Pipelining TestBencher Pro Example

Table of Contents

1.The Pipelining Example

The purpose of the Pipelining example is to demonstrate how pipelined transactors can be created and how they work. There is no model under test used in this example. Instead a simple protocol was invented just to allow the demonstration of "Pipeline Boundary" markers. The address, control, and data signals are driven in three successive clock cycles. There are three diagrams in the Pipelining project:

1. CLK_generator – drives the clock signal.
2. read – performs a three cycle read. Since there isn't a MUT, the data bus will actually be tri-stated during the data phase.
3. write – performs a three cycle write.

The read and write diagrams are very similar. There are two differences:

1. The "WRITE" signal is asserted in the write diagram and de-asserted in the read diagram.
2. The "DATA" signal is specified to be driven in the write diagram, but not in the read diagram. This is set via the “Driven” check box in the “Edit Bus State” dialog (double-click on state to open).

To create the pipeline phases, "Pipeline Boundary" Markers were placed on each clock edge that starts and/or ends a pipeline phase. For each "Pipeline Boundary" Marker that starts a phase, a semaphore name is specified. This is done in the “Edit Marker” dialog and can be any valid identifier. To indicate the end of the last pipeline phase, you can either select “End Boundary” as the semaphore name or create an “End Diagram” marker instead. There are three phases in each of the diagrams: addr, control, and data. This allows for the following simulation results:

CLK1	***CLK2***	***CLK3***	***CLK4***	***CLK5***
addr1	control1	Data1		
	addr2	control2	data2	
		addr3	control3	data3

Apply-nowait diagram calls are also necessary to get these results. For example, to perform 3 pipelined writes, you'd have the following task calls in your sequencer process:

```
Apply_write_nowait(addr1, data1);
Apply_write_nowait(addr2, data2);
Apply_write_nowait(addr3, data3);
```

Note that the final apply call does not need to be a "nowait" call. In fact, if it's the last apply call in your sequencer process you should probably use the "blocking" call, Apply_write, so that the sequencer is blocked from finishing until the last write completes.

2.Pipeline Boundary Markers

Details on how diagrams operate in general can be found in the TestBench Generation Help in section 3.5 "Transaction Architecture". Here, we'll describe specifically how to create "Pipeline Boundary" Markers and how they work.

2.1.Creating Pipeline Boundary Markers

Here are the steps required to create a “Pipeline Boundary” marker.

1. If you want the boundary to be at a clock edge, select (left-click) the clock edge.
2. Right-click to place the marker.
3. Double-click the marker or the marker's name to bring up the “Edit Time Marker” dialog.
4. For Type, select “Pipeline Boundary.”
5. Under “Semaphore for next phase”, select the semaphore name from the drop down list. (Note: Select “End Boundary” if you are ending a pipeline phase but not starting a new one.) If there are no semaphore's created yet, you can do either of the following:
   a) Type in a new semaphore name and it will be created for you automatically.
   b) Select <Edit> from the drop down which will bring up the “Semaphore List” dialog. This is where you edit the semaphores for a given project. Once you've created the semaphore, you can close the “Semaphore List” dialog and the new semaphore should be available in the drop down.

After creating two “Pipeline Boundary” markers in the diagram to define a phase, a line with an arrow on each end will be drawn between the two markers. The name of the semaphore used will be displayed above this line.

Note: like looping markers, “Pipeline Boundary” markers must begin and end on the same type of edge to define a pipeline phase. Otherwise, they would be in different clock domains. For more details on clock domains see section 3.5 “Transaction Architecture” in the TestBencher help.

2.2.How Pipeline Boundary Markers Work

The following two sections describe the two features that go into making pipelined transactors work.

2.2.1.Multiple instantiations of a single diagram

This is what allows the diagram to run multiple times in parallel. The diagram is automatically instantiated once per pipeline phase, so you don't need to worry about this detail when using "Pipeline Boundary" markers. If you're interested in setting the instantiation number manually, you can do so by setting the “Instance Count” in the Diagram Settings dialog.

2.2.2.Semaphores

These are used to block regions of the diagram from running at the same time. By default, semaphores will be initialized to 1. Semaphores are incremented and decremented like integers. When a semaphore is > 0 then it is considered “available”.

In this example there can only be one addr phase running at a time. When the write is applied, it will decrement the addr semaphore count and enter the addr phase. When it leaves the addr phase, it will increment addr back to 1. In the meantime, if any other diagram tries to grab the addr semaphore, it will wait until the addr semaphore's count is incremented.

Note: The semaphores that are used in TestBencher are queue based. So, the order is maintained for all the waiters of a given semaphore.

There are "Semaphore" markers that can be used to wait on or post a specific semaphore. So, you could use "Semaphore" markers to implement a pipelined transactor. But, "Pipeline Boundary" markers reduce the amount of markers that are necessary and avoid the overall complexity of the diagram. The following table compares the two methods.

Using Semaphore Markers	***Using Pipeline Boundary Markers***
Semaphore (Wait) : addr	Pipeline Boundary : addr
perform addr cycle	
Semaphore (Wait) : control	Pipeline Boundary : control
Semaphore (Post) : addr	
perform control cycle	
Semaphore (Wait) : data	Pipeline Boundary : data
Semaphore (Post) : control	
perform data cycle	
Semaphore (Post) : data	Pipeline Boundary : End Boundary

Note how the wait for the next semaphore must happen before the post of the previous semaphore. This is to make sure that the transactor stays in a particular phase. If it posted the semaphore before waiting for the next one, then it could end up being in neither phase. This detail is automated when using Pipeline Boundary markers.

For each semaphore needed a tbfifosemaphore module is instantiated in the component model. That allows the semaphore to be shared between different diagrams in the same project. Currently, there is no way to share one semaphore across multiple projects.

3.Verilog Specific Details

The definition of the tbfifosemaphore module can be found in definition found in lib\verilog\tbfifosemaphore.v. This module's purpose is to preserve the order in which diagrams are waiting for a given semaphore. Here is an example of what is generated for a "Wait Semaphore":

```
   // Wait for Semaphore: pipelining.pipeline_phase_control
   tb_pid = pipelining.pipeline_phase_control.GetPidIfNecessary(0);
   if (tb_pid != 0)
     begin
     while(pipelining.pipeline_phase_control.resume_pid != tb_pid)
       begin
       @(pipelining.pipeline_phase_control.resume_pid);
       end
     end
   pipelining.pipeline_phase_control.WaitComplete(tb_pid);
```

GetPidIfNecessary will return 0 if the semaphore's count is > 0, which means that the transaction doesn't have to wait for the semaphore to be posted (i.e. incremented). If GetPidIfNecessary returns a non-zero number then that number represents the resume_pid to wait for. When a tbfifosemaphore is posted, it will set the resume_pid to the next waiter in line (if there is one). Once the wait is complete, WaitComplete is called on the tbfifosemaphore which will decrement the count. When the transaction wants to release the semaphore, it simply calls Post which will increment the semaphore and set the appropriate resume_pid if necessary.