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

by
Jim Lewis
Director of Training, SynthWorks Design Inc
Jim@SynthWorks.com

Author Biography

SynthWorks

Jim Lewis, Director of Training, SynthWorks Design Inc.

Jim Lewis, the founder of SynthWorks, has seventeen 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). Mr. Lewis can be reached at jim@SynthWorks.com, 1-503-590-4787, or www.SynthWorks.com

Pre-Use of System Testbenches

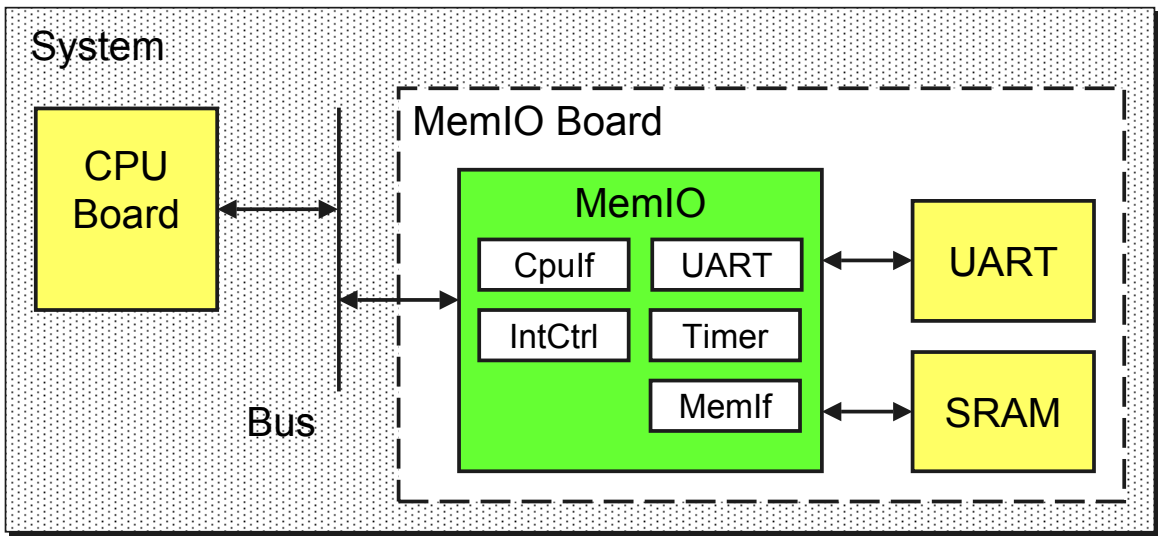
SynthWorks

- Approach
  - Basics
  - Traditional Approach
  - System Only Test Approach
  - Pre-Use Approach
  - Transaction Based Testing
- Details
  - Step by Step Overview of the Testbench Pieces

Getting the Slides: http://www.SynthWorks.com

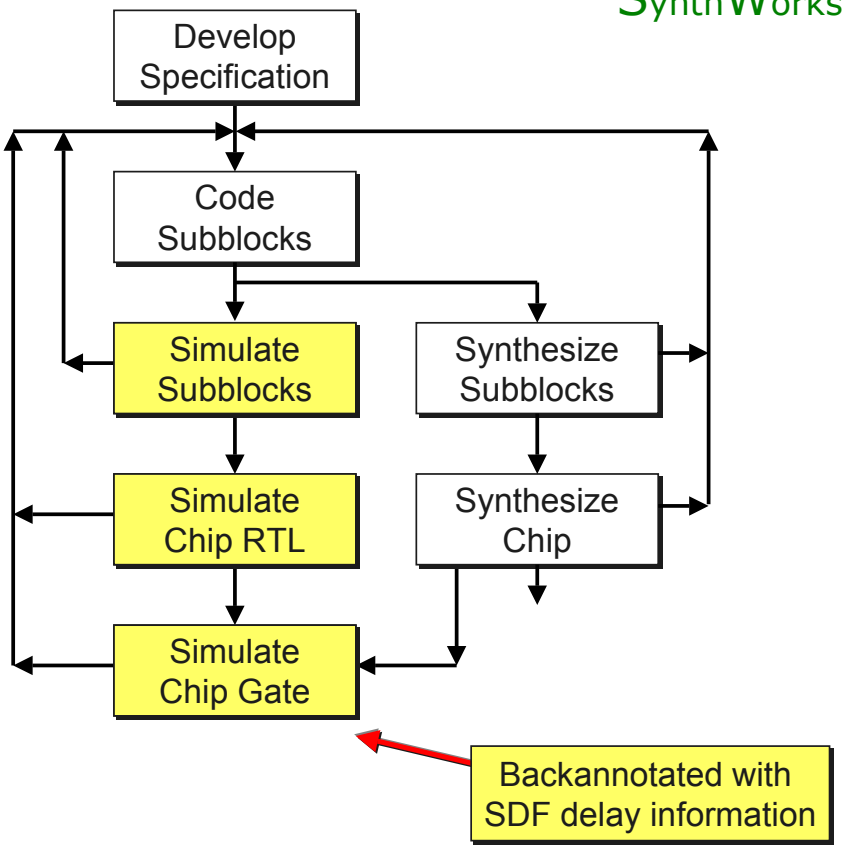
Design Under Test = MemIO

SynthWorks



Design Flow

SynthWorks



Testbench =

SynthWorks

Code structure that allows waveforms to driven to the unit under test and validates the results (visually or automatically)

Testbench =
Entity + Architecture(s)

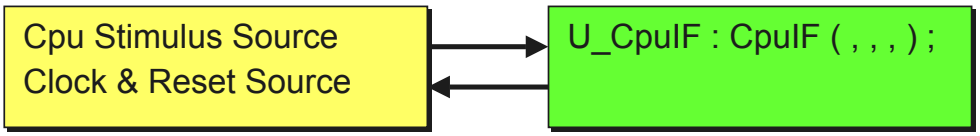
- Unit Under Test (UUT)
  AKA DUT
  Instantiation
- Stimulus Source
  one or more processes
  or instantiations

```
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   . . .
end tb ;
```

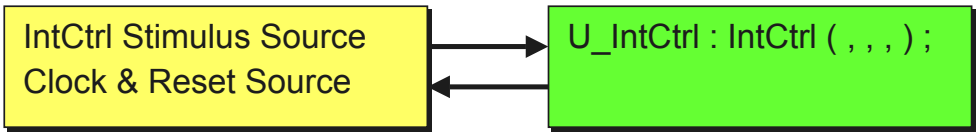
Traditional Approach: Subblock

SynthWorks

- Write a separate, custom testbench for each subblock.
  - Test all functionality in that subblock
- Testing CpuIF Subblock



- Testing IntCtrl Subblock

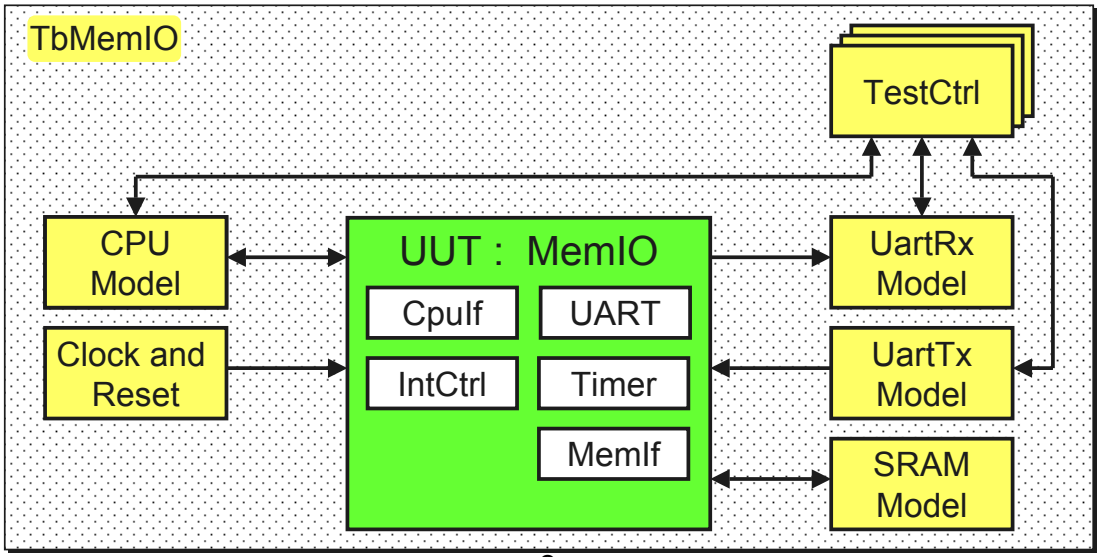


- Test all other subblocks with separate, custom testbenches

Traditional Approach: System

SynthWorks

- Immerse the chip into a system environment
- Each interface in the system = one model (BFM and/or FFM)
- Test by running multiple test scenarios
- Re-validate each subblock in the system environment



Accelerating Verification

SynthWorks

- Goal:
  - Minimize test time without reducing test coverage
- Observations:
  - Traditional subblock testbenches are used at the beginning of testing and then abandoned.
  - Subblock tests are re-validated at the system level
- If we can minimize the abandoned and duplicated work, we can accelerate our verification effort.

Proposal: System Only Tests

SynthWorks

- No custom subblock testbenches
- Integrate all designs, and test at system level
- Hazard:
  - Many designs being simultaneously debugged.
  - When a bug is encountered, increased time may be spent to isolate the error to a particular subblock.
  - May have to fix the current bug before finding next bug.
  - Increased time will be spent to run the subblock simulations since all subblocks in the design are loaded.
- Conclusion:
  - Not worth the risk. May actually increase time.

Proposal: Pre-Use the System Testbench

- No custom subblock testbenches
- Use System Level Testbench for all testing
- Incrementally add and test subblocks
- Incrementally add system interface models

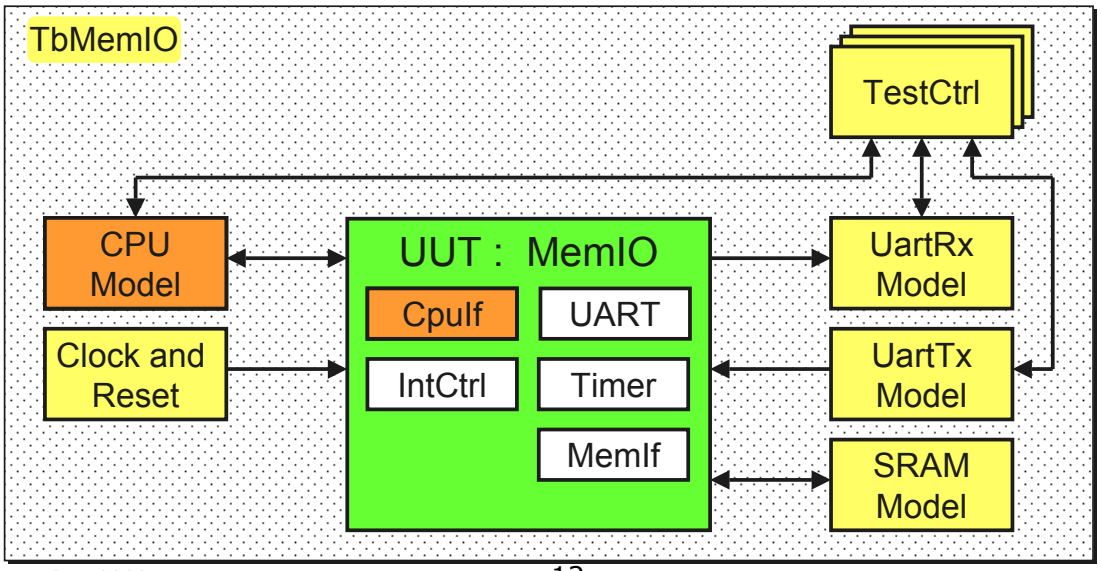
- Benefit
  - One subblock being tested at a time
  - Not writing subblock testbenches that get abandoned later
  - No need to port subblock code to system level

- Conclusion:
  - No additional risk since only testing one block at a time
  - Speed up due to skipping custom subblock testbenches

Pre-Use the System Testbench

SynthWorks

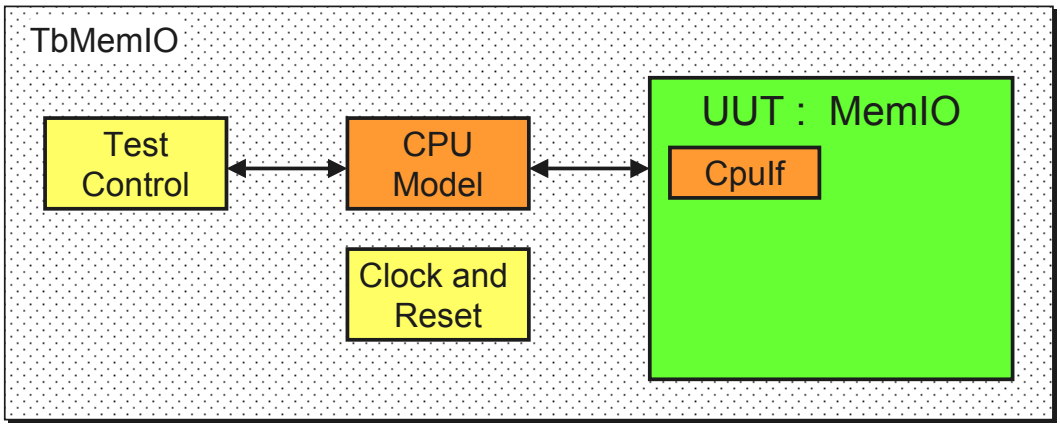
- Step 1: Plan the tests first (Test Plan)
  - Identify key driving interfaces required to get data into/out of the design (CPU, PCI, …)
  - Plan to test these subblocks and testbench models first



Pre-Use: First Subblock Test

SynthWorks

- Step 2: Code and Test Key Interfaces
  - CpuIf (design) and CpuModel (testbench)
- Testing CpuIf using pieces of system-level testbench:

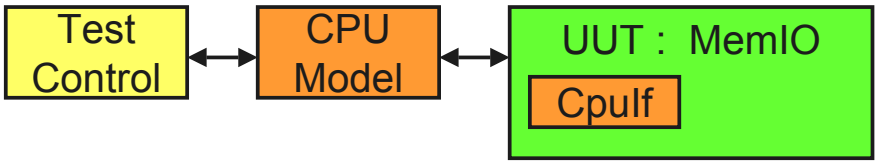


  - Note all of the above functionality would be required in some form in a subblock testbench.

Pre-Use: First Subblock Test

SynthWorks

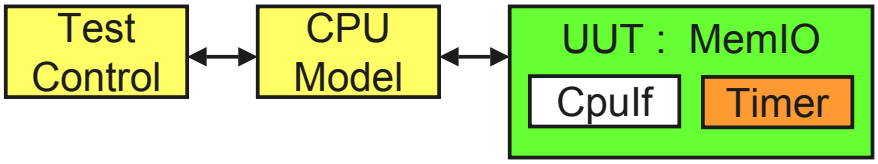
- Step 2, Test 1 Continued: CpuIf + CpuModel



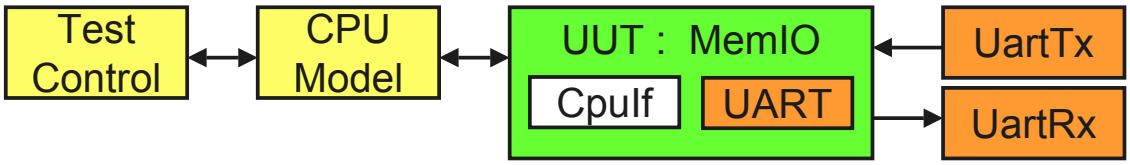
- Test Goal: Gain register IO access to other subblocks in chip
  - Necessary to test other blocks
- Test Method: Write and read one register per internal block.
- Validation Plan
  - Subblock: Visual check.
  - System: Self-Checking. Expect to read back value written.

Pre-Use: Concurrent Subblock Tests*

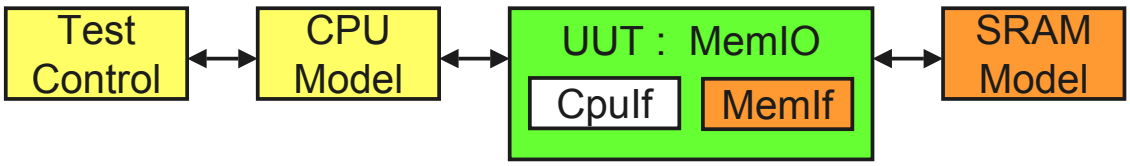
- Test 2A: Test Timer (design) and CpuModel (testbench)



- Test 2B: Test UART (design) and UartBfm (testbench)



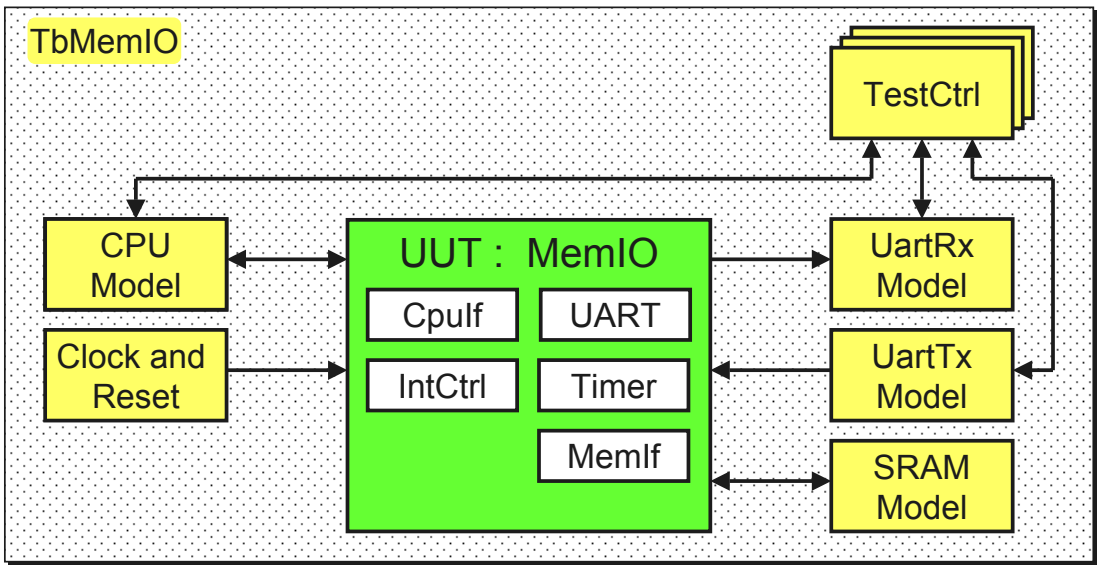
- Test 2C: Test MemIF (design) and SramModel (testbench)



- * Note many tasks can be done by independent design teams.

Pre-Use: Subblock Tests = System Tests

- Once all subblocks are integrated into the design, the testbench becomes a full system test.



- Constraint to approach:
  Order of design and testing must be planned.

Transaction Based Testing

SynthWorks

- A transaction based test programs interface actions.
- Without transaction based testing, wiggle signals:

```
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' ;
  . . .
```

- With transaction based testing, do actions on interfaces:

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

Transaction Based Testing

SynthWorks

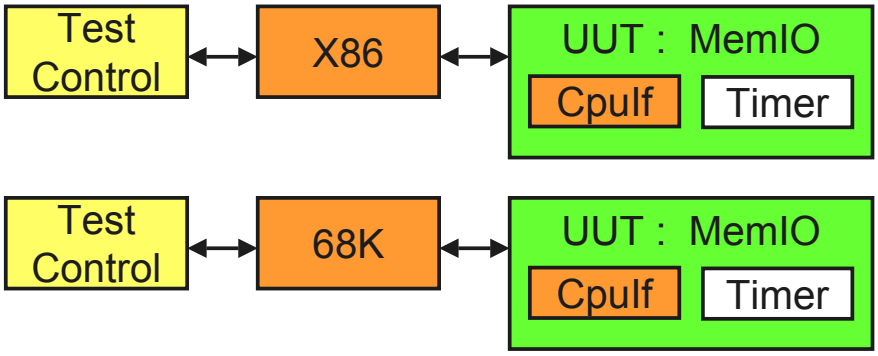
```
CpuWrite(CpuRec, UART_DIVISOR_HIGH, X"0000") ;
CpuWrite(CpuRec, UART_DIVISOR_LOW,  X"000A") ;
. . .
CpuRead (CpuRec, UART_STAT, DataOut) ;
```

- Key Features
  - Program interface actions
  - Procedure call replaces the detailed signaling
  - No longer tied to the detailed signaling
  - Test writer can focus on the tests rather than a HDL/HVL

Transaction Based Testing

SynthWorks

- Flexibility: How does the testbench change if change CPUs?

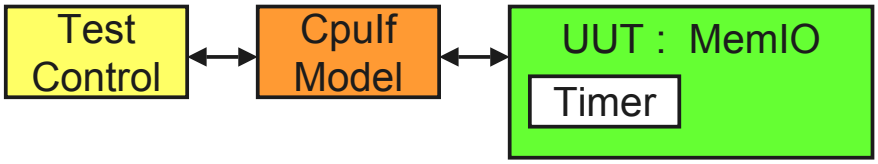


- Only the models change, not the transactions

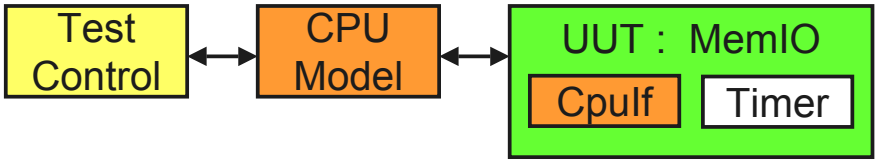
Transactions & Subblocks

SynthWorks

- Flexibility is important for subblock testing.
- What happens if a subblock is unavailable?
  - What if CPU has not been selected?
- Replace CpuIf + CpuModel with CpuIfModel



- Later, when CpuIf available, use CpuModel
  - Note, only the models change, not the transactions.



Summary of Approach

SynthWorks

- Transaction based testbench + planning =
  - Possible to pre-use pieces of the system-level testbench to test subblocks
- Benefit:
  - Amount of development time decreases
  - No longer need to develop subblock testbenches
  - No longer need to port each test case to the system level
    - It automatically runs.

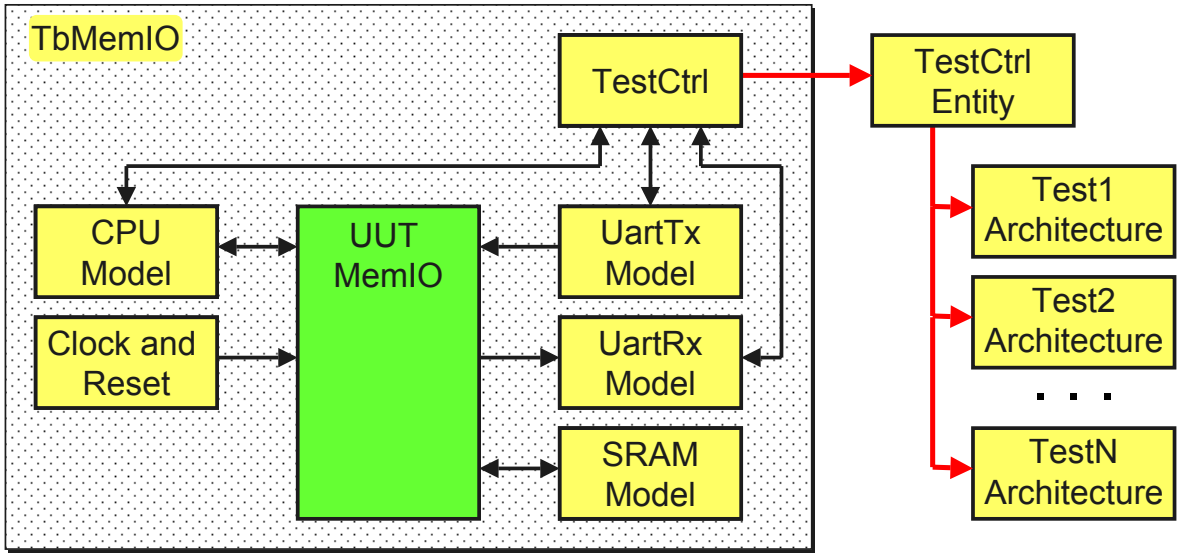
Pre-Use of System Testbenches

SynthWorks

- Approach
  - Basics
  - Traditional Approach
  - System Only Test Approach
  - Pre-Use Approach
  - Transaction Based Testing
- Details
  - Step by Step Overview of the Testbench Pieces

Testbench Structure

SynthWorks



Key Features:

- Bus Functional Models (BFMs) implement interface signaling
- TestCtrl contains transactions to sequence BFMs
- Each test is a separate architecture of TestCtrl.

Testbench Structure

SynthWorks

- TbMemIO = Top level of testbench = Netlist = Test Harness

```
architecture Structural of TbMemIO is
  -- Signal and Component Declarations go here
begin
 U_MemIO      : MemIO port map ( . . .) ;
 U_TestCtrl   : TestCtrl port map ( . . .) ;
 U_CpuModel   : CpuModel port map ( . . .) ;
 U_UartTxBfm  : UartTxBfm port map ( . . .) ;
 U_UartRxBfm  : UartRxBfm port map ( . . .) ;
 U_Sram1      : SRAM1 port map ( . . .) ;
 U_ClkReset   : ClkReset port map ( . . .) ;
end Structural ;
```

DUT

Transaction Source

Bus Functional Models

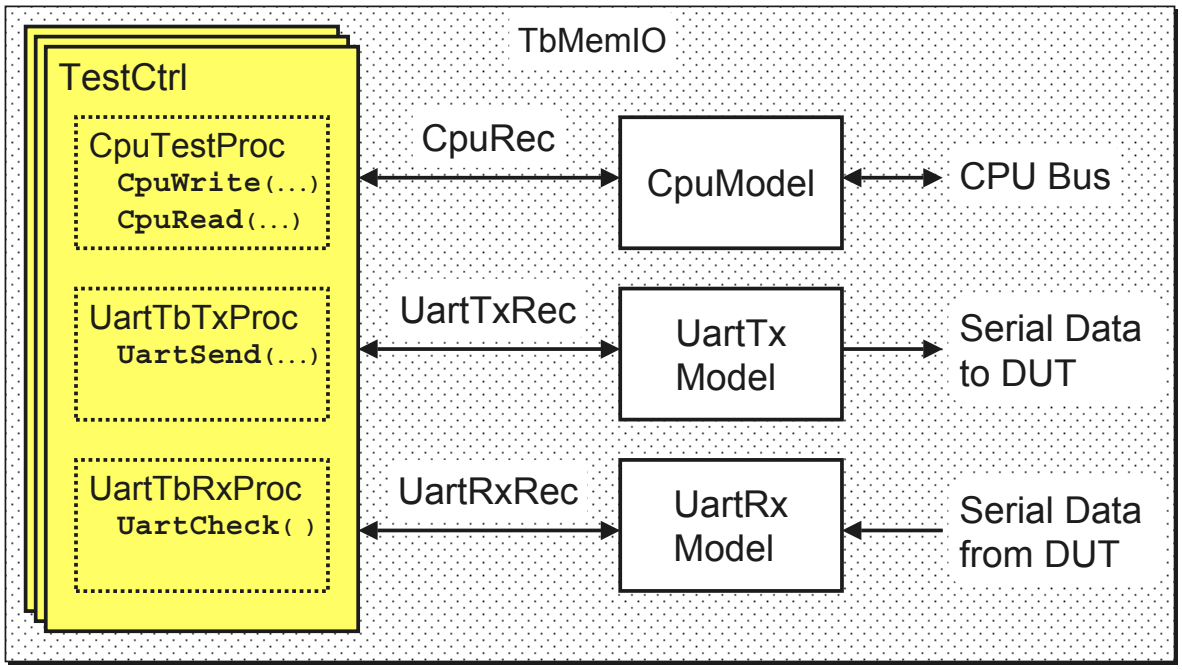
FFM

Clocks + Reset

Testbench Structure: TestCtrl

SynthWorks

- TestCtrl contains transactions to interact/sequence each BFM



TestCtrl Entity

SynthWorks

```
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 ;
```

Recommendation:
Keep TestCtrl entity in a separate file from the architecture(s).
Facilitates using multiple architectures.

TestCtrl Architecture: Big Picture

SynthWorks

```
architecture UartRx1 of TestCtrl is
begin
  CpuTestProc : process
  begin
    wait until nReset = '1';
    CpuWrite(. . .);
    CpuRead(. . .);
    . . .
  end process ;

  UartTbTxProc : process
  begin
    SyncTo(. . .);
    UartSend(. . .) ;
    . . .
  end process ;

  UartTbRxProc : process
  begin
    UartCheck(. . .) ;
    . . .
  end process ;
end Test1 ;
```

One or more processes for each independent source of stimulus

Interface Stimulus is generated with one or more procedure calls

Each test is a separate architecture of TestCtrl (TestCtrl_UartRx1.vhd, TestCtrl_UartRx2.vhd, …)

A test developer only needs to understand TestCtrl and not additional details of the testbench approach

---

CpuTestProc : process -- TestCtrl_UartRx1

```
  -- Declarations left out
begin
  wait until nReset = '1' ;

  CpuWrite(CpuRec, UART_DIVISOR_HIGH, X"0000");
  CpuWrite(CpuRec, UART_DIVISOR_LOW,  X"000A");
  CpuWrite(CpuRec, UART_CFG1, X"00" & "00" &
     PARITY_EVEN & STOP_BITS_1 & DATA_BITS_8);
  . . .
  CpuRead (CpuRec, UART_TX_INT_STAT,  DataO);

  SyncTo(SyncIn => UartTxRdy, SyncOut => CpuRdy);

  loop
    CpuRead (CpuRec, UART_RX_INT_STAT,  DataO);
    exit when (DataO(RX_DATA_VALID) = '1') ;
    wait for (100 * tperiod_Clk) - 1 ns ;
  end loop ;

  CpuReadCheck (CpuRec, UART_DATA,  X"4A", true);

  . . .
end process ;
```

Start test after reset

Configure UART

Synch with UartTbTxProc

Poll for Data

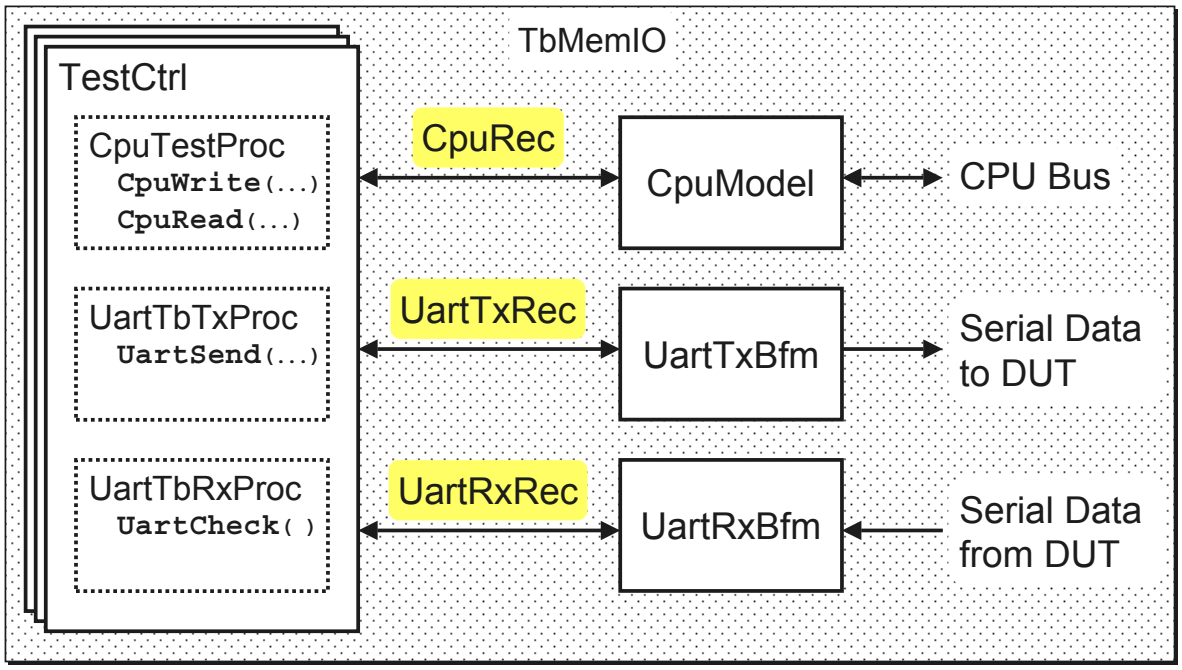
Check Data

Continue Polling and Checking Data

Testbench Structure: UartTxRec

SynthWorks

- Abstract Interface between TestCtrl and the UartTxBfm



Record: UartTbRecType

SynthWorks

```
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

- Issues with records
  - UartTxRec has two drivers (TestCtrl and UartTxBfm)
  - All types are based std_logic to facilitate resolving contention

Initializing UartTxRec

SynthWorks

- Initialize UartTxRec at entity ports to avoid contention:

```
port (
  UartTxRec     : InOut UartTbRecType := InitTbUartTbRec ;
  . . .
) ;
```

Initialization

- Undriven fields are initialized to 'Z' using the following constant:

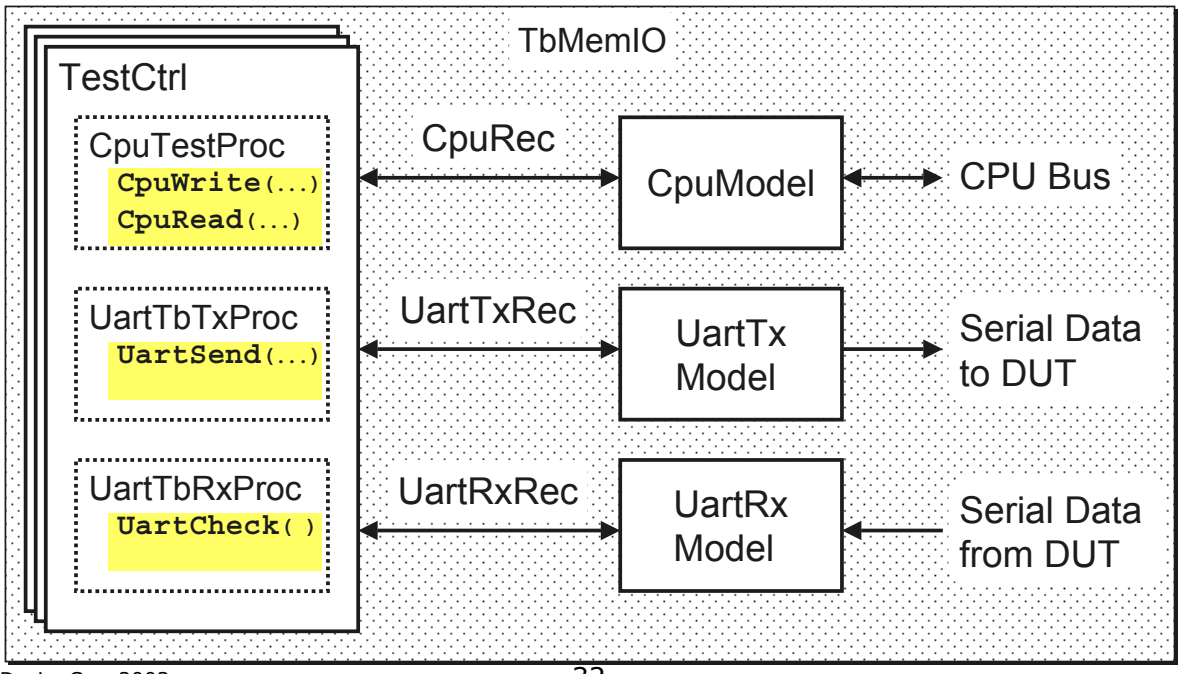
```
constant InitTbUartTbRec : UartTbRecType := (
  CmdRdy           => '0',
  CmdAck           => 'Z',
  Data             => (others => 'Z'),
  StatusMode       => (others => 'Z'),
  TbErrCnt         => (others => '0'),
  UartBaudPeriod   => to_unsigned(. . .),
  NumDataBits      => UARTTB_DATA_BITS_8,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

Fields driven by UartTxBfm are in bold text

Testbench Structure: Procedures

SynthWorks

- Procedures handshake data/sequencing to the BFMs
- Not a lot of magic in the procedures



SynthWorks

```
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 ;
```

Basic Flow
- Put Transaction into Record
- Handshake with Model
- Check Results

Package: UartTbPkg

SynthWorks

- All Constants, Types, and procedures that support UartTxBfm get stored in the package UartTbPkg

```
library ieee ;
  use ieee.std_logic_1164.all ;
package UartTbPkg is
  type UartTbRecType is record . . . ;
  constant InitTbUartTbRec : . . . ;
   . . .
  procedure UartSend (. . . ) ;
  . . .
end UartTbPkg ;
```

Declare Types, Constants and Subprograms

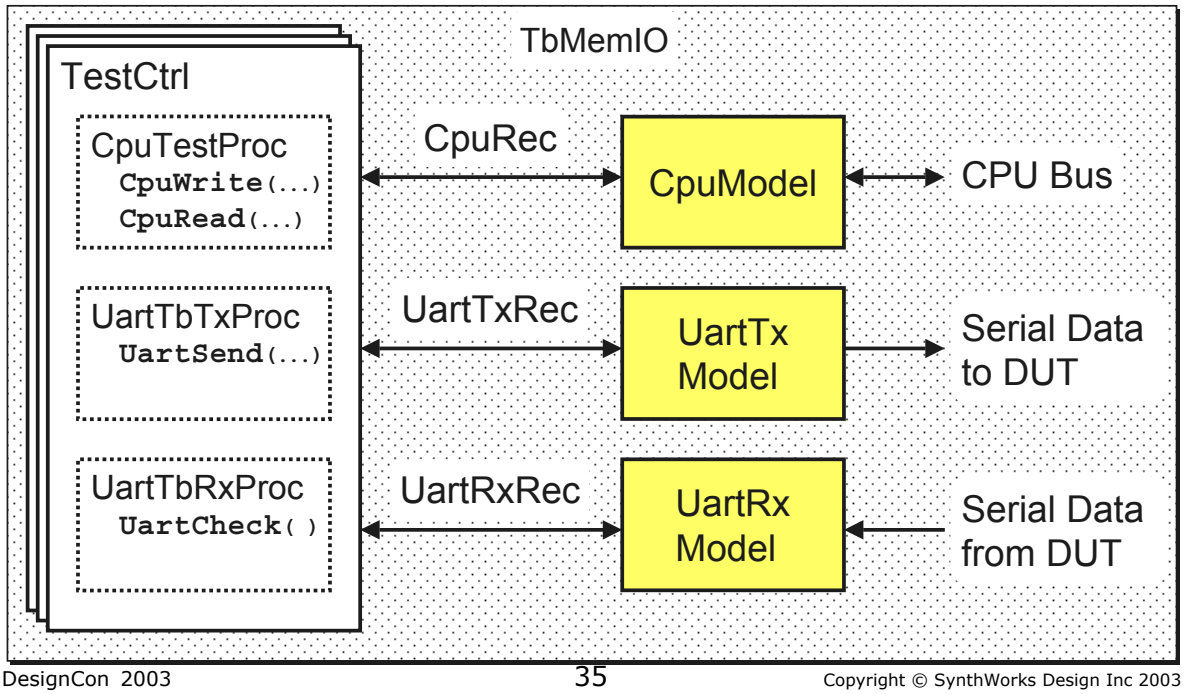
```
package body UartTbPkg is
  procedure UartSend (. . .) is
  begin
    . . .
  end procedure ;
  . . .
end UartTbPkg ;
```

Implement Subprograms

Testbench Structure: Models

SynthWorks

- Perform interface specific signaling
- Sequencing/Data values determined by values in record



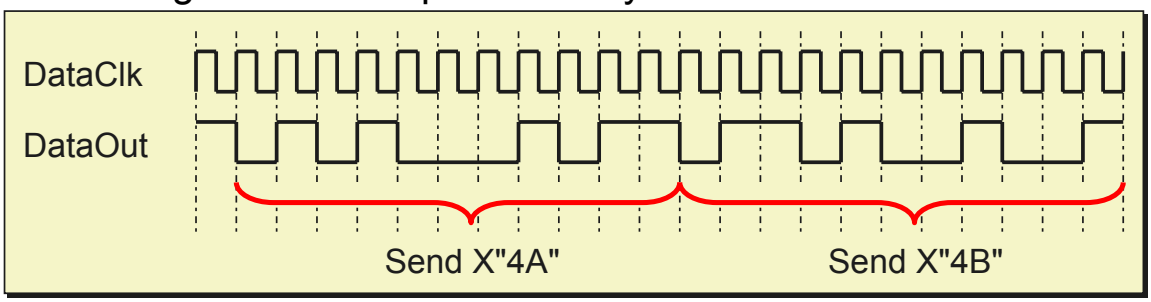
UartTxBfm

SynthWorks

- Models execute transactions requested by TestCtrl
- Transactions in TestCtrl

```
UartSend(UartTxRec,  X"4A") ;
UartSend(UartTxRec,  X"4B") ;
```

- Resulting Waveforms produced by UartTxBfm



entity UartTxBfm is

UartTxBfm: Overview

```
  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
    WaitForRequest( . . .) ;
    -- Send Start Bit
    -- Send Data Bits
    -- Send Parity Bit
    -- Send Stop Bit
  end process ;
end Model ;
```

Basic Elements of a BFM

- Input Processing
- Internal Resources
- Functionality
- Protocol Checks
- Setup and Hold Checks

UartTxFunction : process

UartTxBfm: Details

```
    -- 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 <= UartRec.TxData(i) ;
  end loop ;
  -- Send Parity Bit
  wait until UartClk = '1' ;
  DataOut <= UartRec.TxParity ;
  -- Send Stop Bit
  wait until UartClk = '1' ;
  DataOut <= '1' ;
end process ;
```

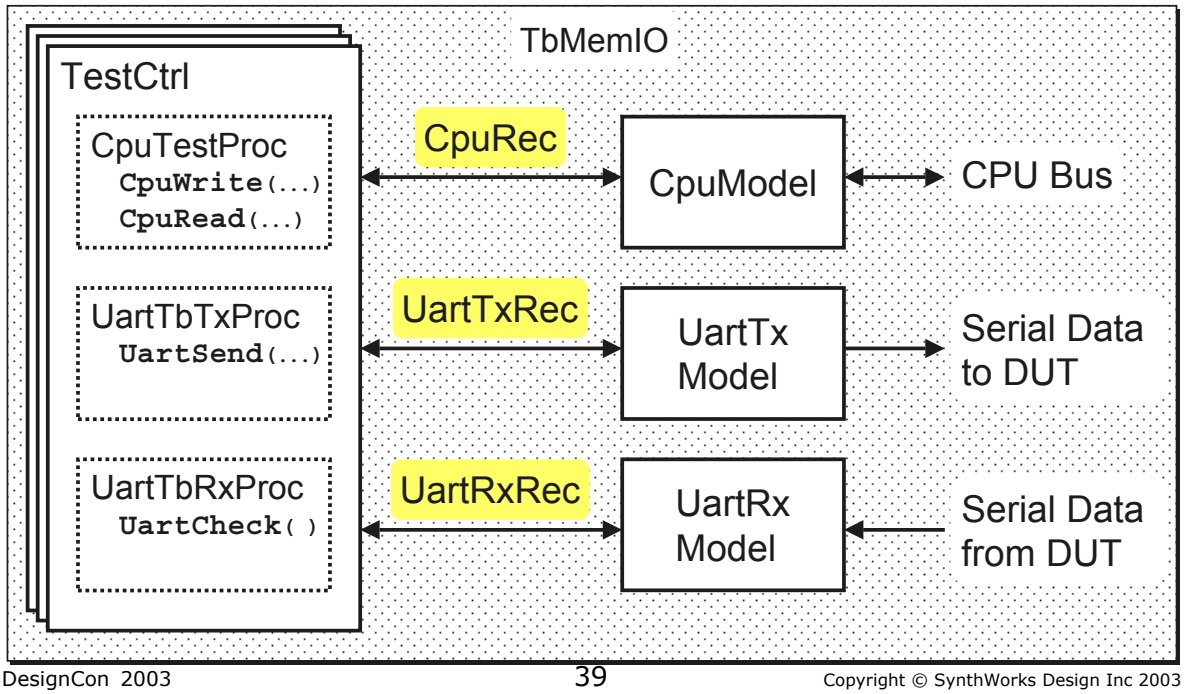
Functionality

- Wait for Transaction
- Code Functionality
- Put return value(s) in Record
- Signal End of Transaction

Testbench Details: Handshaking

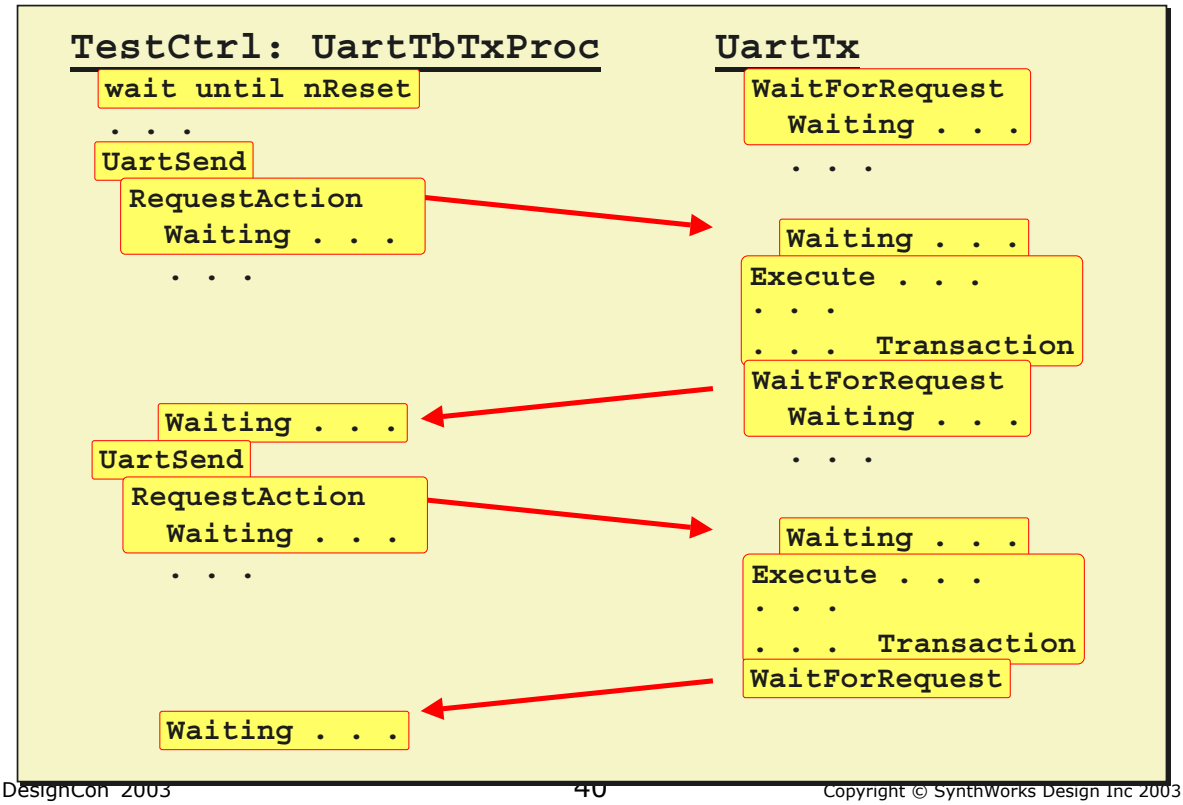
SynthWorks

- Handshaking between CPU Transactions and CpuModel is done through CpuRec



Handshaking

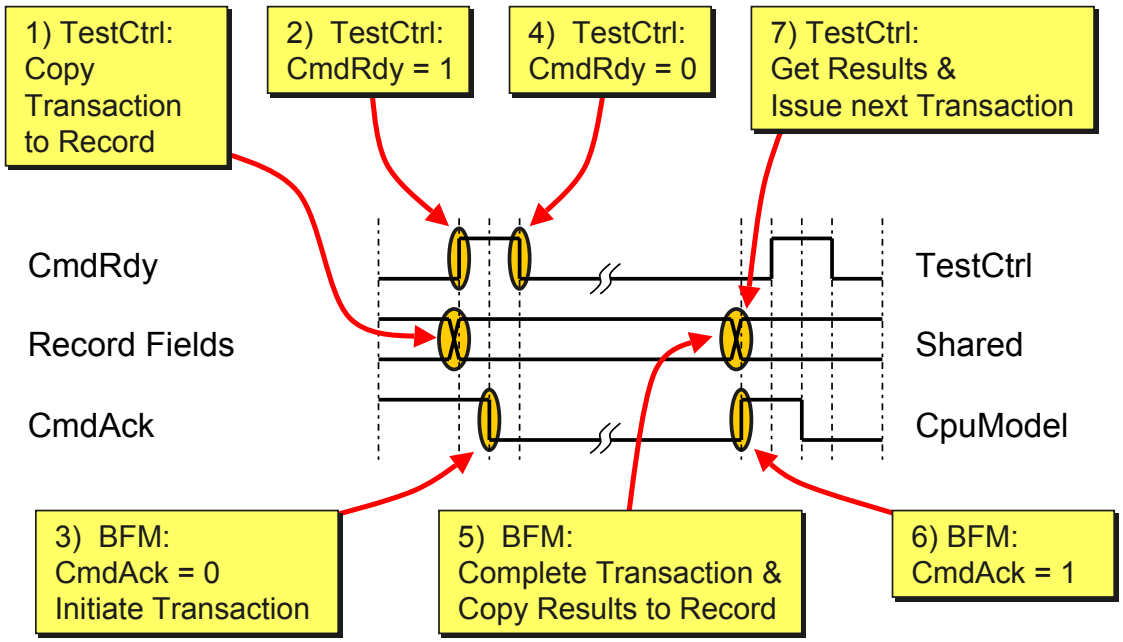
SynthWorks



Testbench Details: Handshaking

SynthWorks

- CpuRec fields CmdRdy and CmdAck are used for handshaking.



Procedure RequestAction

SynthWorks

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

  -- Find Ack at the level '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 ;
```

Procedure WaitForRequest

SynthWorks

```
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 ;
```

End of Previous Cycle

Start of Cycle

Details Summary

SynthWorks

- Using transaction tests + BFMs + a good set of abstractions,
  - Facilitates a subblock to system-level test pre-use methodology
  - Increases Readable, Usability
    - Decreases the complexity of writing a test
    - Readable by software and system engineers
  - Straight forward to implement all features of hardware verification languages (HVLs).
    - No additional costs for expensive EDA tools
- Major investment
  - Planning tests up front
  - Really should be doing this anyway

Want to Know More?

SynthWorks

Take SynthWorks' VHDL Testbenches and Verification Class

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.

SynthWorks VHDL Training

SynthWorks

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
A design and verification engineers introduction to VHDL syntax, RTL coding, and 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.

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