AMBA APB VIP Document

Draft Version

# Revision History:

|  |  |  |  |
| --- | --- | --- | --- |
| ***Version*** | ***Date*** | ***Changes Done*** | ***Author*** |
| Draft | 14/07/2020 | Initial Version | Dheenadayalan & Vaibhav Joshi |
|  |  |  |  |

# Team Members:

1. *Dheenadayalan*
2. *Jagruthi Kappara*
3. *Seshananda Reddy*
4. *Shubham T*
5. *Veeresh*
6. *Vaibhav*

# Introduction

The advanced peripheral bus (APB) is part of the advanced microcontroller bus architecture AMBA protocol family. It defines a low-cost interface that is optimized for minimal power Consumption and reduced interface complexity. The APB protocol is not pipelined, use it to connect to low-bandwidth peripherals that do not Require the high performance of the AXI protocol. The APB protocol relates a signal transition to the rising edge of the clock, to simplify the Integration of APB peripherals into any design flow. Every transfer takes at least two cycles.

The APB can interface with:

• AMBA advanced high-performance bus (AHB)

• AMBA advanced high-performance bus LITE (AHB- LITE)

• AMBA advanced extensible interface (AXI)

• AMBA advanced extensible interface LITE (AXI4-LITE)

**Transfer methods:**

Write transfer.

This section describes the following types of write transfer:

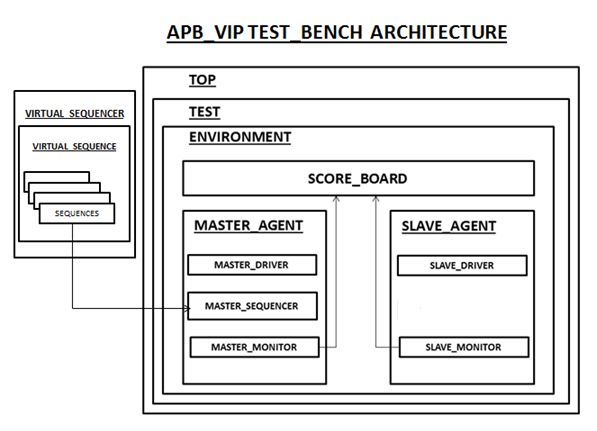
* With no wait states.
* With wait states.
* Write strobes.

Read transfer.

This section describes the following types of read transfer:

* With no wait states.
* With wait states.

# Test Bench Architecture

****

**Agent**

The most important and basic element in UVM Architecture is the Universal Verification Component (UVC) or Agent. Because of Agent, the Test-bench of UVM is re-usable. Agent is an encapsulation of Driver, Monitor and sequencer. An UVM environment can consists of one or more agents. Agent can be configurable. There are two types of Agents-Active agent and Passive agent. If the agent is Active, then the agent will have all the driver, monitor and sequencer. But if the Agent is Passive, then the agent will have only Monitor.

**Driver**

Driver is an active entity that emulates the logic that drives. It fetches data repeatedly from the sequencer. Driver has to drive according to the protocol using the interface. Driver drives the data to slave using interface.

**Monitor**

Monitor is a passive entity that can sample the signals, but does not drive them. A monitor extracts signal information from the bus and translates the information into a transaction that can be made available to other components and to the test case writer.

**Sequencer**

The sequencer in UVM just acts as a gateway between sequence and driver. This is the only “non-virtual” class in our UVM test bench architecture. The sequencer is also parameterized by transaction class. The sequencer takes the randomized data from sequence and passes it to the driver to which it is connected, thus it connects several sequences to driver.

**Sequence**

The actual driven data is randomized in the sequence. From the sequence, this randomized data is given to driver via sequencer.

**Test**

Test is a place where we start the sequences on sequencer. In base test we will set all the parameters of configuration database class according to our requirements. We will also get the interface set in top and again set it to the interface handles in local configuration database classes. The base test also creates the environment. All these things happen in build phase of base test. The further child tests will be made from this base test class. In child test we just create the handle of virtual sequence and start it on virtual sequencer, before starting the sequence an objection is raised and this objection is dropped again after starting the sequence. If we don’t raise the objections, the simulator will think that there is no run phase to execute, so the simulator can jump directly to extract phase. The total number of raised objections should be equal to the number of dropped objections.

**Environment**

The object of this class is created in test and is a most important component of UVM test bench. Environment creates agents, scoreboard, virtual sequencer etc. The environment makes connection between monitors and scoreboard. If it has virtual sequencer, here in environment we have to connect physical sequencers with the handles of physical sequencers in virtual sequencer.

**Virtual Sequence/Virtual Sequencer**

To reduce the dependency of test case writer and TB developer, these two virtual sequence and virtual sequencer we are considering.

**Scoreboard**

It compares the data sent from one side to another i.e. from master to slave and vice versa. This also includes the cover groups which help in knowing the functional coverage. It gets the data from both write monitor and read monitor and compares both of them. Also it samples the signal values which have been covered.

**Configuration database**

It is reusable and efficient mechanism for organizing the configuration of the test bench. If we want to build re-usable test bench, we have to design Test bench such that the components are easily configurable. To build a re-usable test bench, the main thing is UVM Configuration. We have to use configuration API uvm\_config\_db.

**Coverage**

Coverage is a generic term used to measure the progress to complete verification for any VIP. The coverage tools gather information during the simulation and post process it to produce a progress report. We can use this report to look for coverage holes and then modify existing test cases or create new test cases to fill the holes. This iterative process continues until we are satisfied with the coverage level. There are two types of Coverage.

**Code Coverage**

The easiest way to measure the Verification progress is with Code Coverage. Here we are measuring how many lines of the RTL code have been executed (Line coverage), which paths through the code and expressions have been executed (Path Coverage), which single variables have had the values 0 or 1(Toggle Coverage). We don’t need to write any extra HDL Code. The tool captures our design automatically by analyzing the source code and tool will add the hidden code to gather statistics about the code coverage. Code Coverage is used to measure the efficiency of verification process. Code coverage provides a quantitative measurement of the testing space. It describes the degree to which the source code of apb has been tested. Code coverage is simulator dependent. Code Coverage is also called as “Structural Coverage”.

**Functional Coverage**

Code coverage does not know anything about what the design is supposed to do. There is no way to find what is missing in the code by the code coverage, but a functional coverage can catch the missing functionality in the Design code. The main goal of verification is to make sure that a design behaves correctly in its real environment. Functional Coverage is user specified to tie the verification environment to the design intent or functionality. Functional coverage tells us about whether all the functionalities given in specification are included in the RTL or not. Functional coverage is also called as “Specification Coverage”.

# Test Plan:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | S.No | Category | Test\_case | Description | | 1 | Reset | hard reset | Checking the default value. | |  |  | mid\_reset | Calling reset after 1 transfer and checking default value. | | 2 | Error | Write transfer | Writing data to slave without enabling the slave select or slave enable | |  |  | Read transfer | Without writing data to slave, read the data from slave.(no data) | | 3 | Basic Transfer | write transfer and read | Consecutive write and read | |  |  | With wait states | Consecutive write and read (PREADY is asserted 1 clock cycle or later after PENABLE and PSEL are asserted. | |