PCI TestBencher Pro Example

Table of Contents

1.Overview....2
2.Master.hpj....3
    2.1.write.btim....3
3.Slave.hpj....4
    3.1.write.btim....4
4.PCI.hpj....5
    4.1.arbiter.btim....5
    4.2.Sequencer Process (in Component Model)....5

1.Overview

This example demonstrates how to create PCI master and PCI slave bus-functional-models (BFMs) using TestBencher Pro. These are not 100% complete BFMs that can test every feature of the PCI protocol. Instead, they are just partial BFMs to help in the understanding of TestBencher Pro. Each BFM is modeled using a TBP project, which is then instantiated in another project named PCI.hpj. Some features used in this example include:

1. Component Signals & Ports dialog – used to specify ports of slave and master BFMs.
2. "Loop" Markers – several of the transaction diagrams.
3. "Wait Until" Markers – master's transaction diagrams and arbiter.btim.
4. "HDL Code" Markers – slave's transaction diagrams.
5. State Variables – master's transaction diagrams.
6. Blocking, Simple Expect Samples – master's and slave's transaction diagrams.
7. Non-blocking, Simple Expect Samples – master's transaction diagrams.
8. User-defined condition Samples – slave's transaction diagrams.
9. "Trigger Sample" Samples – arbiter.btim
10. "Do Delayed Transition" Samples – arbiter.btim
11. Sensitive Edges – slave's transaction diagrams and arbiter.btim.

This PCI example consists of the following TBP projects:

1. slave.hpj – represents a PCI slave bus-functional-model (BFM).
2. master.hpj – represents a PCI master BFM.
3. PCI.hpj – instantiates two masters (master0 and master1) and one slave (slave0).

Open PCI.hpj and look at the Project window to see the hierarchy of projects. The instantiations are displayed below the top level Component Model. You can browse the contents of master.hpj and slave.hpj via the Project Library folder.

2.Master.hpj

The master project contains a two diagrams: write.btim and read.btim. These diagrams are fairly similar so only the write.btim diagram is explained here.

The master write diagram is a "Master Transaction" which means when it is applied, it will run once from beginning to end then stop. This setting is accessible from the "Diagram Settings" dialog. This transaction takes three parameters: numDataCycles, $$address and $$be. "numDataCycles" is a user-defined variable which can be viewed by clicking the "View Variables" button in the diagram window.

2.1.write.btim

1. Assert REQn.
2. Wait for acquisition of the bus using the "WaitForBusAcquisition" Wait Until Marker. Double-click on the marker name to see the "Wait Until" condition.
3. Address Cycle:
   a) Assert FRAMEn to indicate start of transaction.
   b) Put address on AD using the "$$address" state variable that was passed in as a parameter.
   c) Drive out "7" on the CBEn signal to indicate the "write" command.
4. Data Cycle(s): A for loop is used to perform all of the data cycles except for the very last one. Open up the for loop's properties to see that it loops from 0 to "numDataCycles-2", inclusive.
   a) Put data on AD using the "writeData" Component Model Variable. The "writeData" variable is defined in the "Class Libraries and Variables" dialog for the master project.
   b) Drive out byte enable on the CBEn signal using the "$$be" state variable that was passed in as a parameter.
   c) Assert IRDYn.
   d) Wait for TRDYn to assert, which indicates that the slave has seen the data and is ready to continue. This is done using the "WaitForTRDY" Sample. Double-click on the Sample's name and then click on the HDL Code button to see it's "Code Generation Properties." The Multiplier is set to 100, which indicates that the sample should wait up to 100 clock cycles for TRDYn to assert. "Full Expect" is disabled because otherwise it would mean that TRDYn should be asserted for the entire 100 clock cycles. And finally, "Blocking" is enabled to block the rest of the diagram from executing until the sample finishes.
5. Last Data Cycle: this is the same as the other data cycles, except that FRAMEn is released to indicate that the transaction is finishing.

Keep in mind that there are other ways to implement a PCI write transaction. For instance, all of the data cycles could have been done within the for loop, instead of putting the last data cycle afterwards. In that case, you would have to use a conditional expression to release FRAMEn for the last data cycle. For example, the state for FRAMEn inside the for loop would look like:

(cycle == numDataCycles) : Z : 0

3.Slave.hpj

The slave project contains two diagrams: write.btim and read.btim. As in the master project, these diagrams are closely related and only the write.btim will be explained here.

The slave read diagram is a “Slave Transaction” which means that it will loop continuously when applied. Instead of being applied once for each write transaction, it is applied once at the beginning of the test bench and determines when to perform a write based on the FRAMEn, AD, and CBEn signals.

3.1.write.btim

1. Wait for a falling edge on FRAMEn. This was set up by enabling “Falling Edge Sensitive in the “Signal Properties” dialog for FRAMEn then drawing the falling edge where it should wait. This will block the entire diagram from running until that falling edge is detected.
2. Address Cycle:
   a) DecodeAddress Sample: read address from AD and restart transaction if AD[31:28] is not “F”. This is an example of how to react to a given address range. This address range was chosen arbitrarily.
   b) DecodeCommand Sample: read command from CBEn and restart transaction if the command is not “7” (write).
3. Data Cycle(s): A while loop is used to perform all of the data cycles except for the last one. It will loop until FRAMEn is de-asserted.
   a) WaitForIRDY Sample: waits until IRDYn is asserted. The Multiplier is set to 100, Full Expect is disabled, and Blocking is enabled. This combination of settings will cause the sample to wait up to 100 clock cycles for IRDYn to assert while blocking the diagram from executing.
   b) SampleData Sample: reads the data from the AD bus and stores it in a project level variable named “sram”. To see where this variable is created, right click on slave.hpj project in the Project Window -> Classes and Variables -> Variables.
   c) IncAddress Marker: increments the address.
   d) If FRAMEn is detected to be de-asserted at the end of the data cycle, then two delays will be triggered: Release_TRDY and Release_DEVSEL.
4. Last Data Cycle: After FRAMEn de-asserts, one more data cycle occurs. This data cycle doesn't look any different than the rest of the data cycles, but two more Samples must be used since it doesn't occur within the loop: LastSampleData and LastWaitForIRDY. These two Samples are equivalent to the SampleData and WaitForIRDY Samples that are used within the loop.

To help with starting and stopping this slave BFM, there were two tasks written in the Component Model (slave.v).

1. StartRunning – calls “Apply_nowait” for both the read and write transactions.
2. StopRunning – calls “Abort” for both the read and write transactions.

4.PCI.hpj

This is the top-level project of this example and contains one instance of slave.hpj, two instances of master.hpj and three diagrams: clk_generator.btim, reset.btim, and arbiter.btim. The clock generator and reset should be self explanatory. The arbiter.btim diagram is explained below. And following that is an explanation of the sequencer process contained in the Component Model for this project (PCI.v).

4.1.arbiter.btim

1. Wait for RSTn to de-assert. This is done by making RSTn a "Rising Edge Sensitive" signal and drawing a rising edge where the diagram should wait.
2. InitLoop Marker: Waits for 5 clock cycles.
3. MainArbiterLoop Marker: loops forever
   a) WaitForRequest Marker: waits for the assertion of any request signal.
   b) Release previous GNT assertion if any GNT is asserted. This is done using a conditional expression for each of the GNT signals. If GNT is asserted and REQ is not, then GNT will be released. If GNT is asserted and REQ is still asserted, then GNT will be left asserted. Note: Because of this the arbiter is not necessarily very fair.
   c) CheckReq0 Sample: if REQ0 is asserted, then GNT0 will be asserted by the AssertGNT0 delay, which is triggered by this sample. Otherwise, the Else_CheckReq1 Sample is triggered and this sequence continues until the arbiter determines which request was asserted.
   d) WaitForTransStart Sample: waits for FRAMEn to assert which indicates that the transaction has started. The Multiplier is set to 16, Full Expect is disabled, and Blocking is enabled. This combination will make the arbiter wait up to 16 clock cycles for FRAMEn to assert. If FRAMEn doesn't assert within that time, a "Failure" message will be displayed and the simulation will be halted. This failure action is indicated by the "Else Action" for the Sample.
   e) WaitForLastDataPhase Sample: waits for FRAMEn to de-assert. Works similar to the WaitForTransStart sample, except that the multiplier is set to 100, so it will wait up to 100 clock cycles. This number was arbitrarily chosen for this example.

4.2.Sequencer Process (in Component Model)

Near the bottom of this file is the sequencer process (search for "Sequencer process" to find the beginning of the process). It does the following:

1. Starts clock generator and arbiter.
2. Applies the reset transaction.
3. Starts the slave BFM.
4. Performs some writes and reads using the master instances.
5. Stops the slave BFM, arbiter, and clock.