DesignCon 2003

System-On-Chip and ASIC Design Conference

Accelerating Verification Through Pre-Use of System-Level, Transaction-Based Testbench Components

Jim Lewis
jim@SynthWorks.com

SynthWorks Design Inc.

Abstract

Traditional functional verification methodology first tests subblocks and then later tests the entire chip. To test a subblock, a small testbench is written, used, and then discarded after the subblock is integrated into the chip. To test the chip, a system-level, transaction-based testbench is developed. In the system-level testbench, each interface to the chip is driven by a model. Each test run at the subblock level is repeated at the system level to validate that it also works correctly in the system environment.

This paper shows an alternate methodology that tests subblocks by using (pre-using) components of the system-level, transaction-based testbench. Using this approach, the transactions used for the subblock test can be reused in the system level test. This approach accelerates the verification process by eliminating the custom subblock testbenches and minimizing the amount of porting required to make a subblock test work at the system-level. Each subblock is still tested independently, so this approach will not incur the increased risk that a methodology that skips subblock tests would incur.

Using the "Pre-Use" approach, the verification cycle will be shortened since custom subblock testbenches do not need to be developed. Keep in mind that although the implementation shown in this paper is coded in VHDL, the techniques can be applied to any hardware description language (HDL) or hardware verification language (HVL).

Author Biography

Jim Lewis, Director of Training, SynthWorks Design Inc.

Jim Lewis, the founder of SynthWorks, has sixteen years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis does ASIC and FPGA design, custom model development, and consulting. Mr. Lewis is an active member of VHDL Standards groups including, RTL Synthesis (IEEE 1076.6), Std_Logic (IEEE 1164), and Numeric_Std (IEEE 1076.3). The author can be reached at jim@SynthWorks.com, +1-503-590-4787, or http://www.SynthWorks.com.

Copyright © by SynthWorks Design Inc 2003. All rights reserved.

1. Introduction

Traditional functional verification methodology first tests subblocks and then later tests the entire chip. To test a subblock, a small testbench is written, used, and then discarded after the subblock is integrated into the chip. To test the chip, a system-level, transaction-based testbench is developed. In the system-level testbench, each interface to the chip is driven by a model. Each test run at the subblock level is repeated at the system level to validate that it also works correctly in the system environment.

This paper shows an alternate methodology that tests subblocks by using (pre-using) components of the system-level, transaction-based testbench. Using this approach, the transactions used for the subblock test can be reused in the system level test. This approach accelerates the verification process by eliminating the custom subblock testbenches and minimizing the amount of porting required to make a subblock test work at the system-level. Each subblock is still tested independently, so this approach will not incur the increased risk that a methodology that skips subblock tests would incur.

This paper gives background on the test process. It provides a methodology overview of the "Traditional", "System Only Tests", and "Pre-Use" approaches. It shows the benefits of transaction-based testbenches. Finally, the paper covers a VHDL implementation of a system-level, transaction-based testbench that can be used to test both subblocks and the entire chip. Keep in mind that although the implementation shown is in VHDL, the techniques shown can also be applied to any other HDL or HVL.

2. Testbench Approaches

2.1 Background

The chip being tested in this paper is named MemIO. A block diagram for the system that uses the MemIO chip is shown in Figure 1.

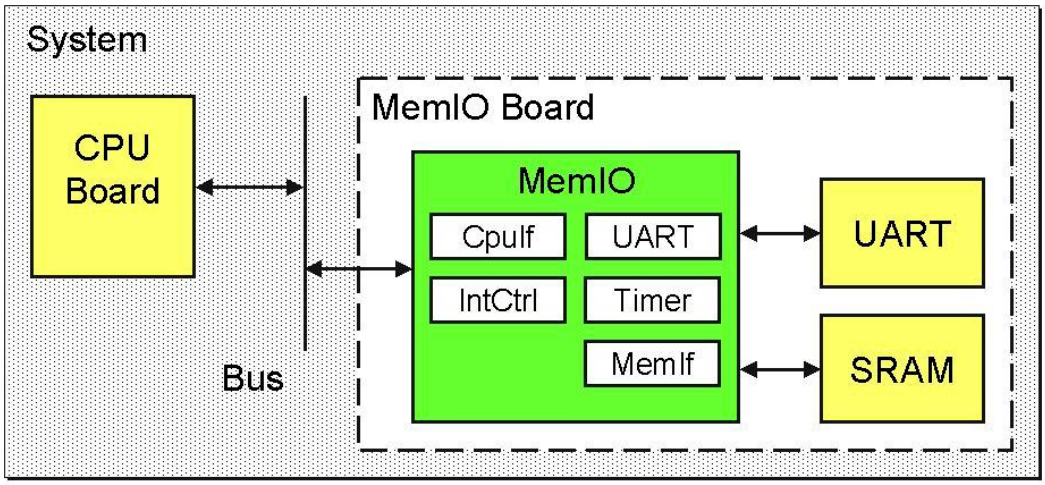

Figure 1. Block Diagram showing MemIO,the unit under test (UUT)

The MemIO chip connnects a CPU to an SRAM and an RS-232 serial port. In addition, it has a programmable timer and interrupt controller.

 Copyright © SynthWorks Design Inc

The basic steps to design and test a chip is shown in Figure 2. The chip design starts with a specification for a chip. The specification contains block diagrams, timing, and detailed descriptions of each subblock in the chip. The next step is to code and test each subblock. Typically each subblock is tested after it is coded and before coding the next subblock. After all subblocks are tested, then simulations containing multiple subblocks and finally the entire chip are run. After the chip is synthesized (converted from RTL code to gates), gate level simulations are run on the chip.

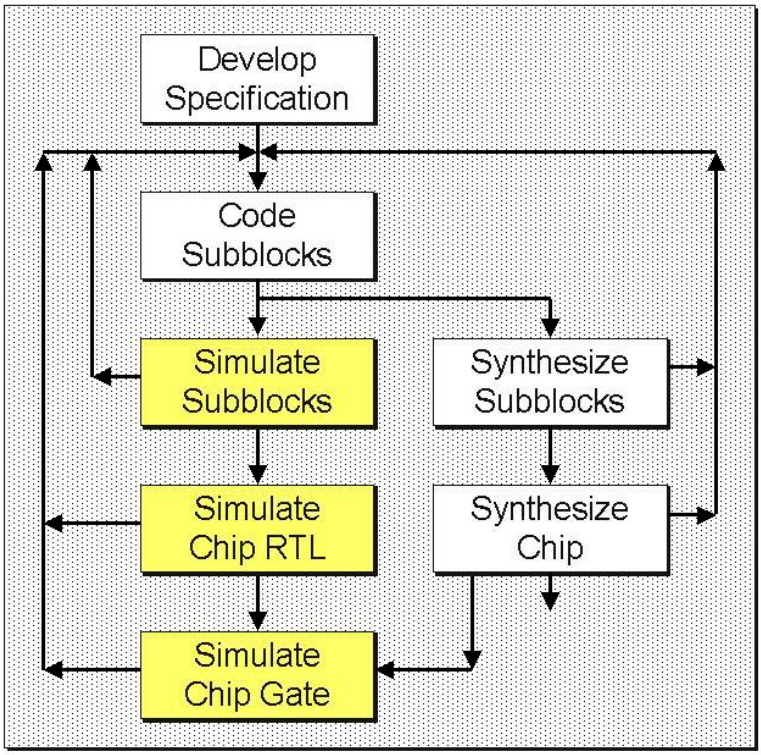

Figure 2. Design Flow for an HDL based design

2.2 Traditional Approach

The traditional test approach starts with coding custom subblock testbenches. A testbench is a code structure that allows waveforms (stimulus) to be driven to the design under test and validates the results. An outline of a subblock testbench for CpuIf is shown in Figure 3. Note ". . ." is used to show where details are left out. The testbench consists of a reference to CpuIf, (the unit under test or UUT), a concurrent statement to create clock, a process to create reset, and an additional process to create CPU bus cycles. In this testbench a CPU bus cycle is created by driving values on individual signals.

 Copyright © SynthWorks Design Inc

```
library IEEE ;
    use ieee.std_logic_1164.all ;
entity TbCpuIf is
end TbCpuIf ;
architecture tb of TbCpuIf is
begin
  U_CpuIf : CpuIf
    port map ( . . . ) ;

  Clk <= not Clk after 10 ns ;

  ResetProc : process  . . .

  CpuProc : process
  begin
    -- Wait until UUT out of reset
    wait until nReset = '1' ;
    wait until Clk = '1' ;

    -- Write X0000 to UART_DIVISOR_HIGH register
    nAds <= '0' after tpd, '1' after tperiod + tpd;
    Addr <= UART_DIVISOR_HIGH after tpd ;
    Data <= X"0000" after tperiod + tpd ;
    Read <= '0' after tpd;
    wait on Clk until nRdy = '0' and Clk = '1' ;

    -- Write X000A to UART_DIVISOR_LOW register
    . . .

  end process ;
end tb ;
```

Figure 3. CpuIf Subblock Testbench

To test the subblocks, a custom subblock testbench similar to the one shown in Figure 3 for CpuIf needs to be created for the remaining subblocks: IntCtrl, MemIf, UART, and Timer. Some of these blocks have similar interfaces (such as the one to CpuIf) and would benefit from sharing portions of their testbenches. Unless this is specifically planned into the tests, it is uncommon for this to happen.

After subblock testing is completed, system-level testing begins. In a system-level testbench, the chip is immersed in a system like environment. Each interface in the system is represented by a model. During system test, each subblock is re-validated to prove if functions correctly with the other subblocks in the design.

Figure 4 shows a block diagram of a system-level, transaction-based testbench. The system testbench consists of a reference to MemIO (UUT), system component models (ClockReset, CpuModel, UartTxBfm, UartRxBfm, and SramModel), and a transaction controller (TestControl). Details of these blocks are presented towards the end of the paper.

 Copyright © SynthWorks Design Inc

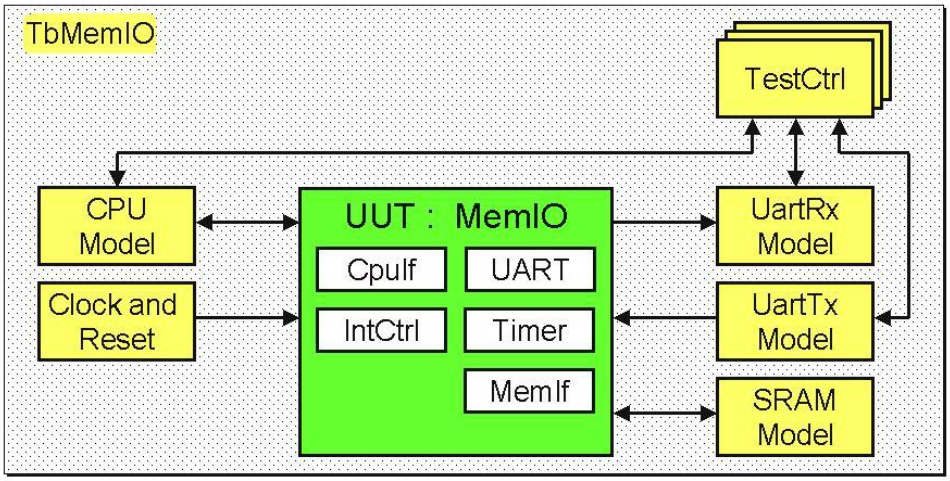

Figure 4. System-Level, Transaction-Based Testbench Block Diagram

2.3 Reducing Test Time

The goal in this paper is to minimize time spent validating the design (accelerate verification) without increasing risk of errors. This means that once we pick a set of tests that will cover the design, we cannot choose to skip some just to complete testing on time. Instead, to reduce test time, inefficiencies of the "Traditional" approach need to be identified and removed.

An inefficiency of the "Traditional" test approach is that it requires writing subblock testbenches that are later abandoned. Another potential inefficiency of subblock testbenches is the failure to share common elements. In addition, since subblock tests must be re-validated at the system-level, there can be a significant amount of time spent on recreating (or porting) these tests to the system-level. If these inefficiencies can be eliminated, we can reduce the time required to validate the design.

2.4 Acceleration Proposal: System Only Test

The "System Only Test" approach proposes to skip the subblock testbenches, integrate all subblocks together, and test only at the system level. This addresses the inefficiency of abandoning the subblock tests and recreating the subblock tests at the system-level. At first, this seems to be a good solution since each subblock test is re-validated at the system level anyway.

The hazard with approach is that it results in many designs are being debugged simultaneously. When a bug is encountered, increased time may be spent to isolate the error to a particular subblock. In addition, since all of the designs are loaded during testing, simulations will take longer to run. This approach could potentially increase the time taken to validate a design. As a result, it is not worth the risk.

2.5 Acceleration Proposal: Pre-Use the System Testbench

The "Pre-Use" approach proposes to use the system-level, transaction-based testbench to test subblocks. The key difference between this approach and the "System Only Test" approach is that subblocks and system level testbench components are incrementally added to the testbench. Only subblocks and system models needed for a test are included in the simulation.

 Copyright © SynthWorks Design Inc

Similar to the "Traditional Approach", only one subblock is tested at a time. However, since the same testbench framework is used for both subblock tests and system tests, custom subblock testbenches are not required. Reuse of common stimulus elements is implicit in the approach. In addition, only a minimal amount of work is required to make subblock test work at the system level.

Unlike the "System Only Test" approach, the "Pre-Use" approach does not incur increased debug time since only one subblock is being tested at a time. As a result, the "Pre-Use" approach should result in less verification time than the "System Only Test" approach. In addition, since the "Pre-Use" approach removes some of the inefficiencies of the "Traditional" approach, the "Pre-Use" approach will reduce the amount of time spent verifying a design.

2.5.1 Pre-Use: Step 1, Plan Tests First

A common problem with the "Traditional" approach is that a design specification is written, subblocks assigned to members of the design team, and subblock design and test begins. Then when all of the subblocks are available, they are integrated and system level tests are planned and written.

The key to the "Pre-Use" approach is to reverse this process. System-level tests must be planned up front at the same time that the design is being planned. The subblock test flow is incremental. Subblocks that are needed to test other subblocks need to be identified. These subblocks and the models that drive them need to be developed first.

2.5.2 Pre-Use: Step 2, First Subblock Test(s)

For the MemIO design, the CpuIf is needed to access registers in other subblocks in the design. As a result, the CpuIf and CpuModel need to be coded and tested first. At this point the testbench will have the components shown in Figure 5. Note if we were using the "Traditional" test approach, all of these pieces of this testbench would be required. As a result, this testbench does not require significantly more work than the disposable subblock testbenches of the "Traditional" approach. Only good planning is required.

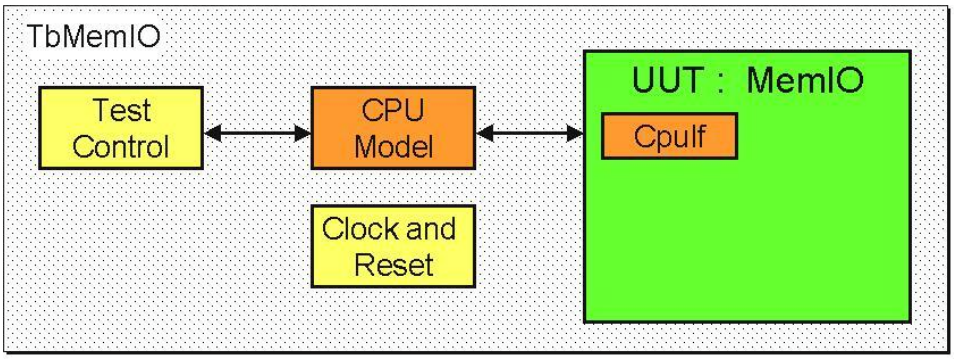

Figure 5. Testing CpuIf Subblock with System-Level Testbench Components

The goal of this first test is to validate that CpuIf handles its portion of the register IO access correctly. This will be accomplished by writing and reading one register of each internal block. To validate this during the subblock tests, visual checking will be required. To validate this test during system level tests the values read will be checked against the value written.

 Copyright © SynthWorks Design Inc

2.5.3 Pre-Use: Concurrent Testing

For the MemIO design, CpuIf acts as a gateway to the registers in other subblocks in the design. Once CpuIf is tested, other subblocks can be independently tested using CpuModel and CpuIf. Conceptual block diagrams for testing the Timer, UART, and MemIf are shown in Figure 6, Figure 7, and Figure 8 respectively. Note that the ClockReset block is not shown. Note that testing of the UART requires that both the UartTxBfm and UartRxBfm models and testing of MemIf requires the SramModel.

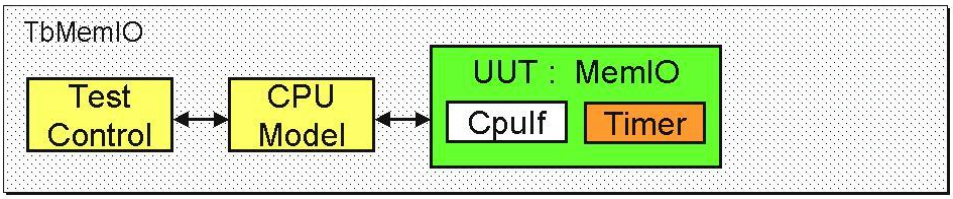

Figure 6. Testing Timer with System-Level Testbench Components

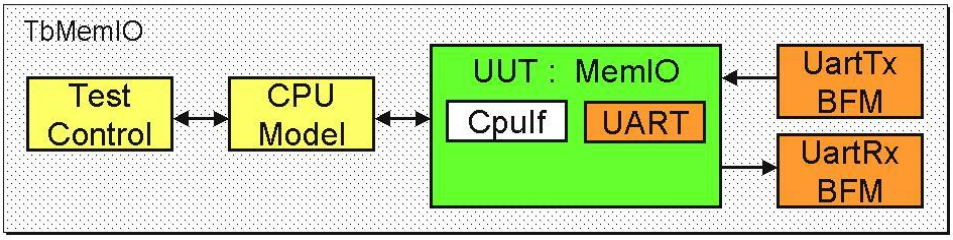

Figure 7. Testing UART with System-Level Testbench Components

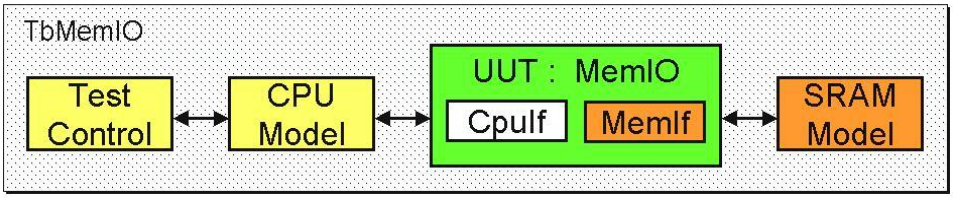

Figure 8. Testing MemIf with System-Level Testbench Components

2.5.4 Pre-Use: Subblock Tests are System Tests

The "Pre-Use" approach tests subblocks at the system level. The best result of this approach is that once all of the subblocks are integrated into the testbench, the tests become full system level tests. The only cost of this approach is that test planning needs to be done up front (as it should be anyway).

2.6 Transaction-Based Testbenches

A transaction-based testbench codes a test in terms of interface actions rather than individual signal transitions. The difference can best be illustrated with an example. Figure 9 shows a CPU write operation coded using individual signal transitions. Figure 10 shows the same operation coded in as a transaction.

 Copyright © SynthWorks Design Inc

```
CpuProc : process
begin
  . . .
  nAds <= '0' after tpd, '1' after tperiod + tpd;
  Addr <= UART_DIVISOR_HIGH after tpd ;
  Data <= X"0000" after tperiod + tpd ;
  Read <= '0' after tpd;
  wait on Clk until nRdy = '0' and Clk = '1' ;
  . . .
```

Figure 9. CPU Write Operation Using Individual Signal Transitions

```
CpuProc : process
begin
  . . .
  CpuWrite(CpuRec, UART_DIVISOR_HIGH, X"0000");
  . . .
```

Figure 10. CPU Write Operation Using Transactions

In a transaction-based testbench, a subprogram call replaces individual signal transitions. This allows a test writer to focus on testing a design rather than the using particular language features. This also decreases the level of difficulty to write a test and understand the intent of a test. As a result, the tests are easier to review and, in a crunch, system and software engineers could potentially write some of the tests without having much HDL coding experience.

In a transaction-based test, all interface signaling is handled by either a subprogram or a separate model. This de-couples the detailed signaling from the testbench. If a design change or implementation error is found in the signaling, it only needs to be changed in one place. If signals were driven directly in the testbench, the number of changes would be significant.

Using BFMs to implement the interface signaling increases the flexibility of the approach. If the CPU were changed from an X86 family to the 68K family, only the CpuModel and CpuIf would change. The transactions and the tests would not need to change.

2.7 Transaction Tests and Subblock Tests

The flexibility gained by a transaction-based testbenches is important for subblock testing. Consider what the impact would be if the CPU had not been selected, but the design needed to be started. In this case, the CpuModel and CpuIf could be replaced by a CpuIfModel as shown in Figure 11. This way, the design and testing can continue in a normal fashion until the CPU has been selected. Once the CPU is selected, CpuModel and CpuIf can replace the CpuIfModel. This is not a preferred way for the design to progress since extra work is required to write CpuIfModel, but it does permit a smooth transition when a key interface is not fully specified at the start of a design.

 Copyright © SynthWorks Design Inc

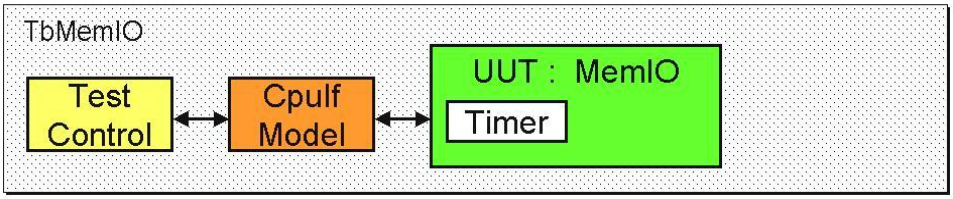

Figure 11. System-Level Testbench when CPU not Available

3. Testbench Details

So far we have been looking at the high level approach. It is time to look at the details and see how it is coded in VHDL. Top level of the testbench can be visualized as shown in Figure 12. The testbench, TbMemIO, consists of a reference to MemIO (UUT), system models (ClockReset, CpuModel, UartTxBfm, UartRxBfm, and SramModel), and a transaction controller (TestCtrl).

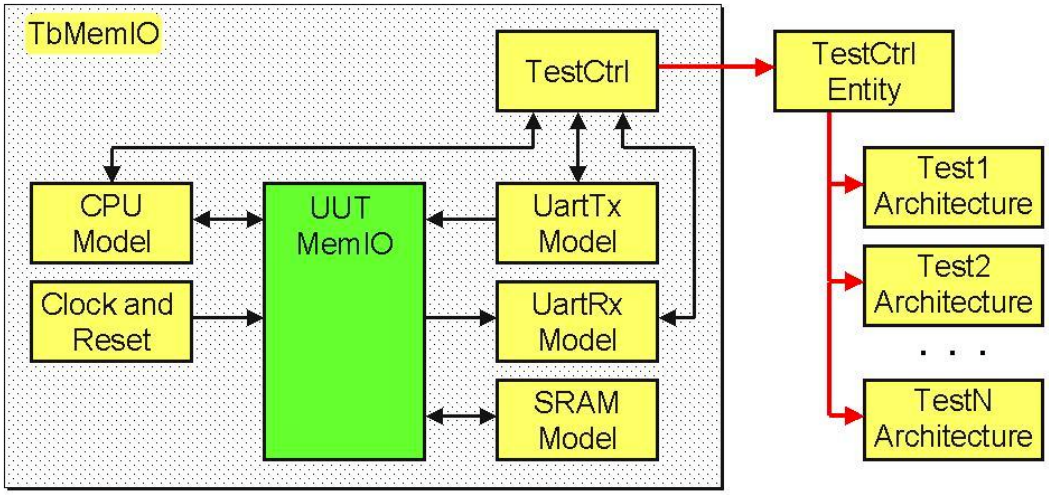

Figure 12. Detailed System-Level, Transaction-Based Testbench Block Diagram

The ClockReset block contains the clock and reset logic. Having this logic in a separate block ensures that clock and reset are set up the same for every simulation. TestCtrl contains transactions to sequence the bus functional models: CpuModel, UartTxBfm, and UartRxBfm. SramModel is a fully functional model of a static RAM. Each test in the test suite is a separate architecture of TestCtrl.

3.1 TbMemIO: Top Level of Testbench

TbMemIO connects MemIO, the unit under test, to the testbench components. As a result it is a structural netlist. This is shown in Figure 13.

 Copyright © SynthWorks Design Inc

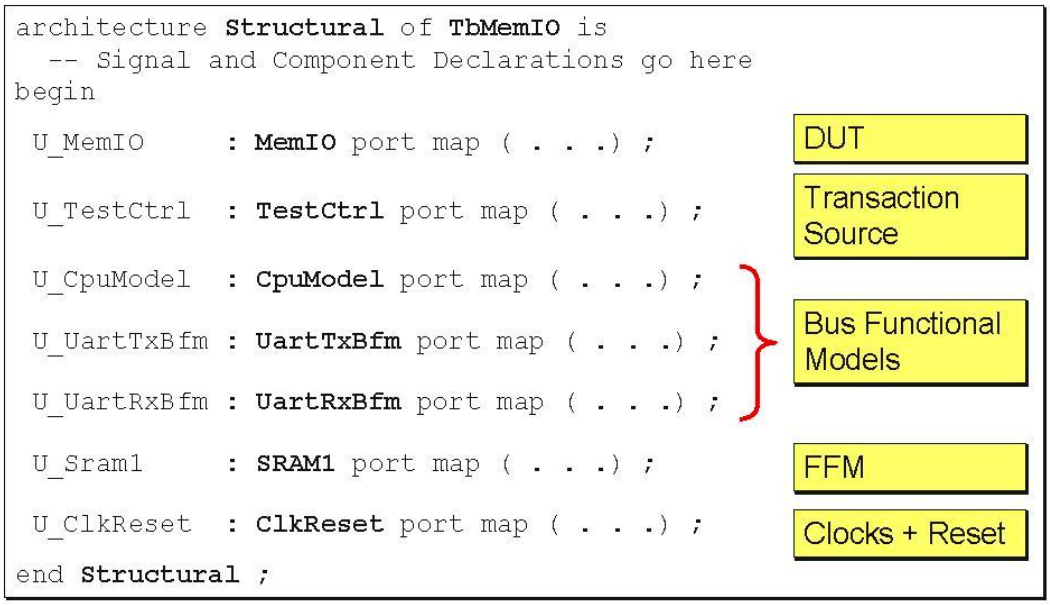

Figure 13. Code for TbMemIO

3.2 TestCtrl

TestCtrl contains transactions to sequence and interact with each bus functional model (BFM). TestCtrl connects to each BFM using one or more record objects. The entity interface for TestCtrl is shown in Figure 14. TestCtrl uses one or more record to interface with each BFM.

```
entity TestCtrl is
  generic (
    tperiod_Clk         : time := 10 ns ;
    CPU_STATUS_MSG_ON   : std_logic := CPUTB_STATUS_MSG_OFF
  ) ;
  port (
    Clk           : In    std_logic ;
    nReset        : In    std_logic ;

    UartTxRec     : InOut UartTbRecType := InitTbUartTbRec ;
    UartRxRec     : InOut UartTbRecType := InitTbUartTbRec ;

    CpuRec        : InOut CpuRecType := InitTbCpuRec ;
    IntRec        : InOut CpuRecType := InitTbCpuRec
  ) ;
end TestCtrl ;
```

Figure 14. TestCtrl Entity

Each test in the test suite is implemented with a separate architecture of TestCtrl. Figure 15 shows an architecture of TestCtrl that sends UART data to the UUT. In the architecture, each independent source of stimulus is supported by one or more processes. For each interface, the stimulus is generated by one or more procedure calls.

 Copyright © SynthWorks Design Inc

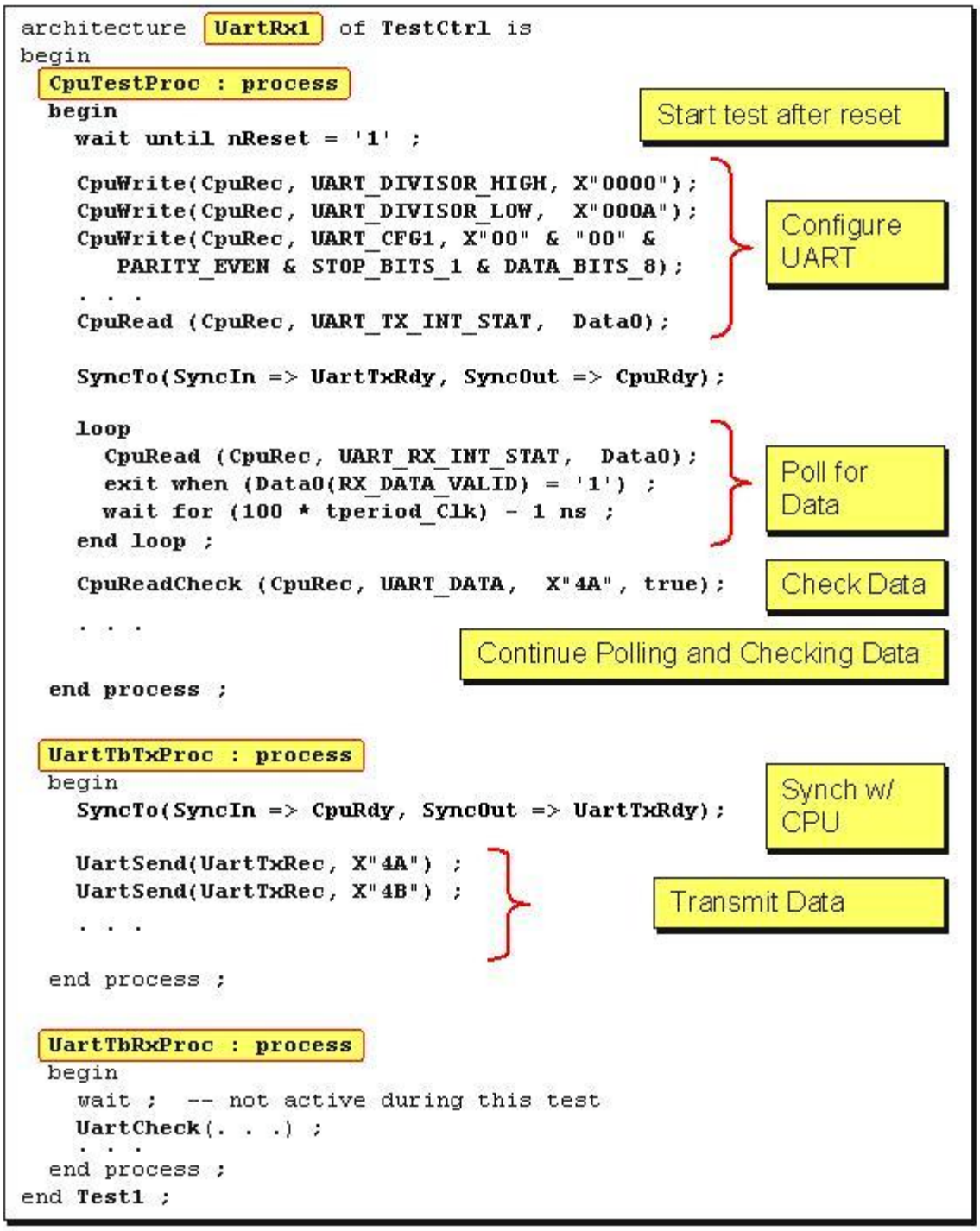

Figure 15. TestCtrl Architecture to test UUT's UART receiver function

3.3 UartTbTxRec

One or more record objects are used as an abstract connection between TestCtrl and a BFM. These records contain handshaking and transaction information. Figure 16 shows the implementation of UartTbRecType. Both TestCtrl and UartTxBfm both drive UartTbTxRec as an InOut. To simplify resolving contention, all types in UartTbRecType are based on std_logic. Note that both TestCtrl (see Figure 14) and UartTxBfm initialize the UartTbTxRec port to avoid contention. The InitTbUartTbRec

 Copyright © SynthWorks Design Inc

is defined as shown in Figure 17. Note that the values shown in bold are driven by TestCtrl. All fields not driven by TestCtrl are initialized to 'Z' to avoid contention.

```
type UartTbRecType is record
  CmdRdy          : std_logic ;
  CmdAck          : std_logic ;

  Data            : std_logic_vector (7 downto 0);
  StatusMode      : unsigned ( 3 downto 0) ;
  TbErrCnt        : unsigned (15 downto 0) ;
  UartBaudPeriod  : unsigned (31 downto 0) ;
  NumDataBits     : unsigned ( 2 downto 0) ;
  ParityMode      : unsigned ( 2 downto 0) ;
  NumStopBits     : std_logic ;
end record ;
```

Control / Handshaking

Data Fields

Figure 16. UartTbRecType Definition

```
constant InitTbUartTbRec : UartTbRecType := (
  CmdRdy           => '0',
  CmdAck           => 'Z',
  Data             => (others => 'Z'),
  StatusMode       => (others => 'Z'),
  TbErrCnt         => (others => '0'),
  UartBaudPeriod   => to_unsigned(. . .),
  NumDataBits      => UARTTB_DATA_BITS_8,
  ParityMode       => UARTTB_PARITY_EVEN,
  NumStopBits      => UARTTB_STOP_BITS_1
) ;
```

Figure 17. Constant InitTbUartTbRec Definition

3.4 Procedure UartSend

The purpose of the procedures is to coordinate handing off transactions to the BFMs. There is not a lot of magic in the procedures, just tedious record manipulations and handshaking. The UartSend procedure is shown in Figure 18. UartSend fills in fields in the record and then handshakes with UartTxBfm using the RequestAction procedure. After the BFM finishes the transaction, UartSend inserts idle time to allow spacing between consecutive transfers.

 Copyright © SynthWorks Design Inc

```
procedure UartSend (
  signal UartRec   : inout UartTbRecType ;
         Data      : in  std_logic_vector (7 downto 0) ;
         IdleTime  : in  time := 0 ns ;
         ErrorMode : in  UartTb_StatusModeType := UARTTB_NO_ERROR
) is
begin
  -- Put Transaction into the Record
  UartRec.Data        <= Data ;
  UartRec.StatusMode  <= ErrorMode ;

  -- Handshake with UartTxBfm
  RequestAction(Rdy => UartRec.CmdRdy, Ack => UartRec.CmdAck);

  -- Insert idle time between transactions
  if (IdleTime > 0 ns) then
    wait for IdleTime ;
  end if ;

end UartSend ;
```

Figure 18. UartSend Procedure

UartSend is simple since the BFM does not return any values. In general, a procedure will put the transaction into the record, print status information, handshake with the BFM, and check results. Details of the RequestAction procedure are shown in section 3.7 Handshaking Details. Note that other models may require different handshaking methods.

3.5 Package UartTbPkg

All constants, types, subtypes, and procedures that support UartTxBfm and UartRxBfm get stored in the package UartTbPkg. Excerpts from UartTbPkg are shown in Figure 19.

 Copyright © SynthWorks Design Inc

```
library ieee ;
  use ieee.std_logic_1164.all ;

package UartTbPkg is
  constant UART_BAUD_PERIOD_250K  . . .;
  type UartTbRecType is record . . . ;
  constant InitTbUartTbRec : . . . ;
  . . .
  procedure UartSend (. . .) ;
  . . .
end UartTbPkg ;
```

Declare Types, Constants and Subprograms

```
package body UartTbPkg is
  procedure UartSend (. . .) is
  begin
    . . .
  end procedure ;
  . . .
end UartTbPkg ;
```

Implement Subprograms

Figure 19. Exerpts from UartTbPkg

3.6 UartTxBfm

The purpose of UartTxBfm is to convert transactions shown in process UartTbTxProc of Figure 15 into the waveforms shown in Figure 20. The code for UartTxBfm is shown in Figure 21. UartTxBfm contains a concurrent statement to create the internal UART clock and the process UartTxFunction to create the UART functionality. UartClk supports the functionality of in UartTxFunction. The process UartTxFunction waits for a transaction to arrive from the testbench, creates the required interface signaling for the transaction, and then signals that it has completed the transaction. Other models often have additional code that puts return values in the record before signaling the end of the transaction. Note that the WaitForRequest procedure handles all of the handshaking with the testbench. First it signals that a transaction is done and then it suspends until the next transaction arrives.

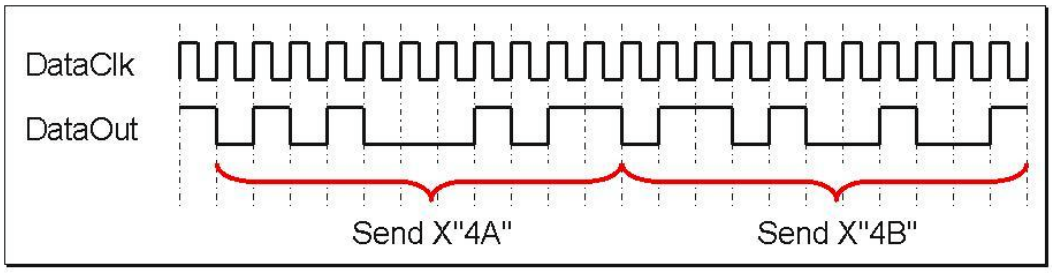

Figure 20. Waveforms Created by UartTxBfm

 Copyright © SynthWorks Design Inc

```
entity UartTxBfm is
  port (. . .) ;
end UartTxBfm ;
architecture Model of UartTxBfm is
  -- declarations not shown
begin

  -- Create UART Clock
  UartClk <=  . . . ;
  . . .

  -- Implement Model Functionality
  UartTxFunction : process
    -- declarations not shown
  begin

    -- Signal end of Transaction and
    -- Wait For next Transaction
    WaitForRequest( . . .) ;

    -- Send Start Bit
    wait until UartClk = '1' ;
    DataOut <= '0' ;

    -- Send Data Bits
    for i in 0 to 7 loop
      wait until UartClk = '1' ;
      DataOut <= TxData(i) ;
    end loop ;

    -- Send Parity Bit
    wait until UartClk = '1' ;
    DataOut <= TxParity ;

    -- Send Stop Bit
    wait until UartClk = '1' ;
    DataOut <= '1' ;

  end process ;

end Model ;
```

Figure 21. UartTxBfm Code

In general the a BFM contains code to process inputs, create internal resources, model the functionality, check for interface protocol errors, and check for setup and hold errors. UartTxBfm omits much of this code since it does not have any inputs. Input processing logic resolves all input values to '0', '1', and 'X'. This facilitates checking for a particular logic value. Protocol check logic looks for interface errors. The SramModel contains protocol checkers to look for a pulse on the write signal during a read operation. The CpuModel contains protocol checkers to look for the done strobe (nRdy) being active when the CPU is idle. Setup and Hold checkers are used to check for timing violations. These checkers are only used in the gate-level simulations. The author often skips adding setup and hold checkers when there are not any exceptions (ignored paths) in the static timing analysis.

 Copyright © SynthWorks Design Inc

3.7 Handshaking Details

Transaction handoff between TestCtrl and the BFMs is handled by handshaking through the CmdRdy and CmdAck fields of the record. This handshaking is required to synchronize otherwise independently executing models (due to the concurrent nature of the language). The flow of handshaking is shown in Figure 22. At the start of time, UartTxBfm executes WaitForRequest and stops until RequestAction is called by TestCtrl (indirectly). At the start of time, TestCtrl process UartTbTxProc stops until SyncTo wakes up. SyncTo does not wake up until the CpuTestProc has initialized the CPU side of the UART. Next, TestCtrl does it first call of UartSend. UartSend puts the transaction in the record and calls RequestAction. RequestAction causes UartTxBfm's call to WaitForRequest to resume and perform the requested bus signaling. When UartTxBfm encounters WaitForRequest a second time, it causes the call to RequestAction in UartSend to resume and then UartTxBfm suspends until the next call to RequestAction.

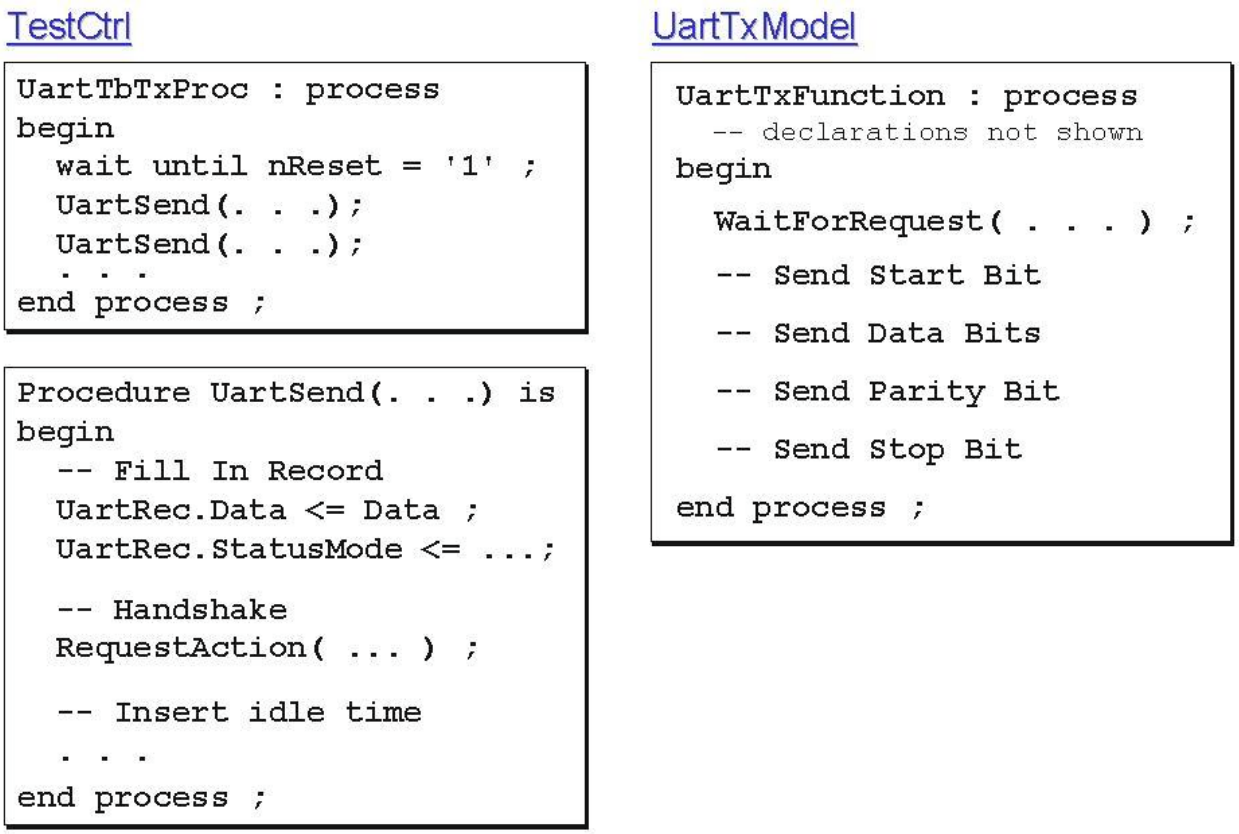

Figure 22. Handshaking flow TestCtrl/UartSend/RequestAction and UartTxBfm/WaitForRequest

The signaling for CmdRdy and CmdAck is shown in Figure 23. The protocol is entirely arbitrary and there are other methods that will also work just as well. The procedure RequestAction, shown in Figure 24, handles the CmdRdy signaling. The procedure WaitForRequest, shown in Figure 25, handles the CmdAck signaling.

 Copyright © SynthWorks Design Inc

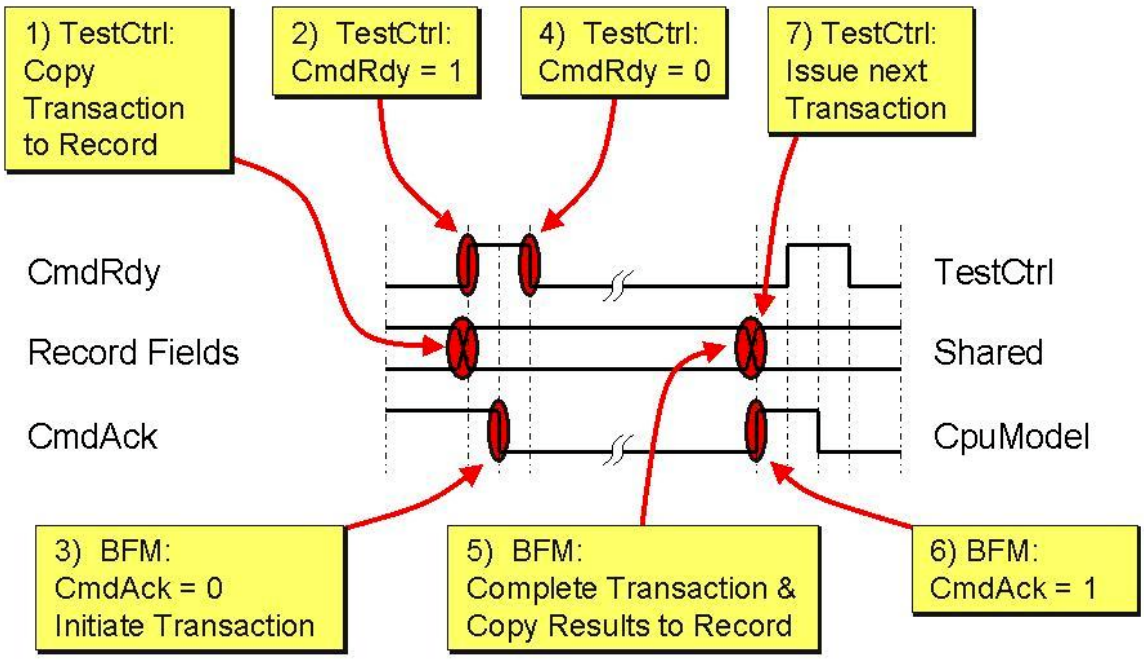

Figure 23. Signaling for CmdRdy and CmdAck

```
procedure RequestAction (
  signal Rdy  : Out std_logic ;
  signal Ack  : In  std_logic
) is
begin
  -- Record contains new transaction
  Rdy        <= '1' ;

  -- Find Ack low = '0'
  if Ack /= '0' then
    wait until Ack = '0' ;
  end if ;

  -- Prepare for Next Transaction
  Rdy        <= '0' ;

  -- Transaction Done
  wait until Ack = '1' ;
end procedure ;
```

Figure 24. RequestAction Procedure

 Copyright © SynthWorks Design Inc

```
procedure WaitForRequest (
  signal Clk  : In  std_logic ;
  signal Rdy  : In  std_logic ;
  signal Ack  : Out std_logic
) is
begin
  -- Prepare for handshaking
  Ack        <= '1' ;

  -- Allow Ack and Rdy to settle
  wait for 0 ns ;  -- Ack Valid, Set Rdy
  wait for 0 ns ;  -- Rdy now valid

  -- Find Rdy high at a bus cycle boundary
  if Rdy /= '1' then
    wait until Rdy = '1' ;
    wait until Clk = '1' ;
  end if ;

  -- Model active and owns the record
  Ack        <= '0' ;
end procedure ;
```

Figure 25. WaitForRequest Procedure

Note in the procedure WaitForRequest that there are two "wait for 0 ns ;" statements. "Wait for 0 ns ;" causes the process to stop for exactly one delta cycle. The first wait allows RequestAction to see CmdAck rise and resume. At this point in time, TestCtrl can make a second call to UartSend which would in turn call RequestAction. The second wait allows CmdRdy to be updated with the new value set by RequestAction to be valid when checking for CmdRdy to be high. For some models it would be possible to replace both of these wait statements with a single "Wait for 1 ns ;". However, this would cause the relationship with clock to drift when scheduling multiple back to back UartSend (or more importantly CpuWrite/CpuRead) cycles. If the starting point with respect to clock drifts, so will the time at which value are driven on signals.

4. Conclusions

For the up front cost of planning tests at the start of a design, custom subblock testbenches can be replaced by using the system level testbench and incrementally adding subblocks and corresponding BFMs. This will reduce the verification cycle and get the product done sooner.

5. References

1. Jim Lewis, "VHDL Testbenches and Verification", student manual for SynthWorks class. For more information see http://www.SynthWorks.com/VhdlTest.htm.

 Copyright © SynthWorks Design Inc

About SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
Engineers learn VHDL Syntax plus basic RTL coding styles and simple procedure-based, transaction testbenches. Our designer focus ensures that your engineers will be productive in a VHDL design environment.

VHDL Coding Styles for Synthesis 4 Days
http://www.synthworks.com/vhdl_rtl_synthesis.htm
Engineers learn RTL (hardware) coding styles that produce better, faster, and smaller logic.

VHDL Testbenches and Verification 3 days
http://www.synthworks.com/vhdl_testbench_verification.htm
Engineers learn how create a transaction-based verification environment based on bus functional models.

For additional courses see: http://www.synthworks.com