AMBA TestBencher Pro Example

Table of Contents

1.Overview

This example demonstrates how to create and use an AMBA AHB master and is packaged with TestBencher Pro in under the <INSTALL>\Examples directory. There is no model under test which means there is no slave device to respond to the master. This will cause errors during the simulation which are all output by the master BFM. There are two TestBencher Pro projects contained in this example: amba.hpj and ahb_master.hpj. The ahb_master transactors can be pipelined and make extensive use of "Pipeline Boundary" markers to do so. So, you may want to read through the Pipeline example's documentation for more details on how "Pipeline Boundary" markers work. Some of the notable features used in this example are:

1. "Pipeline Boundary" Markers – all master transactors – used to model the pipeline behavior defined by the AMBA specification.
2. Blocking Samples – master write and read – used to wait for HREADY.
3. State Variables – all master transactors.
4. Loop Markers – master idle and busy transactors – used to insert variable number of idle or busy cycles.
5. "Store Sampled Value As Subroutine Output" - master write transactor – used to pass the read data back to the sequencer process.
6. User-defined Class Method – used on AMBA.hpj component model to determine burst length based on a given burst code (defined by AMBA spec.).
7. Constrained randomization – idle and busy transactors are applied using the "random" versions of their apply calls which results in a random number of idle or busy cycles.

2.ahb_master.hpj

The master BFM contains four transactors: write, read, idle, and busy. Each of these transactors were created so that they would handle the pipelining behavior of the AMBA bus protocol. Basically, only one diagram can be executing the address phase at any given time. And the same goes for the data phase. But, a data phase can occur while an address phase is occurring. "Pipeline Boundary" Markers are used to handle this. With these types of markers, you define the pipeline phases in the diagram. Then, when the transactor is applied multiple times, concurrently, pipelining will occur based on the pipeline phases you have defined. Also, each pipeline phase is associated with a semaphore name, which will get automatically defined for you. These semaphores are global to all of the diagrams within a given project. In the master project, there are two semaphores used: addr_phase and data_phase. All of the diagrams use the addr_phase semaphore, but only the read and write diagrams use the data_phase semaphore. This should be more obvious as you examine each diagram.

2.1.write.btim

This address starts with the address phase followed by the data phase. If another transactor in the master project is already in the middle of an address phase when this transactor starts, then it will wait for that address phase to finish. This is all handled by the "Pipeline Boundary" Markers. Here is the sequence of events that occur for this transactor:

1. addr_phase Marker: Wait for address phase to finish if there is one in progress. This is done by waiting for the addr_phase semaphore to be greater than 0. Then, it will decrement the semaphore and continue with the address phase. By default, semaphores start out with a value of 1.
2. Execute the address phase.
   a) Drive HTRANS to $$trans (state variable passed in as parameter).
   b) Drive HADDR to $$addr.
   c) Drive HWRITE high to indicate that this transaction is a write.
   d) Drive HSIZE to $$size.
   e) Drive HBURST to $$burst.
3. data_phase Marker: Increment the addr_phase semaphore and wait for the data_phase semaphore to be greater than 0. Zero indicates that another transactor is still in the middle of the data phase. When it becomes > 0, decrement and continue with the data phase.
4. Execute the data phase.
   a) Drive HTRANS to IDLE.
   b) Drive HWDATA to $$data.
5. WaitForSlaveReady Sample: This sample has a "Multiplier" of 100, "Full Expect" is disabled, and "Blocking" is enabled. This configuration will cause this sample to block the diagram from executing, up to 100 clock cycles, until HREADY is asserted. If HREADY doesn't assert within that amount of time, an error message will be displayed. In this example, there is no slave BFM and no slave MUT, so you *will* see this error during simulation since there is no slave to respond.

6. end_phase Marker: Increment the data_phase semaphore. This indicates that the data_phase is complete and will allow another transactor to perform its own data phase if it was waiting.

2.2.read.btim

This transactor is very similar to the write transactor (see details above). The difference is that $$data is no longer passed in as an argument. Instead, the SampleData Sample will read the data driven by the slave and return it as an output parameter. Notice that the same semaphore names are used with the “Pipeline Boundary” markers. This is very important. Otherwise, if you applied a read and a write in parallel they'd both execute the address phase and data phases at the same time.

2.3.idle.btim

This transactor simply drives 00, indicating IDLE, on HTRANS for a specified number of cycles. The number of cycles is passed in as an argument to the apply call. This input variable is defined in the “Variables” dialog for the diagram. Note again, that the same semaphore name is used for the address phase, addr_phase.

2.4.busy.btim

This transactor works very much the same way as the idle transactor except that it drives 01, indicating BUSY, on HTRANS. The second difference is that while the bus is BUSY, the address and control signals must reflect the next transfer in the burst. So, these values are all passed in via state variables: $$addr, $$write, $$size, $$burst.

3.amba.hpj

This top level project instantiates one master and calls its various transactors. Again, there's no slave MUT, but if you have an AMBA AHB slave device, you could place it in this project as the MUT. This project also contains a clock transactor and a reset transactor.

3.1.Sequencer Process (Component Model)

Double-click on the Component Model for this project and search for "Sequencer Process". This is very near the bottom of the file. This process contains a directed sequence of transactor calls. Here's a summary of what the sequencer process does:

1. Start the clock generator.
2. Apply the reset transactor.
3. Perform the following actions 5 times:
   a) Perform a random number of IDLE cycles (see next section for how this works).
   b) Randomize the data to be written.
   c) For the length of the burst
      1. Insert a random number of BUSY cycles (see next section for how this works).
      2. Perform the write.
      3. Randomize the data to be written next.
4. Stop the clock generator.

3.2.How "ApplyRandom" Works

Every transactor has a set of apply tasks associated with it that can be used to run the transaction is various ways. If you're looking at the Verilog example, you can see these tasks by scrolling through amba.v (Component Model). In VHDL, these tasks (actually procedures in VHDL) are generated to a package in a separate file named <projectname>_tasks.vhd. In both cases, there is are "Apply" and "ApplyRandom" tasks.

3.2.1.The "Apply" Tasks

You should already be familiar with the normal "Apply" tasks. When you call these, you must specify *all* of the inputs for the transactor.

3.2.2.The "ApplyRandom" Tasks

When you use "ApplyRandom", you do *not* have to specify those parameters that are marked as "random". Instead they are randomly generated based on a set of simple constraints. By default, all parameters are marked as random. So, if you call "ApplyRandom" then you don't need to pass in any transactors arguments. Note that the "random" option has no affect on how the normal "Apply" tasks work. To enable or disable "random" for a particular argument do the following:

1. Right-click on the Component Model that contains the transactor.
2. Select “Classes and Variables.”
3. In the “Class Definitions” tab, select the class that contains the arguments for the transactor. These classes are named “<transactorName>_Parameters” and are automatically created when you generate the test bench (Make TB).
4. Select the argument name in the “Class Fields” list.
5. Change the “Random” option by double-clicking the field and selecting what you want. By default, these are all “rand”. “randc” means “cyclic random” and is *not* supported in Verilog and VHDL.
6. You can setup the constraints for the “rand” fields by clicking the “Constraints” button. By default, the constraints will be set up based on the size and type of the signal (i.e. the full range of the field is the constraint).