UARTTEST TestBencher Pro Example

Table of Contents

1.Overview

This example can be found in the Examples\TestBencher directory of your SynaptiCAD product installation. It contains three transactors: CLK_generator, WriteSerial, and ReadSerial. There is not a Model Under Test (MUT) in this example. Instead, the WriteSerial and ReadSerial transactors communicate with each other. They send and receive serial data on a signal named UART. But, they work with parallel data at the transactor level. For example, the write transactor has an eight bit argument which it then converts to serial data. Both of these diagrams also have a parameter named "speed" which controls how many clock cycles are to be used for each bit of data. Here are a few of the features used in this example:

1. Input Variables – WriteSerial – used to create "data" argument.
2. Output Variable – ReadSerial – used to send 8-bit data back to sequencer process.
3. Store Sampled Value As Subroutine Output – ReadSerial – stores read in data.
4. Variable Clocked Delays – WriteSerial – used to control how many clock cycles are used for each bit of data.
5. Reference Model – used to check that the data received by ReadSerial is the same as the data written by WriteSerial.
6. For Loop Markers – ReadSerial – used to insert variable number of clock cycles.

2.WriteSerial.btim

This transactor takes two arguments: data and speed. The “data” argument is an 8-bit value that will be output one bit at a time on the UART signal. This argument is defined in the “Variable List” dialog for the diagram, which can be opened by clicking the “Variables” button. The “speed” argument determines how many clock cycles to use for each bit. This argument is defined in “Parameter” window as a “Free Parameter” since it will be used to control Delay behavior.

Note: The project window can be used to display an overview of all the arguments to a particular transaction. For example, go to the Project window and browse to the following:

Component Model -> Class Library List -> uarttest.hpjl

There are two classes defined here: uarttest_WriteSerial_Parameters and uarttest_ReadSerial_parameters. Each of these contains all of the arguments for their respective transactors. You can also right-click on the Component Model and select “Classes and Variables” to bring up a dialog that lists all of the classes. These “Parameters” classes are created automatically for you bases on the different types of arguments you have in the diagram. These include State Variables, input and output variables from the Variable List, and “Applied Subroutine Input” Parameters such as Delays and Samples.

Here's a breakdown of the events that occur when this transaction is applied:

1. Drive out the first bit of data.
2. Wait for a positive clock edge.
3. Wait for a delay of D1. The value of this delay is “speed – 1.0”, which represents clock cycles since “Count Clock Edges” is set to CLK pos. For example, if speed is 1 then there will be no delay inserted. If speed is 2, then there will be 1 clock cycles inserted (in addition to the clock edge we already waited for).
4. Drive out the second bit of data.
5. Wait for a delay of D2. The value of this delay is “speed*2 – 1.0”. If speed is one then 1 clock cycle will be inserted. If speed is 2, then 3 clock cycles will be inserted, etc.
6. Continue driving bits based on the delays drawn in the diagram.

Note: You can see how a particular “speed” affects the diagram by changing its value in the “Parameter” window. This value has no affect on simulation since it has “Is Apply Subroutine Input” checked. To change this value, go to the “Parameter” window and double-click on “speed”. It will bring up a “Free Parameter Properties” dialog where you can modify the Min value. When you apply changes to this dialog the waveform will update.

3.ReadSerial.btim

This transactor reads serial data from the UART signal and returns it as byte named "data." It also has input argument named "speed". Like the WriteSerial transactor, the "speed" argument sets how many clock cycles are used per bit. But, in this case a for loop makes use of "speed" to insert extra clock cycles. Note that a similar for loop could have been used in the WriteSerial transactor. Here's the sequence of events for this transactor:

1. For each bit (loop started by BitLoop Marker)
   a) SAMPLE0: Store the bit in the appropriate location of the "data" variable. Double-click on SAMPLE0, then click on "HDL Code". "Store Sampled Value to Variable" is set to "@data[bitCount]".
   b) Wait for a rising clock edge
   c) InsertClockCycles Marker: this loop will insert the additional clock cycles required based on the "speed" variable. Double-click on the marker's name to see how the for loop is configured.

4.Sequencer Process (in Component Model)

Double-click on the "Component Model" in the Project Window, uarttest.v, to open it up. The sequencer process is near the bottom of the file. This process is where the test bench sequence is controlled. It tests out the WriteSerial and ReadSerial transactors using various speeds (i.e. various number of clock cycles per bit). The ReadSerial transactor responds to data driven by the WriteSerial transactor, so for each WriteSerial transaction applied there is a corresponding ReadSerial transaction applied in parallel. Here is a breakdown of what this process does:

1. Start clock generator.
2. For speeds between 1 and 6
   a) Start a WriteSerial transaction. A "nowait" apply call is used so that we can immediately start up a "ReadSerial" transaction.
   b) Start a ReadSerial transaction. A "blocking" apply call is used here so that the sequencer process waits for the transaction to finish before continuing to the next speed.
3. Stop the clock generator.

5.Reference Model

A reference model is being used in this example to help verify that the UART transactions operate correctly. This option is enabled by right-clicking on the Component Model, selecting "Project Generation Properties, and selecting "Enable Reference Model". When this is enabled, a skeleton reference model will be generated during "Make TB". This skeleton is put in the Component Model -> Associated Files folder and has a "txt" extension. The uarttest_emulator.v module was created from this skeleton. Open up both of these files to get an idea of what TestBencher will generate and what was added to make a valid reference model.

5.1.How it Works

Each transactor in the project has a corresponding task in the reference model. These tasks have the same set of arguments (both input and output) as the transactors. When a "Master Transactor" is applied, the corresponding reference model task will be applied once the transactor finishes. The same inputs are passed into the reference model task as were passed into the transactor. Then, the outputs of the reference model are compared to the outputs of the transactor. Any differences are reported to the simulation log.

When a "Slave Transactor" is applied, the reference model task will be called for each completion of the transaction. In this example, only the CLK_generator is a slave transactor so it's not too interesting. See Possible Enhancements to this Test Bench for how ReadSerial could have been implemented as a "Slave Transactor".

6.Possible Enhancements to this Test Bench

1. Use “ApplyRandom” diagram calls. Instead of calling “Apply_WriteSerial_nowait”, you could call “ApplyRandom_Serial_nowait” where the data and speed would be randomly generated. Or you could set it up so that only the data was randomly generated. For more information on “ApplyRandom” see the “AMBA” pdf. There is a section named “How ApplyRandom Works.”
2. Make ReadSerial a “Slave Transactor”. If ReadSerial were a slave it could be started up at the beginning of the test bench with a given speed and set up so that it responds to all of the WriteSerial transactions. The catch here is that the speed passed into WriteSerial must match the speed passed into ReadSerial, so all of the WriteSerial transactions would have to run at the same speed for a given ReadSerial transaction instance. Of course, you could always stop the ReadSerial slave and start it back up with a different speed. In addition to this consideration, ReadSerial would also have to be set up so that it knows when a transaction is starting. This may be as simple as using a control signal.
3. Use for loop in WriteSerial (i.e. like ReadSerial). A similar for loop could have been set up in WriteSerial, in which case none of the delays would be necessary. In this case, the state drawn inside of the for loop would be “@data[index]”.