

**Table of Contents**

[CHAPTER–1 DESIGN OVERVIEW 3](#_Toc25295)

[1.1 INTRODUCTION 4](#_Toc32525)

[1.2 FEATURES OF THE APB 4](#_Toc6226)

[1.3 SIGNAL DESCRIPTION OF APB 4](#_Toc14649)

[1.3.1 BLOCK DIAGRAM 5](#_Toc11344)

[1.3.2 PIN DESCRIPTION 5](#_Toc15828)

[1.4 STATE DIAGRAM 7](#_Toc17502)

[1.4.1 IDLE STATE: 8](#_Toc19754)

[1.4.2. SETUP STATE: 8](#_Toc13435)

[1.4.3. ACCESS STATE: 8](#_Toc13465)

[CHAPTER–2 ARCHITECTURE 9](#_Toc8318)

[2.1 STANDARD ARCHITECTURE 9](#_Toc11461)

[2.2 APB ARCHITECTURE 10](#_Toc6650)

# CHAPTER–1 DESIGN OVERVIEW

## 1.1 INTRODUCTION

The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification used for connecting and managing functional blocks within system-on-a-chip (SoC) designs. It provides a standardized framework for high-performance communications in 16-bit and 32-bit microcontrollers, signal processors, and complex peripheral devices. The AMBA specification is built around a Master-Slave protocol, enabling efficient communication and control between different components on the chip.

## 1.2 FEATURES OF THE APB

* Low Power Consumption: APB uses very little power, making it perfect for battery-powered devices and simple applications.
* Simple Protocol: It has an easy communication method, making it simple to connect to other devices.
* Single-Transaction Interface: APB processes one task at a time, which simplifies control for basic devices.
* Wide Compatibility: Since it's part of the AMBA system, APB works well in many ARM-based devices.
* No Burst Transfers: APB focuses on single transfers instead of handling multiple tasks at once, keeping things straightforward.
* Easier Implementation: Its simple design makes it easier to build and connect peripheral devices.

## 1.3 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 paddress. 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\_paddress 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.

#### 1.3.1 BLOCK DIAGRAM

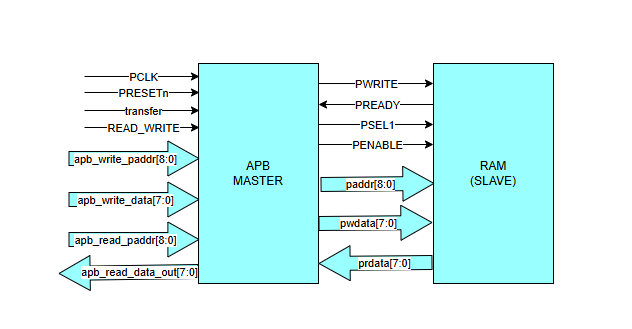


Figure 1.1 Block diagram

## 1.4 SIGNAL DESCRIPTION OF APB

#### 1.4.2 PIN DESCRIPTION

|  |  |  |
| --- | --- | --- |
| SIGNALS | DESCRIPTION | WIDTH |
| **Transfer**  Transfer signal | APB enable signal. If high APB is activated else APB is disabled | 1 |
| **PCLK** Bus clock | The rising edge of PCLK is used to time all transfers on the APB. | 1 |
| **PRESETn** APB reset | The APB bus reset signal is active LOW and this signal will normally be connected directly to the system bus reset signal. | 1 |
| **PADDR** APB address bus | This is the APB address bus, which may be up to 32-bits wide and is driven by the peripheral bus bridge unit. | 9 |
| **PSEL1** APB select | This signal indicates that the slave device is selected and a data transfer is required. | 1 |
| **PENABLE** APB enable | The enable signal is used to indicate the second cycle of an APB transfer. The rising edge of PENABLE occurs in the middle of the APB transfer. | 1 |
| **PWRITE** APB transfer direction | When HIGH this signal indicates an APB write access and when LOW a read access. | 1 |
| **PREADY**  APB ready | This is an input from Slave. It is used to enter the access state. | 1 |
| **PSLVERR**  APB slave error | This indicates a transfer failure by the slave. | 1 |
| **PRDATA** APB read data bus | The read data bus is driven by the selected slave during read cycles (when PWRITE is LOW). The read data bus can be up to 32-bits wide. | 8 |
| **PWDATA** APB write data bus | The write data bus is driven by the peripheral bus bridge unit during write cycles (when PWRITE is HIGH). The write data bus can be up to 32-bits wide. | 8 |

## 1.5 STATE DIAGRAM

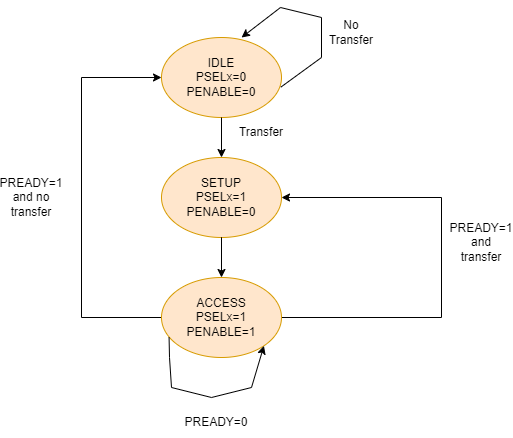


Figure 1.2 State diagram

Transitions:

* IDLE → SETUP: Happens when a transfer is initiated (PREADY = 1 and transfer request occurs).
* SETUP → ACCESS: Happens when PREADY = 0, meaning the system is still preparing to transfer.
* ACCESS → IDLE: Happens when PREADY = 1 and the transfer completes.
* IDLE (No Transfer): The system remains idle if no transfer is initiated (PREADY = 1 and no transfer).

### 1.5.1 IDLE STATE:

* PSELx = 0, PENABLE = 0
* This is the default or inactive state. No transfer is initiated here.
* The system will remain in the IDLE state if PREADY = 1 and no transfer is required.
* Transition occurs to the SETUP state if a transfer request is initiated (trigger not explicitly shown but implied by the transition arrow).

### 1.5.2. SETUP STATE:

* PSELx = 1, PENABLE = 0
* In this state, the peripheral select signal (PSELx) is asserted, indicating the target peripheral is selected for communication.
* The system is preparing for the actual data transfer.
* Transition to the ACCESS state occurs when PREADY = 0, meaning the bus is not yet ready to complete the transfer.

### 1.5.3. ACCESS STATE:

* PSELx = 1, PENABLE = 1
* This state represents the active transfer phase, where both PSELx and PENABLE are asserted.
* The data transfer occurs during this state.
* If PREADY = 1, indicating the bus is ready, and the transfer completes, the system transitions back to the IDLE state.
* If PREADY = 0, the system stays in the ACCESS state, waiting for the bus to be ready.

# CHAPTER–2 ARCHITECTURE

## 2.1 STANDARD ARCHITECTURE

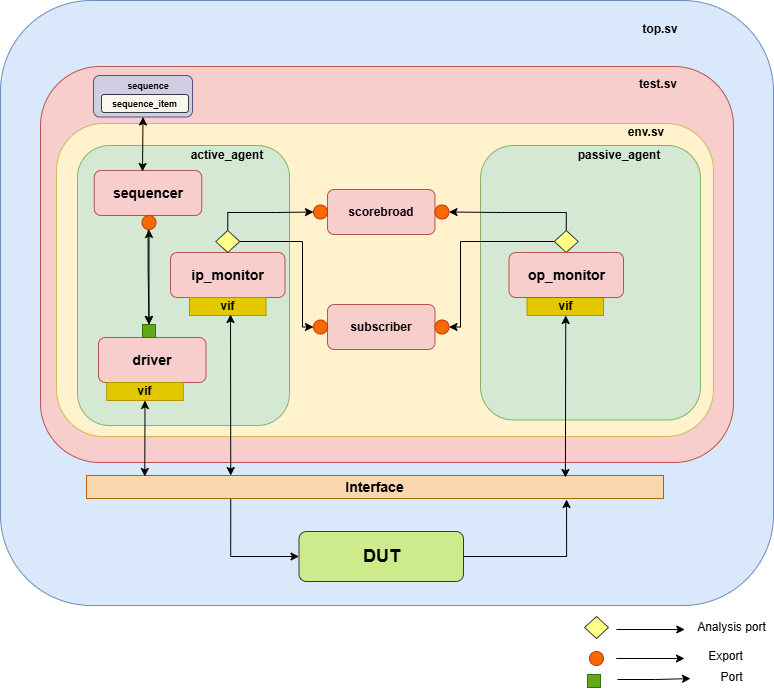


Figure 1.3 Standard Architecture

## 2.2 APB ARCHITECTURE

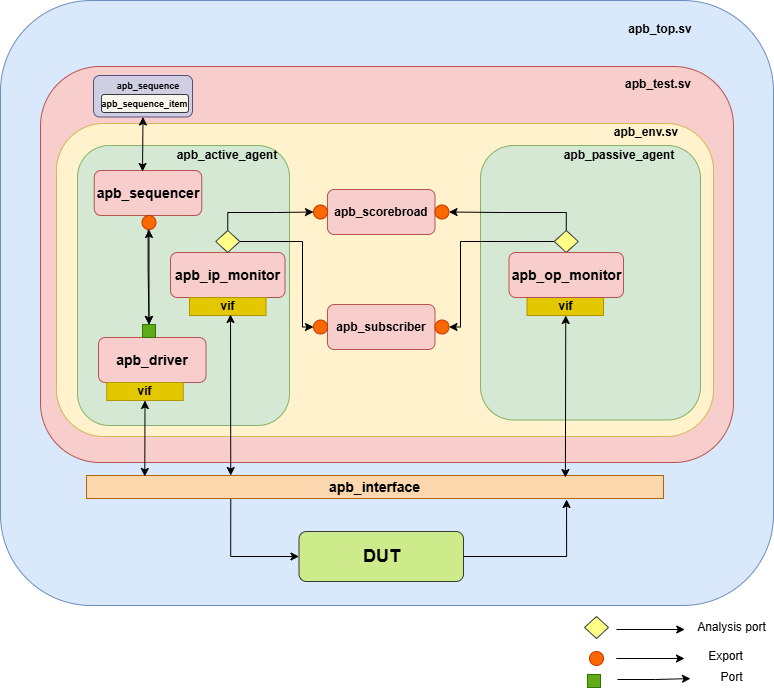


Figure 1.4 APB Architecture

## 2.3 APB DESIGN SPECIFICATION ARCHITECTURE

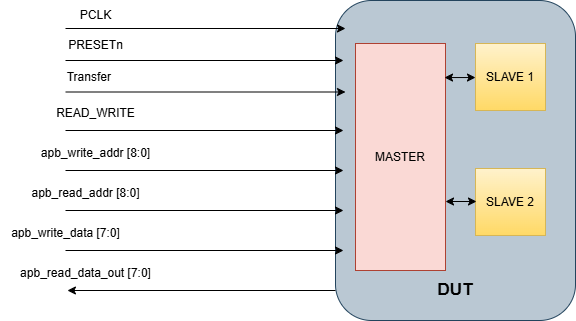
****

Figure 1.5 APB Design Specification Architecture

2.4 Testbench Components:

2.4.1 Interface(apb\_interface.sv)

• Interface name: apb\_inf

• Signal Declaration

• Clocking blocks for apb\_driver, apb\_monitor

• Modports for apb\_driver, apb\_monitor

2.4.2 Sequence\_item(apb\_seq\_item.sv):

• Class name: apb\_seq\_item

• Derived from uvm\_sequence\_item

• Input variables are declared as rand and output variables are as non-rand

• Consists of constraints

2.4.3 Sequence(apb\_sequence.sv):

• Class name: apb\_sequence

• Derived from uvm\_sequence

• Consists of 6 methods

• They are as follows:

(f) Create\_item

(f) Wait\_for\_grant()

(f) Randomize()

(f) Send\_request()

(f) Wait\_for\_item\_done()

(f) Get\_response()

2.4.4 Sequencer(apb\_sequencer.sv):

• Class name: apb\_sequencer

• Derived from uvm\_sequencer

• Use of uvm macros for factory registration

• Use of class constructor

2.4.5 Driver(apb\_driver.sv):

• Class name: apb\_driver

• Virtual interface handle

• A function build\_phase():

 `uvm\_config\_db is used to bring back a configuration setting from the UVM

configuration database

• A task run\_phase():

 Methods to retrieve the next sequence item from the sequencer:

 seq\_item\_port to connect driver to the sequencer

 get\_next\_item() to fetch the next sequence item from the sequencer

queue

 drive() task is called

 item\_done() method indicates to the sequencer that the current sequence

item processing has been completed

• Virtual task drive()

 drive() task is used to drive the values of the sequence item onto the DUT through

the interface

2.4.6 Input Monitor (apb\_input\_monitor.sv):

• Class name: apb\_input\_monitor

• User-defined monitor class extended from uvm\_monitor and register it in the uvm factory

• Analysis port to broadcast the sequence items or transactions

• Virtual interface handle to retrieve actual interface handle

• Standard class constructor to create an instance for sequence\_item

• Use build\_phase, and retrieve the interface handle from the configuration table

• Create transactions by implementing run\_phase to a sample DUT interface using a virtual

interface handle

• The write() method sends transactions to the collector component

• In apb\_op\_mon, DUT signals are captured

2.4.7 Output Monitor (apb\_output\_monitor.sv):

• Class name: apb\_output\_monitor

• User-defined monitor class extended from uvm\_monitor and register it in the uvm factory

• Analysis port to broadcast the sequence items or transactions

• Virtual interface handle to retrieve actual interface handle

• Standard class constructor to create an instance for sequence\_item

• Use build\_phase, and retrieve the interface handle from the configuration table

• Create transactions by implementing run\_phase to a sample DUT interface using a virtual

interface handle

• The write() method sends transactions to the collector component

• In apb\_ip\_mon, input signals are captured

2.4.8 Scoreboard(apb\_scoreboard.sv):

• Class name: apb\_scoreboard

• Derived from uvm\_scoreboard

• Establish a fifo to hold the value apb\_seq\_item

• The analysis port "item\_collected\_export" is where the sequence items are transferred

to the analysis component of the apb\_scoreboard.

• function build\_phase ()

 creates an instance of the uvm\_analysis\_imp class

• Virtual function write ()

 function used to handle the incoming mem\_sequence\_item objects ( transactions)

• Virtual task run\_phase ()

2.4.9 Coverage(apb\_coverage.sv):

• Class name:apb\_cov

• This class extended from uvm\_subscriber

• Define a coverage group that contains coverpoints and cross coverage

• Coverage group constructor is created to initialize the coverage group

• write() method to collect data

• Register the apb\_coverage class with the UVM factory

2.4.10 Agent\_Active(apb\_agent\_active.sv):

• Class name:apb\_agent\_active

• Define apb\_agt\_active class

 Extend apb\_agt\_active from uvm\_agent

 Usage of macros to register the apb\_agent\_active with uvmfactory

 Creating a class constructor

• Declaration of an handle for apb\_sequencer, apb\_monitor and apb\_driver

• Declartion of a flag “is\_active” to determine whether the agent is active or passive

• Implement the build phase()

 Instantiate the monitor and sequencer

 The build\_phase of the agent class checks this flag and decides whether to

instantiate the driver

• Implement the connect phase()

 Connect sequencer to driver

 Connect monitor’s analysis port

2.4.11 Agent\_Passive(apb\_agent\_passive.sv):

• Class name:apb\_agent\_passive

• Define apb\_agt\_passive class

 Extend apb\_agt\_passive from uvm\_agent

 Usage of macros to register the apb\_agent\_passive with uvm factory

 Creating a class constructor

• Declaration of an handle for apb\_sequencer, apb\_monitor and apb\_driver

• Declaration of a flag “is\_active” to determine whether the agent is active or passive

• Implement the build phase()

 Instantiate the monitor and sequencer

 The build\_phase of the agent class checks this flag and decides whether to

instantiate the driver

• Implement the connect phase()

 Connect monitor’s analysis port

2.4.12 Environment(apb\_env.sv):

• Class name:apb\_env

• Define alu\_env class

 Extend apb\_env from uvm\_environment

 Usage of macros to register the apb\_env with uvm factory

 Create a class constructor

• Declare and Instantiate agents

 Declaration of a handle for active agent (apb\_agent\_active) and passive agent

(apb\_agent\_passive)

• Declare and Instantiate scoreboard

• Declare and Instantiate coverage

• Implement the build phase ()

 Creating an instance of all the components

• Implement the connect phase()

 Connecting agents and scoreboard

Setup analysis port to transfer data from monitor to scoreboard and coverage

2.4.13 Test(apb\_test.sv):

• Class name: apb\_test

• Define apb\_test class

 Extend apb\_test from uvm\_test

 Usage of macros to register the apb\_test with uvm factory

 Creating a class constructor

 Declaration of a handle for apb\_env

• Implement the build Phase()

 Instantiation of the apb\_env in the build phase

• Check for the end of elaboration

• Implement the report Phase

 Generate report

 Print summary

2.4.14 Top(apb\_top.sv):

• Module name:apb\_top

• Includes uvm packages

• Include uvm macros

• Instantiation of the DUT

• Creating Virtual Interface(vif)

 Instantiation of vif

 Pass the VIF to the UVM environment using `uvm\_config\_db.

• Clock and reset generation

• Start the uvm test [run\_test()]

• Waveform generation