

APB PROTOCOL VERIFICATION

Contents

CHAPTER 1 – APB OVERVIEW	2
1.1 APB PROTOCOL	2
1.2 KEY FEATURES	2
CHAPTER 2 – DESIGN OVERVIEW	3
2.1 Design specification	3
CHAPTER 3 – VERIFICATION ARCHITECTURE	4
3.1 Flow Chart of UVM Components	5
3.1.1 apb_top	5
3.1.2 apb_test	5
3.1.3 apb_env	5
3.1.4 apb_interface	5
3.1.5 apb_active_agent	5
3.1.6 apb_passive_agent	6
3.1.7 apb_sequence_item	6
3.1.8 apb_sequence	6
3.1.9 apb_sequencer	6
3.1.10 apb_driver	6
3.1.11 apb_active_monitor	6
3.1.12 apb_passive_monitor	6
3.1.13 apb_scoreboard	7
3.1.14 apb_coverage	7
3.2 Pseudo Codes	7
CHAPTER 4 – TEST PLAN	9
4.1 Test Cases	9

CHAPTER 1 – APB OVERVIEW

1.1 APB PROTOCOL

The ARM AMBA (Advanced Microcontroller Bus Architecture) protocol family includes the Advanced Peripheral Bus (APB). APB offers a straightforward and effective interface for controlling slower peripherals that don't need high data transfer rates. It is intended for interfacing low-bandwidth peripherals in a system-on-chip (SoC).

1.2 KEY FEATURES

- **Low Power Consumption:** APB is optimized for low power, making it suitable for battery-powered devices and low-performance applications.
- **Simple Protocol:** The protocol has a straightforward handshake mechanism, minimizing the complexity of integration for peripheral devices.
- **Single-Transaction Interface:** APB supports single transactions, which simplifies control and reduces overhead for less demanding peripherals.
- **Clock Gating Support:** APB allows for clock gating, enabling power savings by shutting off the clock to inactive peripherals.
- **Low Latency:** APB is designed for low-latency operations, making it effective for peripherals that require quick responses.
- **Wide Compatibility:** Being a part of the AMBA protocol suite, APB is widely supported in various ARM-based systems, facilitating interoperability.
- **No Burst Transfers:** Unlike other buses, APB does not support burst transfers, focusing on single accesses for simplicity.
- **Easier Implementation:** The simplicity of the protocol allows for easier design and implementation of peripheral devices.

CHAPTER 2 – DESIGN OVERVIEW

2.1 Design specification

The design consists of a single APB master controlled by external signals, communicating with two connected slaves. The master selects one slave at a time based on the least significant bit of the address. The APB is enabled only when the transfer signal is high; otherwise, it remains disabled.

1. Parallel bus operation. All the data will be captured at rising edge clock.
2. Two slave design based on 9th bit of apb_write_address bit it will elect the slave1 and slave2.
3. Signal priority: 1.PRESET (active low) 2. PSEL (active high) 3. PENABLE (active high) 4. PREADY (active high) 5. PWRITE
4. Data width 8 bit and address width 9 bit.
5. PWRITE=1 indicates write PWDATA to slave. PWRITE=0 indicates read PRDATA from slave.
6. Start of data transmission is indicated when PENABLE changes from low to high. End of transmission is indicated by PREADY changes from high to low.

Top Module Name: apb_protocol.v

APB Interface Block Diagram:

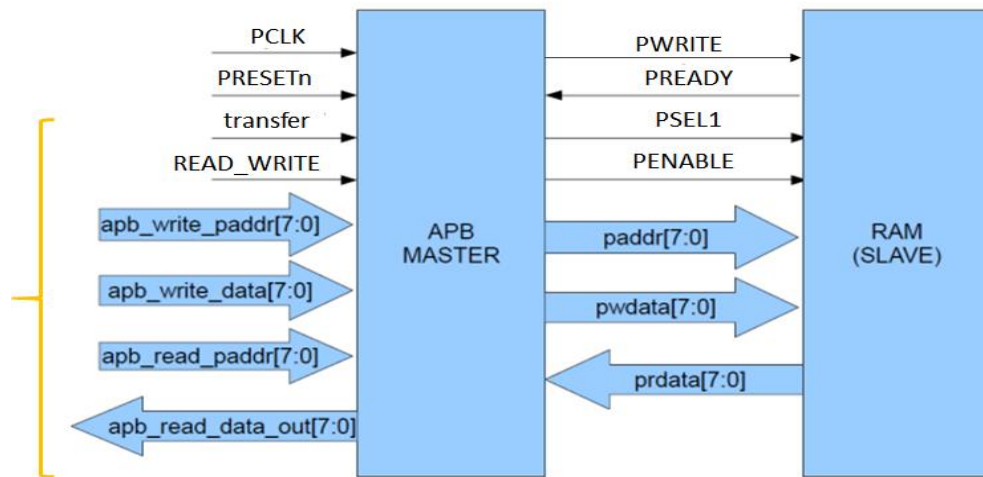


Figure 2.1 Block Diagram

CHAPTER 3 – VERIFICATION ARCHITECTURE

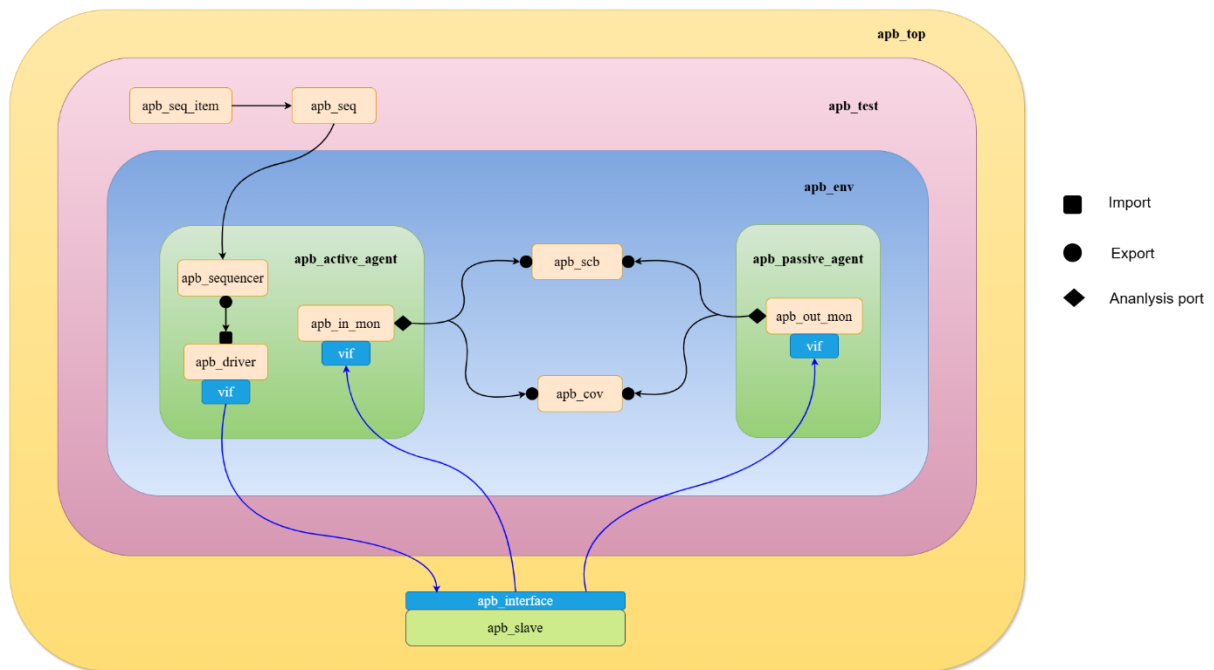


Figure 3.1 Verification Architecture

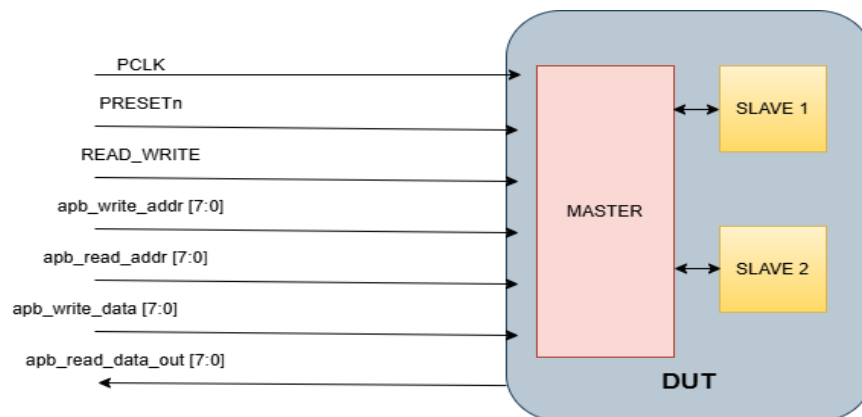


Figure 3.2 Design Specification Architecture

3.1 Flow Chart of UVM Components

3.1.1 apb_top

- A static container with instances of DUT and interfaces.
- Connects interface instance to DUT signals in the testbench top.
- Generates clock and applies an initial reset to the DUT, which then passes to the interface handle.
- Stores the interface in `uvm_config_db` using `set` and retrieves it with `get`.
- UVM testbench top triggers tests using `run_test()`.

3.1.2 apb_test

- Configures components (environment, agents, sequences) to set active/passive components and pass down configurations via `uvm_config_db`.
- In `build_phase()`, the environment is created, which in turn creates other components like agents, monitors, drivers, and scoreboards.
- `run_phase()` executes the test, triggering sequences that generate stimuli for the DUT, while the driver drives these values and the monitor observes DUT responses.
- Testbench components verify DUT behavior during this phase.
- When the stimulus completes, objections are dropped, signaling simulation end.
- Final verification checks conclude with a pass/fail report.

3.1.3 apb_env

- `apb_env` class extends `uvm_env` and registers with the factory.
- Contains two agents (active and passive), a scoreboard, and a coverage collector.
 - The active agent includes a sequencer, driver, and active monitor.
 - The passive agent includes only a passive monitor.
- Declares handles for active and passive agents, scoreboard, and coverage collector.
- Instantiates components in `build_phase()` and connects components via the analysis port in `connect_phase()`.

3.1.4 apb_interface

- Connects DUT to the testbench.
- Declares input and output signals as logic.
- Uses clocking blocks to specify signal sampling/driving relative to a clock, with separate blocks for driver and monitor.
- Defines modports for driver and monitor.

3.1.5 apb_active_agent

- `apb_active_agent` class extends `uvm_agent` and registers with the factory.
- Declares handles for sequencer, driver, and input monitor.
- In `build_phase()`, if the agent is active (`UVM_ACTIVE`), it creates instances of the sequencer, driver, and input monitor.
- In `connect_phase()`, connects the `seq_item_port` of the driver to the `seq_item_export` of the sequencer, allowing transactions to flow from sequencer to driver and then to the DUT.

3.1.6 apb_passive_agent

- `apb_passive_agent` class extends `uvm_agent` and registers with the factory.
- Contains only a monitor for observing and capturing DUT output signals.
- Declares a handle for the output monitor and creates it in `build_phase()`.

3.1.7 apb_sequence_item

- Extends `uvm_sequence_item`, inheriting properties for UVM sequences.
- Registers the class with the UVM Factory.
- Declares inputs as randomizable and output as non-randomizable members.

3.1.8 apb_sequence

- `apb_sequence` class extends `uvm_sequence`.
- Registers with the UVM factory.
- Implements a constructor, then creates child sequences extending the base sequence.
- Child sequences instantiate sequence item handles in `body()` task, using `uvm_do()` for inline constraint-based execution.

3.1.9 apb_sequencer

- Connects sequence to driver by passing transactions or sequence items.
- `apb_sequencer` extends `uvm_sequencer` with request (REQ) and response (RSP) parameters.
- Registers with the factory and declares a constructor.

3.1.10 apb_driver

- Extends the parameterized `uvm_driver` base class and registers with the factory.
- Creates a virtual interface to drive DUT signals.
- Retrieves interface handle from `uvm_config_db` in `build_phase()`.
- In `run_phase`, waits for transactions from sequencer using `seq_item_port.get_next_item(req)` and signals completion with `seq_item_port.item_done()`.

3.1.11 apb_active_monitor

- Extends `uvm_monitor` and registers with the factory.
- Declares a virtual interface to connect with APB signals.
- Creates a sequence item handle for sending data to the analysis port.
- Implements `build_phase()` for setup, `connect_phase()` for connections, and `run_phase()` to monitor signals in a loop, generating sequence items on clock edges.

3.1.12 apb_passive_monitor

- Extends `uvm_monitor` and registers with the factory.
- Declares a virtual interface for capturing DUT output signals.
- Sets up an analysis port for sending captured transactions to the scoreboard or coverage collector.

- Retrieves the virtual interface in `build_phase()`.
- `run_phase()` captures output signals, stores them in transaction objects, and sends them to an analysis port.

3.1.13 apb_scoreboard

- Extends `uvm_scoreboard` and registers with the factory.
- Declares an analysis export to receive transactions from the monitor.
- Implements `new()` constructor for initialization.
- `build_phase()` creates a TLM analysis export instance.
- `write()` receives transactions from the monitor.
- `run_phase()` continuously checks DUT functionality and reports mismatches between actual and expected values.

3.1.14 apb_coverage

- Extends `uvm_subscriber`.
- Defines a coverage group with coverpoints and cross-coverage for tracking signal behaviors and interactions.
- Implements a constructor for initializing the coverage group and registers with the UVM factory.

3.2 Pseudo Codes

INTERFACE

- The interface (`apb_interface`) connects the DUT with testbench components.
- All input and output signals are declared using `logic`.
- Two **clocking blocks** are defined: `cb_drv` for the driver and `cb_mon` for the monitor.
- **Modports** (`mp_drv` and `mp_mon`) are used to control signal direction for the driver and monitor.
- Clocking blocks are instantiated inside these modports to manage timing and signal direction.

SEQUENCE ITEM

- Class `apb_seq_item` extends `uvm_sequence_item`.
- All inputs and outputs are declared as class variables.
- Inputs are randomized using the `rand` keyword.
- The class is registered with the UVM factory and includes field macros for each variable.
- A constructor (`new`) is defined for object initialization.

SEQUENCER

- `apb_sequencer` is derived from `uvm_sequencer`.
- Its purpose is to coordinate between sequences and the driver.
- The class is registered with the UVM factory and includes a constructor.

DRIVER

- The `apb_driver` class extends `uvm_driver`.
- It is registered with the factory.
- Contains handles for the virtual interface (`vif`) and the sequence item (`pkt`).
- In the `build_phase`, the driver gets the virtual interface through `uvm_config_db` and creates the sequence item object.
- The `run_phase` contains handshake logic with the sequencer.
- A separate task (`drive`) handles actual signal driving to the DUT based on valid transaction data.

INPUT MONITOR

- `apb_in_mon` class extends `uvm_monitor`.
- It monitors and captures input signals sent to the DUT.
- Contains a virtual interface handle and an analysis port (`in_mon2sb_cov`) for communication with scoreboard and coverage.
- A handle for the sequence item is also declared.
- In `build_phase`, it fetches the interface via `uvm_config_db`, creates the sequence item, and initializes the analysis port.
- In `run_phase`, inputs from the DUT are sampled and passed through the analysis port.

OUTPUT MONITOR

- `apb_out_mon` class extends `uvm_monitor` and observes DUT outputs.
- Similar to the input monitor, it uses a virtual interface and an analysis port (`out_mon2sb_cov`) to send sampled data.
- Sequence item handle is declared, initialized in `build_phase`.
- In `run_phase`, DUT outputs are captured and forwarded via the analysis port.

ACTIVE AGENT

- `apb_active_agent` extends `uvm_agent`.
- It generates and drives input stimulus to the DUT.
- Contains handles for sequencer (`seqr`), driver (`drv`), and input monitor (`in_mon`).
- Constructor initializes the agent.
- In `build_phase`, it checks if the agent is active using `get_is_active()` and creates all required components.
- In `connect_phase`, it connects the sequencer and driver via `seq_item_port`.

PASSIVE AGENT

- `apb_passive_agent` extends `uvm_agent`.
- It passively observes DUT outputs without driving anything.
- Contains a handle to the output monitor (`out_mon`), which is created in the `build_phase`.

ENVIRONMENT

- The `apb_env` class extends `uvm_env`.
- It includes handles to the active agent (`a_agent`), passive agent (`p_agent`), coverage model (`cov`), and scoreboard (`sb`).
- Components are instantiated in the `build_phase`.
- Connections between components are made in the `connect_phase`.

TOP MODULE

- The `apb_top` module is the root of the simulation hierarchy.
- It imports `uvm_pkg` and includes `uvm_macros`.
- Declares signals like `pc1k` and `presetn`.
- Generates the clock and drives the reset signal.
- Instantiates the DUT and the `apb_interface`.

CHAPTER 4 – TEST PLAN

4.1 Test Cases

Sl No.	Features	Sub Feature	Description
1	Write operation	Simple write depending on READ_WRITE=0 signal	Write data to address allocated
2	Read operation	Simple read depending on READ_WRITE =1 signal	Read from address given
3	Reset behavior	Active low reset	Assert and deassert reset
4	Write & Read op	Back-to-back write & read	Write data to address, then read from that address
5	Slave selection	It is based on 9th bit of address	If apb_write_paddr[8] = 0, select Slave1; if apb_write_paddr[8] = 1, select Slave2
6	continuous read writes	Continuous Writes	Multiple back-to-back valid writes
		Continuous Reads	Multiple back-to-back valid reads
7	continuous writes to the same address.	continuous writes to the same address and read from the same address	continuous writes to the same address and read from the same address, check if the data is overwritten and in which data is available