verification must include assertions to flag any occurrence of `X` on these signals during simulation.

7.1.8 Bounded Transfer Duration (Protocol Violated)

This check ensures that any APB transfer completes within a bounded number of clock cycles. A valid transfer starts with `psel = 1`, followed by `penable = 1`, and must complete once `pready = 1`. If the transfer remains incomplete (i.e., `pready = 0`) beyond a defined maximum duration (e.g., 10 cycles), it indicates a protocol violation. This condition must be flagged to detect stuck transfers or faulty handshakes.

7.2 APB Protocol Compliant Checks

7.2.1 PENABLE Timing Compliance

This test validates that the PENABLE signal is asserted only in the second cycle of an APB transaction after PSEL is high and remains asserted until the PREADY signal is received. The DUT should not assert PENABLE in the same cycle as PSEL. This timing behavior ensures the proper handshake phase is followed as per the APB protocol specification. The test passes if PENABLE is asserted only after one cycle delay from PSEL and remains valid until PREADY.

7.2.2 PENABLE Deassertion on Transfer Completion

This test ensures that the PENABLE signal is deasserted immediately in the cycle after PREADY is asserted, marking the end of a valid APB data transfer. Proper deassertion of PENABLE after transfer completion is crucial for avoiding unintended transactions. The test passes if the DUT cleanly drops PENABLE in the very next cycle after PREADY goes high, indicating protocol compliance.

7.2.3 Stable Master Signals During Transfer

This test verifies that all APB master signals — PADDR, PWRITE, PSEL, and PWDATA — remain stable once PENABLE is asserted and until PREADY is deasserted. Any change in these signals during this window would constitute a protocol violation. The test ensures that the master maintains signal integrity and adheres to the APB protocol.

7.2.4 No Undefined Values on APB Signals

This check ensures that no APB signals (PADDR, PSEL, PENABLE, PWRITE, PWDATA, PREADY) assume undefined or unknown values (such as 'X' or 'Z') during any phase of the APB transaction. Maintaining defined logic levels is critical to avoid unpredictable behavior and protocol violations in the DUT. The assertion monitors signal stability throughout all cycles.

7.2.5 Bounded Transfer Duration (Protocol Followed)

This check verifies that each APB transfer completes within a defined maximum number of clock cycles. The slave must assert `PREADY` to complete the transfer in a timely manner, preventing the master from being stalled indefinitely. This ensures efficient bus usage and prevents protocol violations due to prolonged transfers.

7.3 MEMORY DATA PROTOCOL (MDP) COMPLIANCE CHECKS

This section ensures the DUT adheres strictly to the Memory Data interface protocol by validating all signal interactions and timing requirements. The assertions confirm that the data, control signals, and handshakes operate within the defined legal constraints to guarantee reliable and predictable data transfer behaviour.

7.3.1 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.

7.3.2 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.

7.3.3 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.

7.3.4 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.

7.3.5 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.

7.3.6 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).

7.3.7 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).

7.3.8 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).

7.3.9 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.

7.3.10 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.

7.3.11 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.

7.3.12 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.

7.3.13 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.

7.3.14 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.

7.3.15 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.

7.3.16 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.

7.4 MEMORY DATA PROTOCOL (MDP) VIOLATION CHECKS

7.4.1 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.

7.4.2 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.

7.4.3 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.

7.4.4 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.

7.4.5 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.

7.4.6 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.

7.4.7 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.

7.4.8 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.

7.4.9 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.

7.4.10 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.

7.4.11 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.

7.4.12 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.

7.4.13 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.

7.4.14 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.

7.4.15 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.

7.4.16 MD transfer cannot 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.