8 AMBA™ AHB

This chapter demonstrates how an AMBA™ AHB bus specification[1] and an IDT 71V433 Synchronous pipelined SRAM[2] are used to design and verify a memory slave controller in Verilog and *PSL*. Simulation with a *PSL* aware simulator was used for verification. The testbench does not have an automatic verifier, which is included in a traditional self-checking methodology. Instead, verification relies on ABV methodology.

8.1 AMBA™ AHB MEMORY SLAVE INTERFACE

The AMBA™ AHB interface represents an ARM bus standard. Figure 8.2-1 represents a high level view of the memory slave controller in the system environment.

---

[1] http://www.arm.com/armtech.nsf/html/AMBA_Spec?OpenDocument&style=AMBA

[2] http://www.idt.com/products/pages/Synchronous_SRAMs-71V433.html

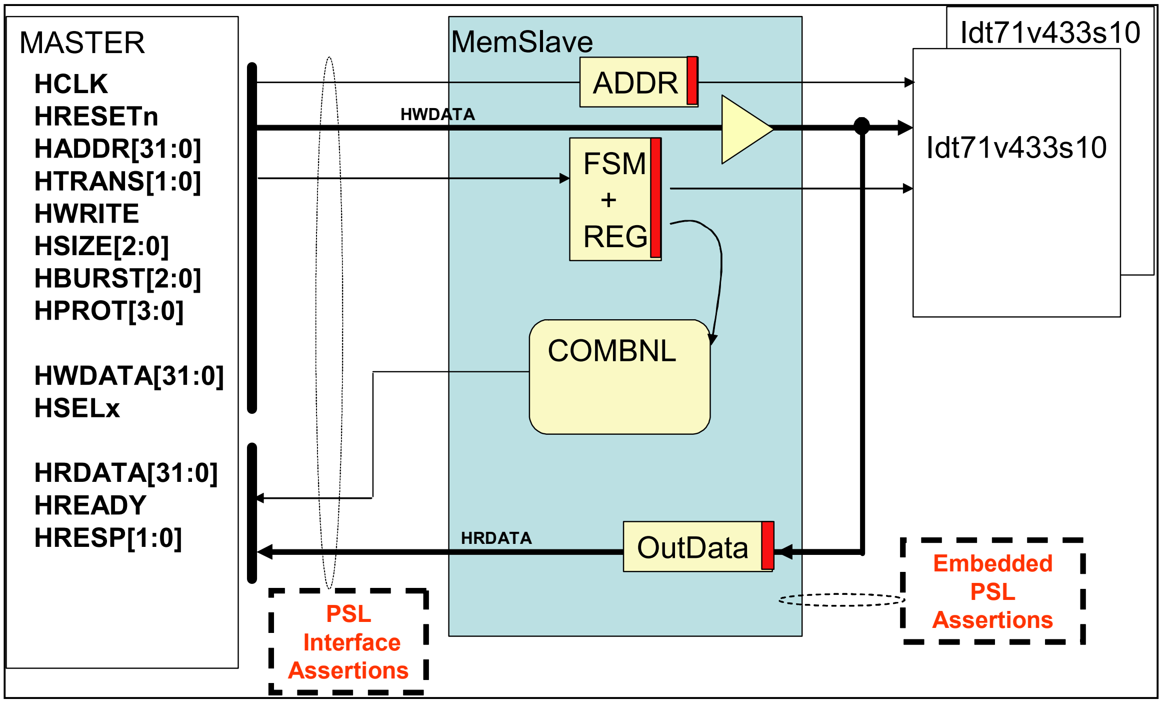


Figure 8.1 High Level View of Memory Slave Controller in System Environment

8.1.1 Design Process Used for AHB

The design process for the AMBA™ AHB memory slave interface consisted of the steps identified below.

8.1.2 Understand AHB AMBA™ Specification, Write Interface Properties

The AMBA™ AHB specification defines the interface signals, the timing relationships, and mode restrictions. As with all natural language specifications, the authors found some ambiguities in the explanations of the modes and timing descriptions. Fortunately, the authors were able to take advantage of the AMBA™ AHB interface properties written in *PSL*[3], which clarified many interface issues. If those properties were not available, then the effort of writing those properties in *PSL* would have been beneficial. Any misunderstandings would have been clarified through discussions with the originators of the specification or with newsgroups, such as *comp.sys.arm*. Figure 8.2.1-1 provides an example of the interface timing. The AMBA™ AHB compliance interface *PSL* properties are defined in a separate module, as shown in Figure 8.1.2-2. Those properties are used to provide a better understanding of the interfaces, and to provide verification that the design and testbench comply with the bus protocol. The module approach, instead of a **vunit** method, permits the definition of local constants, variables, and processes necessary to support the *PSL* compliance description.

---

[3] The Role of Assertions in Verification Methodologies using assertions in a simulation environment, Cadence, February 2003

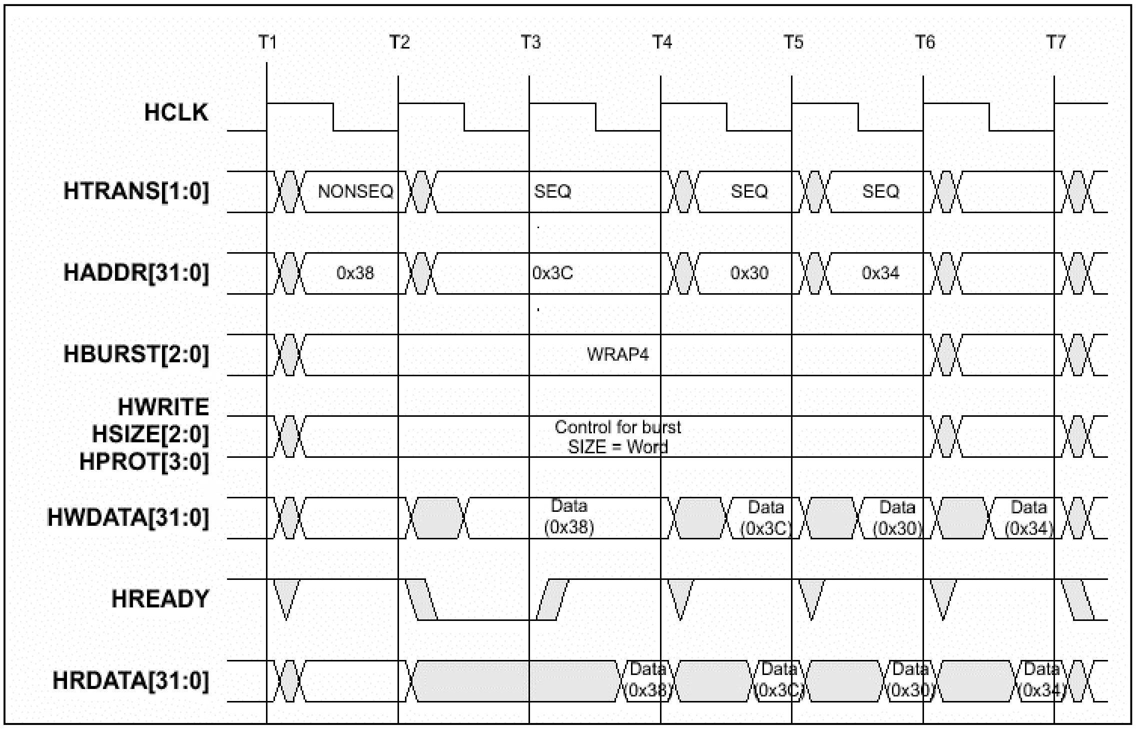


Figure 8.2.1 Interface Timing Example

```
/****************** AHB Interface PSL Sugar Assertions ***************/
`timescale 1 ns / 100 ps
module ahbCompliance (
   // Inputs
   hrdata,
   hclk, hresetn, haddr, htrans, hwrite, hsize, hburst, hprot,
   hwdata, hsel, hready, hresp
   );
// `define SIM 1
`include "ahb_param.v"

   input hclk;
   input hresetn;
   input [31:0] haddr;
   input [htrans_width -1:0] htrans;
   input                   hwrite;
   input [hsize_width -1:0]  hsize;
   input [hburst_width -1:0] hburst;
   input [3:0]                     hprot;
   input [31:0]            hwdata;
   input                   hsel;
   input                   hready;
   input [hresp_width -1:0]  hresp;

   input [31:0]            hrdata;

   // Transfer response: OKAY, ERROR, RETRY and SPLIT.
   parameter                       dataBusWidth = 32;
   parameter                       numSlaves = 32;
   parameter                       busMaster = 0;
```

```
reg [31:0]                  prev_haddr, prev_haddrPlusSize;
reg                         prev_hwrite, prev_hready;
reg [numSlaves-1:0]         prev_hsel;
reg [3:0]                   prev_hprot;
reg [hsize_width-1:0]  prev_hsize;
reg [hburst_width-1:0] prev_hburst;
reg [dataBusWidth-1:0]    prev_hrdata, prev_hwdata;
reg [htrans_width-1:0]                  prev_htrans, prev_hresp;
integer                     AddrIncrement;
integer                     NumberBeats=0;
/* capture data that is needed in the following assertions */
always @(posedge hclk or negedge hresetn)
  begin
        prev_haddr <= haddr;
        prev_hready <= hready;
        prev_hwrite <= hwrite;
        prev_hrdata <= hrdata;
        prev_hwdata <= hwdata;
        prev_htrans <= htrans;
        prev_hburst <= hburst;
        prev_hresp <= hresp;
        prev_hsel <= hsel;
        prev_hsize <= hsize;
        prev_hprot <= hprot;
        case (hsize) /* AddrIncrement is used to check the address */
          bits8: AddrIncrement=1;
          bits16: AddrIncrement=2;
          bits32: AddrIncrement=4;
          bits64: AddrIncrement=8;
          bits128: AddrIncrement=16;
          bits256: AddrIncrement=32;
          bits512: AddrIncrement=64;
          bits1024: AddrIncrement=128;
          default: ;
        endcase
        /* NumberBeats is used to check packet lengths */
        if (htrans == IDLE)
          NumberBeats <= 0;
        else if ( htrans == NONSEQ)
          NumberBeats <= 1;
        else if ( (htrans == SEQ) && (hready == 1) )
          NumberBeats <= NumberBeats + 1;
        else
          NumberBeats <= NumberBeats;
        /* determine the predicted address */
        if ((htrans != BUSY) && (hready != 0))
          prev_haddrPlusSize <= haddr + AddrIncrement;
  end
/*************************** PSL ASSERTIONS ************************/
/* define default clock that will be used to sample all subsequent asserts */
/* NOTE: it is important that data does not change on the same edge as clock
                because, like HDL, assertions can be sensitive to races */
// psl default clock = (posedge hclk);
```

Equivalent to the **prev**() function . Required in this model because tool did not support the **prev**() function when 1st edition of book was published. The use of **prev**() function is preferred.

Used in *PSL* assertions

```
// ------------------------------------------------
  /* Behavior: If reset is active then Transfer Type is IDLE and Resp is OKAY */
  // psl property IdleAndOkayAfterReset = always (
  // {(hresetn == 0);(hresetn == 1)} |->
  // {(htrans == IDLE) && (hresp == OKAY)});
  // psl assert IdleAndOkayAfterReset;
  //
  // ------------------------------------------------
  /* Behavior: If the transfer type is NONSEQ and Burst Mode is SINGLE,
                     then on the next clock cycle, the transfer type is not SEQ and not BUSY */
  // psl property BusyAndSeqNeverFollowNonseqOnSingleTransfer = always (
  // (htrans == NONSEQ) && (hburst == SINGLE ) ->
  // next ((htrans != SEQ) && (htrans != BUSY)) );
  // psl assert BusyAndSeqNeverFollowNonseqOnSingleTransfer;
  //
  // ------------------------------------------------
  /* Behavior: If the transfer type is BUSY and slave is ready,
                     then on the next clock cycle, the transfer type must not be IDLE and
                     must not be NONSEQ unless grant is 0 and ready is 1  */
  // psl property NeverGoFromBusyToIdleOrNonseqUnlessNoGrant = always (
  // ((htrans == BUSY) && (hready == 1)) ->
  // next ((htrans != IDLE) && (htrans != NONSEQ)));
  //// abort ((hGnt == 0) && (hready == 1)) || (hresetn == 0));
  // psl assert NeverGoFromBusyToIdleOrNonseqUnlessNoGrant;
  //
  // ------------------------------------------------
  /* Behavior: if the transfer type is IDLE, then on the next clock
                     cycle, the transfer type must be either IDLE or NONSEQ (default clock)*/
  // psl property AlwaysGoToNonseqFromIdle = always (
  // (htrans == IDLE) ->
  // next ((htrans == IDLE) || (htrans == NONSEQ)) );
  // psl assert AlwaysGoToNonseqFromIdle;
  //%% from code review BD 10/11/03:
  //%% "The property is a duplicate of NeverGoFromIdleToSeqOrBusy below."
  //
  // ------------------------------------------------
  /* Behavior: if the transfer type is BUSY then on the corresponding
                     data transfer , the slave must provide a zero wait state OKAY response */
  // psl property ResponseToBusyMustBeZeroWaitOKAY = always (
  // ((htrans == BUSY) && (hready == 1)) ->
  // next ((hresp == OKAY) && (hready == 1)) );
  // psl assert ResponseToBusyMustBeZeroWaitOKAY;
  //
  // ------------------------------------------------
  /* Behavior: if the transfer type is IDLE then on the corresponding data
                     transfer, the slave must provide a zero wait state OKAY response */
  // psl property ResponseToIdleMustBeZeroWaitOKAY = always (
  // ((htrans == IDLE) && (hready == 1)) ->
  // next ((hresp == OKAY) && (hready == 1)) );
  // psl assert ResponseToIdleMustBeZeroWaitOKAY;
  //
```

```
// -------------------------------------------------
  /* Behavior: if the previous response was OKAY, and the current response
                    is not OKAY, then Ready must be 0 during the second not OKAY response */
  // psl property ReadyMustBeZeroDuringFirstNotOkayResponse = always (
  // {(hresp == OKAY);(hresp != OKAY)} |->
  // {hready == 0}) ;
  // psl assert ReadyMustBeZeroDuringFirstNotOkayResponse;
  //
  //%% From code review BD 10/11/03:
  //%% I can' t find a check anywhere that hready must be true in the following cycle.
  //%% It' s a requirement of the protocol that
  //%% ((hresp != OKAY) && !hready) -> next ((hresp != OKAY) && hready)
  //%% It' s also a requirement that hresp be unchanged in these two cycles
  //%% And this property as written, while correct, fails to handle the condition
  //%% where the data phase is zero wait state
  // psl property HrespUnchangedWhenNewHready=
  //   always (((hresp != OKAY) && !hready) -> next ((hresp != OKAY) && hready));
  // psl assert HrespUnchangedWhenNewHready;
  //
  // -------------------------------------------------
  // Behavior: if the response is SPLIT or RETRY, the master must drive IDLE
  //// psl property FirstNotOkayResponseCausesNextIdle = always (
  //// ((hresp == SPLIT) || (hresp == RETRY)) && (hready == 0) ->
  //// next (htrans == IDLE));
  //%% From Code review BD 10/11/03:
  //%% No, this is absolutely NOT a requirement. It' s not true when the
  //%% current address transfer and current data transfer are from different masters,
  //%% i.e., when bus is in the process of changing mastership
  //
    // -------------------------------------------------
  /* Behavior: if the transfer type is SEQ or BUSY, then the controls of the
                    corresponding data transfer must be the same as their previous control
                    NOTE: the equivalence operator is used with hprot
                    because this is an optional signal and may be an "X" value if not
                    connected */
  // psl property ControlMustBeConstantDuringABurst = always (
  // (htrans == SEQ) || (htrans == BUSY) ->
  // ((hsize == prev_hsize) && (hprot === prev_hprot) &&
  // (hwrite == prev_hwrite)) );
  // psl assert ControlMustBeConstantDuringABurst;
  //
  //%% From code review BD 10/11/03:
  //%% It' s confusing to talk about the controls of the corresponding
  //%% data transfer.  Tthe use of the implication means the property is comparing the
  //%% controls for the CURRENT address with the previous CYCLE. Note this is not
  //%% the previous address TRANSFER.
  //%% In general, it is recommended to use properties that do not rely on
  //%% latches for storage, rather than the expressiveness of the language.
  //%% The requirement could be re-written 100% in PSL without using the latches as:
```

```
//%% forall i in boolean:
//%%   forall j in {bits8:bits1024 }:
//%%     forall k in {0:15 } :
//%%       forall h in {INCR:INCR16} :
//%%         always ({(hready && (htrans==NONSEQ) && (hburst==h)
//%%                   && (hwrite==i) && (hsize==j) &&(hprot==k))} |=>
//%%                     {((hburst==h) && (hwrite==i) && (hsize==j)
//%%                        &&(hprot==k) &&
//%%                        ((htrans==SEQ)|| (htrans==BUSY)))[*];
//%%                        ((htrans!=SEQ)&&(htrans!=BUSY)) } );
//%% which is actually a more comprehensive property because it checks
//%% hburst is constant, which is omitted from the original
//
// -------------------------------------------------
/* Behavior: if Ready is 0 and the response is OK or ERROR, the address
             and control, and data from the master are held constant (default clock)
             NOTE: the equivalence operator is used with hprot because this is an
             optional signal and may be an "X" value if not connected */
// psl property ControlMustBeConstantWhenSlaveNotReady = always (
// (hready == 0) && ((hresp == OKAY) || (hresp == ERROR)) -> next
// (htrans == prev_htrans) && (hsize == prev_hsize) &&
// (hprot === prev_hprot) && (hwrite == prev_hwrite) &&
// (haddr == prev_haddr) && (hburst == prev_hburst)
// abort (hresetn == 0));
// psl assert ControlMustBeConstantWhenSlaveNotReady;
//
//%% From code review BD 10/11/03:
//%% The comment mentions data but it' s not included in the SERE.
//%% The requirement also holds for transfers that end in
//%% SPLIT & RETRY up to the cycle where hready is low,
//%% and HRESP != OKAY, but this doesn' t seem to be checked anywhere
//
// -------------------------------------------------
/* Behavior: if ready is 0 and the previous transfer type was SEQ or NONSEQ,
             then on the next cycle the write data is the same as the previous write data
             (default clock) (This assertion is designed to fail!!!) */
// psl property WriteDataMustBeConstantWhenSlaveNotReadyAndDataBeingTransferred = always (
// (hready == 0) && (hwrite == 1) &&
// ((prev_htrans == SEQ) || (prev_htrans == NONSEQ)) -> next
// (hwdata == prev_hwdata)
// abort (hresetn == 0));
// psl assert WriteDataMustBeConstantWhenSlaveNotReadyAndDataBeingTransferred;
//
//%% From code review BD 10/11/03
//%% IMHO this is not a good way to demonstrate a failing assertion.
//%% The logic of the property is completely confused
//%% because the pipelined address architecture does not allow you
//%% to reason about hwrite and hwdata in this way. It' s
//%% very difficult to show how to make small changes to this property to make it correct.
//
```

```
// --------------------------------------------------
/* Behavior: if transfer type is BUSY then address does not change  */
// psl property AddrHeldWhenMasterBusy = always (
// (htrans == BUSY) -> next (haddr == prev_haddr) );
// psl assert AddrHeldWhenMasterBusy;
//
// --------------------------------------------------
/* Behavior: if transfer is in progress, then 1K boundary is not exceeded  */
// psl property PageAddressNeverExceeds1kBoundary = always (
// ((htrans == SEQ) || (htrans == BUSY)) ->
// (haddr[31:10] == prev_haddr[31:10]) );
// psl assert PageAddressNeverExceeds1kBoundary;
//
//%% From code review BD 10/11/03:
//%% This also applies to the first/second transfer in the burst.
//%% They must also be in the same 1k address page.
//
// --------------------------------------------------
/* Behavior: If the burst type is not INCR and the transfer type is not
                IDLE , then the NumberBeats for the Burst Mode is not exceeded  */
// psl property BurstIsNotTooLong = always (
// (hburst != INCR) && (htrans != IDLE) ->
// ((hburst == SINGLE) && (NumberBeats <= 1)) ||
// ((hburst == WRAP4) && (NumberBeats <= 4)) ||
// ((hburst == WRAP8) && (NumberBeats <= 8)) ||
// ((hburst == WRAP16) && (NumberBeats <= 16)) ||
// ((hburst == INCR4) && (NumberBeats <= 4)) ||
// ((hburst == INCR8) && (NumberBeats <= 8)) ||
// ((hburst == INCR16) && (NumberBeats <= 16)) );
// psl assert BurstIsNotTooLong;
//
//%% From code review BD 10/11/03:
//%% Once again, this does not exploit the expressiveness of PSL,
//%% and I believe it can be re-written 100% in PSL as three
//%% or four alternative outcomes without needing the  forall’
//%% construct, and without requiring the beat counting logic.
//
// --------------------------------------------------
/* Behavior: For all but INCR Burst mode, if the end of the packet is being
                transferred as indicated by a transition from SEQ to IDLE when Resp is ok
                then the NumberBeats for the Burst Mode is the max number unless grant is 0 */
// 7/16/03 Modified to correct error in number because of last increment
// psl property BurstIsNotTooShort = always (
// {((hburst != INCR) && (hresp == OKAY) && (htrans == SEQ));
// (htrans == IDLE) } |->
// {((prev_hburst == SINGLE) && (NumberBeats == 1)) ||
// ((prev_hburst == WRAP4) && (NumberBeats == 4+1)) ||
// ((prev_hburst == WRAP8) && (NumberBeats == 8)) ||
// ((prev_hburst == WRAP16) && (NumberBeats == 16)) ||
// ((prev_hburst == INCR4) && (NumberBeats == 4)) ||
// ((prev_hburst == INCR8) && (NumberBeats == 8)) ||
// ((prev_hburst == INCR16) && (NumberBeats == 16))});
//// abort (hGnt == 0)) ;
// psl assert BurstIsNotTooShort;
//
```

```
//%% From code review BD 10/11/03
//%% Ditto the previous comment.
//%% In addition this looks VERY suspicious. Why is the WRAP4 beat
//%% count 4+1? Also it doesn’ t take care of the situation
//%% where you have back-to-back operations  a burst may be followed by a NONSEQ
//
// -------------------------------------------------
/* Behavior: if the transfer type is IDLE , then on the next clock cycle,
                    the transfer type is not SEQ and the transfer type is not BUSY  */
// psl property NeverGoFromIdleToSeqOrBusy = always (
// (htrans == IDLE) ->
//   next ((htrans != SEQ) && (htrans != BUSY)) );
// psl assert NeverGoFromIdleToSeqOrBusy;
//
//%% From code review BD 10/11/03
//%% The property is a duplicate of AlwaysGoToNonseqFromIdle above.
//
// -------------------------------------------------
/* Behavior: RETRY is always asserted for two cycles unless reset is asserted */
// psl property RetryResponseMustPersistTwoCycles = always (
//   {(hresp != RETRY);(hresp == RETRY)} |=>
//                      {(hresp == RETRY)} abort (hresetn == 0));
// psl assert RetryResponseMustPersistTwoCycles;
//
//%% From code review 10/11/03
//%% This doesn’ t fully verify back-to-back operations without wait states.
//
// -------------------------------------------------
/* Behavior: SPLIT is always asserted for two cycles unless reset is asserted */
// psl property SplitResponseMustPersistTwoCycles = always (
//   {(hresp != SPLIT);(hresp == SPLIT)} |=>
//              {(hresp == SPLIT)} abort (hresetn == 0));
// psl assert SplitResponseMustPersistTwoCycles;
//
//%% From code review 10/11/03
//%% This doesn’ t fully verify back-to-back operations without wait states.
//
// -------------------------------------------------
/* Behavior: ERROR is always asserted for two cycles unless reset is asserted */
// psl property ErrorResponseMustPersistTwoCycles = always (
//   {(hresp != ERROR);(hresp == ERROR)} |=>
//        {(hresp == ERROR)} abort (hresetn == 0));
// psl assert ErrorResponseMustPersistTwoCycles;
//
//%% From code review 10/11/03
//%% This doesn’ t fully verify back-to-back operations without wait states.
//
```

```
// -------------------------------------------------
/* Behavior: if Burst mode is WRAP4 and Size is 8 bits, then if
                    transfer mode is SEQ or BUSY, then if previous transfer mode is not
                    busy and previous Ready is not zero, then check that the address is as predicted */
// psl property CorrectAddressDuringPageSize4BurstWrap = always (
// ((hburst== WRAP4) && (hsize == bits8)) ->
// ((htrans == SEQ || htrans == BUSY)) ->
// ((prev_htrans != BUSY) && (prev_hready != 0)) ->
// ((haddr[31:2] == prev_haddr[31:2]) &&
// (haddr[1:0] == prev_haddrPlusSize[1:0])) );
// psl assert CorrectAddressDuringPageSize4BurstWrap;
// -------------------------------------------------
// psl property CorrectAddressDuringPageSize8BurstWrap = always (
// ((hburst== WRAP4) && (hsize == bits16)) ||
// ((hburst== WRAP8) && (hsize == bits8)) ->
// ((htrans == SEQ) || (htrans == BUSY)) ->
// ((prev_htrans != BUSY) && (prev_hready != 0)) ->
// ((haddr[31:3] == prev_haddr[31:3]) &&
// (haddr[2:0] == prev_haddrPlusSize[2:0])) );
// psl assert CorrectAddressDuringPageSize8BurstWrap;
//
// -------------------------------------------------
// psl property CorrectAddressDuringPageSize2048BurstWrap = always (
// ((hburst== WRAP16) && (hsize == bits1024)) ->
// ((htrans == SEQ) || (htrans == BUSY)) ->
// ((prev_htrans != BUSY) && (prev_hready != 0)) ->
// ((haddr[31:11] == prev_haddr[31:11]) &&
// (haddr[10:0] == prev_haddrPlusSize[10:0])) );
// psl assert CorrectAddressDuringPageSize2048BurstWrap;
//
// -------------------------------------------------
/* Behavior: the address must increment by size during all INCR beats */
// psl property AddressIncBySizeDuringAllBurstIncrBeats = always (
// ((hburst == INCR16) || (hburst == INCR8) ||
// (hburst == INCR4)) ->
// ((htrans == SEQ) || (htrans == BUSY)) ->
// ((prev_htrans != BUSY) && (hready != 0)) ->
// (haddr == prev_haddrPlusSize) );
// psl assert AddressIncBySizeDuringAllBurstIncrBeats;
//
// -------------------------------------------------
/* Behavior: always address must be aligned to the transfer size */
// psl property AddressNotAlignedToTransferSize = always (
// ((hsize == bits8) ||
// ((hsize == bits16) && (haddr[0] == 1'b0)) ||
// ((hsize == bits32) && (haddr[1:0] == 2'b00)) ||
// ((hsize == bits64) && (haddr[2:0] == 3'b000)) ||
// ((hsize == bits128) && (haddr[3:0] == 4'b0000)) ||
// ((hsize == bits256) && (haddr[4:0] == 5'b00000)) ||
// ((hsize == bits512) && (haddr[5:0] == 6'b000000)) ||
// ((hsize == bits1024) && (haddr[6:0] == 7'b0000000)) ) );
// psl assert AddressNotAlignedToTransferSize;
//
```

```
// --------------------------------------------------
  /* Behavior: always the maximum number of wait states is 16 */
  // psl property NeverMoreThan16WaitStates = always (
  // {(hready == 1);(hready == 0)} |=>
  // {{(hready == 0)[*0..15]};{hready == 1}}
  // abort (hresp != OKAY));
  // psl assert NeverMoreThan16WaitStates;
  //
endmodule // ahbCompliance
```

Figure 8.1.2-2 AMBA AHB Compliance Interface *PSL* Properties (ch8/ahb95/ahbCompliance.v)

8.1.3 Understand IDT71v433s10 SRAM specification

Since the memory slave interfaces with the IDT71v433s10 SRAM, a pipelined memory, it is imperative that the memory design and timing information be well understood. Figure 8.1.3 provides the architecture of the memory device.

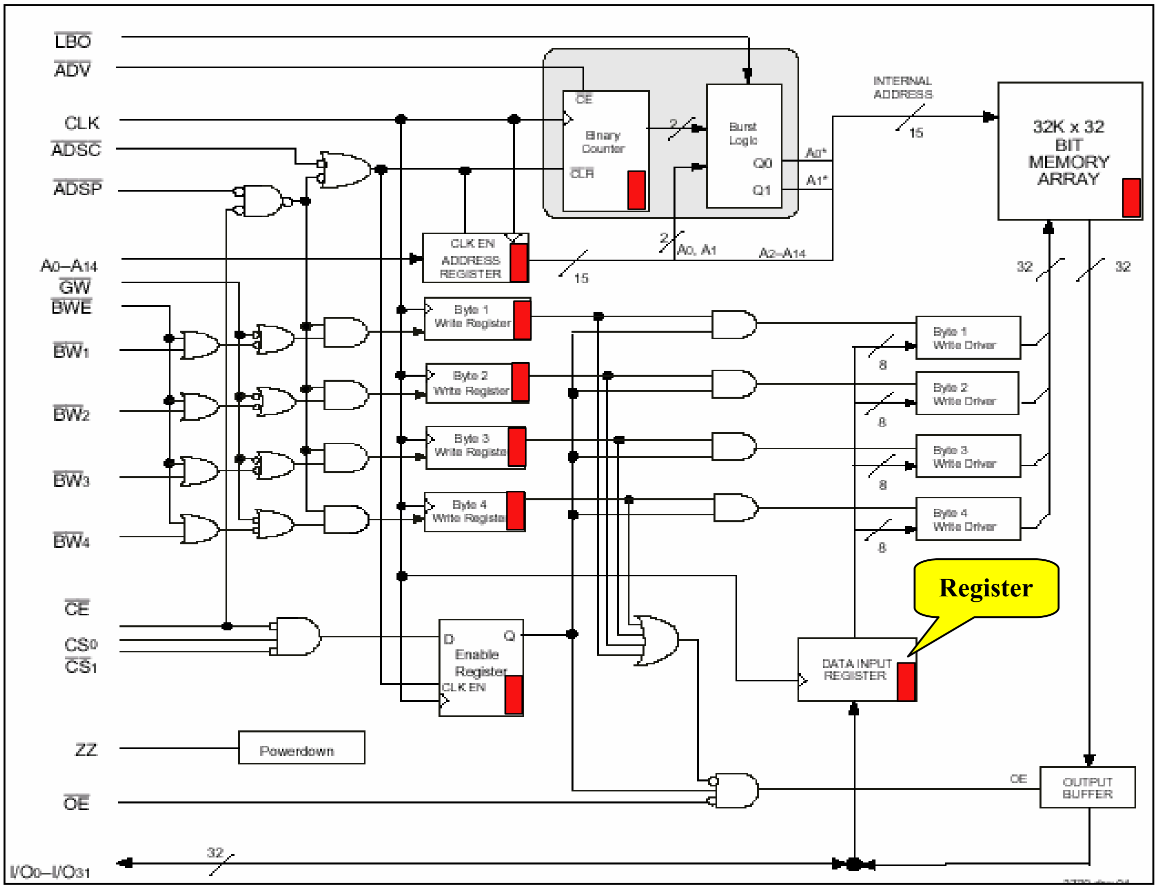


Figure 8.1.3 IDT 71V433 Architectural

8.1.4 Perform and Review Architectural Diagram

Figure 8.1 demonstrates the high-level architecture of the memory slave interface. It basically consists of registers to store the address and memory output data. It also consists of an FSM to control the cycles. A cycle timing diagram is often necessary to understand the cycle implications of the transactions. These diagrams represent a form of “graphical assertions.” Figure 8.1.4 demonstrates the cycle timing for memory access from the AHB interface.

8.1.5 Write RTL Structure and *PSL* Assertions

The RTL structure includes the port interface and definition of design internal constants, parameters, signals, registers, FSM states, and any functions and procedures that are needed for *PSL* support. Figure 8.1.5-1 represents a block view of the ports. Figure 8.1.5-2 is a *PSL* description of the properties of the design. This *PSL* description typically evolves though the lifetime of the design as more information and understanding of the RTL and verification evolves.[4] This description documents the properties of the RTL, and are very useful during architectural and code reviews. They also provide valuable feedback during the verification process (e.g., simulation) because the assertions are closely linked to the design at the white-box level. It is recommended that this description use the “patterns format” described in the book *Assertion-Based Design* to document design insights, assumptions and decisions. [5]

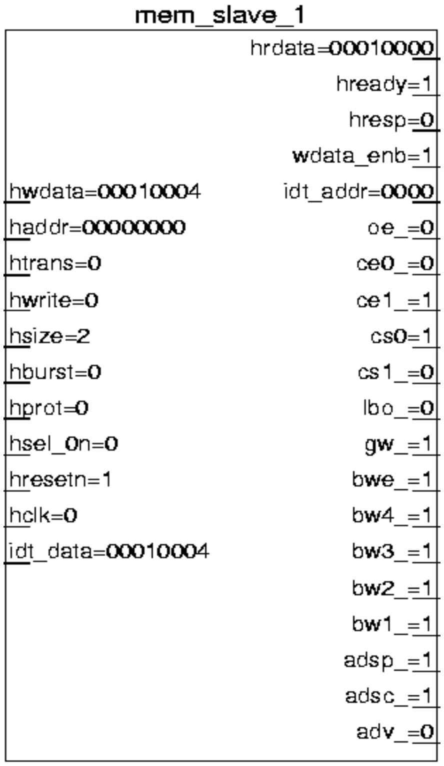


Figure 8.1.5-1 Memory Slave interface Block View

---

[4] All changes made during the design were kept in the file to demonstrate the evolution of the process.

[5] Assertion-Based Design, Harry Foster, Adam Krolnik, David Lacey, 2003, Kluwer Academic Publishers, ISBN 1-4020-7498-0

										Slave	<----- IDT INTERNAL ----->				Slave	
	htrans [1:0]	hwrite	hsize [2:0]	hburst [2:0]	haddr [31:0]	hwdata [31:0]	hrdata [31:0]	hready	hresp [1:0]	addr	Mem Addr	Mwrite Data	Mread Data	ADVn	RdReg	pipe
	NonSeq	0			A1			1	OKAY					1		
	SEQ	0			A2	dA1		1	OKAY	A1				1		
	SEQ	0			A3	dA2		1	OKAY	A2	A1	dA1		1		
	SEQ	0			A4	dA3		1	OKAY	A3	A2	dA2		1		
	NonSeq	1			A5	dA4		1	OKAY	A4	A3	dA3		1		
	SEQ	1			*A6***			0	OKAY	A5	A4	dA4		1		P1
	SEQ	1			*A6***			0	OKAY	A5	A5		dA5	0		P2
	SEQ	1			A6		dA5	1	OKAY	A5			dA6	0	dA5	P3
	SEQ	1			A7		dA6	1	OKAY	A6	A6		dA7	0	Da6	P4
	SEQ	1			A8		dA7	1	OKAY	A7	A7		dA8	1	dA7	P5
	NonSeq	1	WORD	WRAP4	A9		dA8	1		A8	A8			1	dA8	P6
		1	WORD	WRAP4	*A10***			0		A9				1		
		1	WORD	WRAP4	*A10***			0			A9		dA9	0		
		1	WORD	WRAP4	A10		dA9	1			A10		dA10	0	dA9	
		1	WORD	WRAP4	A11		dA10	1			A11		dA11	0	dA10	
					A12		dA11	1			A12		dA12	1	dA11	
							dA12	1						1	dA12	
write with busy																
	NonSeq	0			A1			1	OKAY					1		
	BUSY	0			A2	dA1		1	OKAY	A1				1		
	SEQ															
	SEQ	0			A3	dA2		1	OKAY	A2	A1	dA1		1		
	SEQ	0			A4	dA3		1	OKAY	A3	A2	dA2		1		
	NonSeq	1			A5	dA4		1	OKAY	A4	A3	dA3		1		
	SEQ	1			*A6***			0	OKAY	A5	A4	dA4		1		
	NonSeq	0			A1			1	OKAY							
	SEQ	0			A2	dA1		1	OKAY	A1						
	SEQ	0			A3	dA2		1	OKAY	A2	A1	dA1				
	SEQ	0			A4	dA3		1	OKAY	A3	A2	dA2				
	NonSeq	1			A5	dA4		1		A4	A3	dA3				
	SEQ	1			*A6***			0	OKAY	A5	A4	dA4				
	SEQ	1			*A6***			0	OKAY		A5		dA5			
	SEQ	1			A6		dA5	1	OKAY					dA5	1	
	SEQ	1			*A7***			0	OKAY	A6					1	
	SEQ				*A7***			0			A6		dA6		1	
	SEQ				*A7*		dA6	1						dA6	1	
	SEQ				*A8***			0		*A7*					1	
	SEQ				*A8***			0			*A7*		dA7		1	
	SEQ				A8		dA7	1						dA7	1	
	SEQ							0		A8					1	
	SEQ							0			A8		dA8		1	
							dA8	1						dA8	1	

Figure 8.1.4 Cycle Timing For Memory Access from the AHB Interface (amba2.xls)

```
//| # Context # Only global word writes will be honored.
//| # Solution # For a WRITE the following is required:
//|    @cycle x   adsp_ = 1'b0  // load address
//|              adsc_ = 1'b1
//|              adv_  = 1'b1  // no advance
//|              lbo_  = 1'b0  // linear burst mode sequence
//|              ce_   = 1'b0  // enable chip
//|              cs0   = 1'b1  // chip select
//|              cs1_  = 1'b0  // chip select
//|              gw_   = 1'b0  // global write
//|              {bwe_, bw4_, bw3_, bw2_, bw1_} == 5'b11111 }; // could be don't care,
//|                    but will enforce for consistency
//| # Considerations #  Powerdown ZZ signal is not modeled, and should be
//|               in the deasserted dated, i.e. ZZ = 1'b0
// ********** Check hResp and hready in response to a NONSEQ write cycle **********
// ** New address cycle when htrans==NONSEQ
// psl sequence qNonSeqWriteTransfer =
//              {done_xfrn==1'b0 && htrans==NONSEQ &&
//               hburst==SINGLE && hwrite==1'b1 && hsel_0n==1'b0 };
//
// psl property HrespOKHready1ForNonSeqWriteTransfer =
//   always({qNonSeqWriteTransfer} |-> {hresp==OKAY && hready==1'b1});
//
// psl assert HrespOKHready1ForNonSeqWriteTransfer;
//
// ********** Check that writes are in SINGLE bursts **********
// ** This is a check to ensure design assumption are maintained during verification
// psl sequence qWriteTransfer =
//      {done_xfrn==1'b0 &&  hwrite==1'b1 && hsel_0n==1'b0 };
//
// psl property NeverWriteInOtherThanSingleBurst =
//   always ({qWriteTransfer} |-> {hburst == SINGLE});
//  BAD never ({qWriteTransfer} |-> {(hburst != SINGLE)});
//
// psl assume NeverWriteInOtherThanSingleBurst;
//
// ********** Check states of WRITE FSM  **********
// ** The WRITE FSM states: FSMW_IDLE, FSMW_BUSY, FSMW_WRITE
// ** Check of proper state transition, in response to a new WRITE request
// psl property FsmWriteAfterNonSeqCycle =
//   always({qNonSeqWriteTransfer} |=> {fsm_write==FSMW_WRITE});
//
// psl assert FsmWriteAfterNonSeqCycle;
//
// ** Check state and hready when transfer is write in BUSY
// psl property FsmWriteState2Busy =
//      always({fsm_write==FSMW_WRITE && htrans==BUSY}
//                 |=> {hready==OKAY && fsm_write==FSMW_BUSY});
//
// psl assert FsmWriteState2Busy;
//
```

```
//  ** FSMW busy to Write
 // psl sequence qWriteTransferInBusy2Write =
 //     {done_xfrn==1'b0 &&  hwrite==1'b1 && hsel_0n==1'b0
 //         && htrans==NONSEQ && fsm_write==FSMW_BUSY};
 //
 // psl property FsmWriteBusy2Write=
 //   always({qWriteTransferInBusy2Write} |=> {fsm_write==FSMW_WRITE});
 //
 // ** Check FSMW to IDLE mode
 // ** in IDLE or read mode
 // psl property FsmWriteState2Idle =
 //    always({htrans==IDLE ||
 //          (done_xfrn==1'b0 &&  hwrite==1'b0)} |=> {fsm_write==FSMW_IDLE});
 //
 // psl assert FsmWriteState2Idle;
 //
 // ** check FSM to
 //  ** Ensure for verification that htrans is never in SEQ for WRITE
 // psl sequence qSeqWriteTransfer =
 //      {done_xfrn==1'b0 && htrans==SEQ && hwrite==1'b1 && hsel_0n==1'b0 };
 //
 // psl property SeqWriteTransfer=
 //     never (qSeqWriteTransfer);
 //
 // psl assert SeqWriteTransfer;
 //
 // ********** Check hready and proper memory controls for Write to memory **********
 // ** Check load of memory address, and write of data in cycles following the SEQ,
 // ** provided htrans is not BUSY
 // psl sequence qWriteToMem =
 //    {done_xfrn==1'b0 && fsm_write==FSMW_WRITE && htrans!= BUSY};
 //
 // ** Write control signals to idt71v433s10 device
 //   7/05/03  failed in 1st run ??
  // caused by incorrect assumption in TB in wdata_enb
  // other error in ";" always ({qWriteToMem} |-> { hready==1'b1;
 // TB is correct, 0-> write, 1-> read , fixed this code
 //  7/07 error in handling memory device.
 // must change for writes: from adsc_ = 1'b1 TO adsc_ = 1'b0
 //                      from adsp_ = 1'b0 TO adsp_ = 1'b1
 // psl property WriteToIdtMem =
 //   always ({qWriteToMem} |-> { hready==1'b1 &&
 //       adsp_ == 1'b1 && adsc_ == 1'b0 && adv_ == 1'b1 &&
 //       lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
 //       cs1_  == 1'b0 && gw_   == 1'b0 && bwe_ == 1'b1 &&
 //       ({bwe_, bw4_, bw3_, bw2_, bw1_} == 5'b11111) &&
 //        wdata_enb==1'b0});
 //  psl assert WriteToIdtMem;
 //
 // ** Check wdata_enb is in WRITE direction, only when in qWriteToMem sequence
 // psl property WriteDataEnb =
 //    always ({qWriteToMem} |-> {wdata_enb==1'b0});
 // BAD    never ({qWriteToMem} |-> {wdata_enb==1'b1});
 //
 // psl assert WriteDataEnb;
 //  ** check for no writes to mem when htrans is BUSY or fsm_write is
 //  ** in states other thatn FSMW_WRITE
 // psl sequence qNotWriteSeq =
 //     {done_xfrn==1'b0 && htrans==BUSY &&
 //      (fsm_write==FSMW_BUSY | fsm_write==FSMW_IDLE | fsm_read==FSMR_IDLE)};
 //
 // psl property NoWriteToIdtMem =
 //   always ({qNotWriteSeq}
 //       |-> { adsp_ == 1'b1 && adsc_ == 1'b1 && adv_ == 1'b1 &&
 //             lbo_  == 1'b0 && ce_   == 1'b1 && cs0   == 1'b1 &&
 //             cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 &&
 //             ({bwe_, bw4_, bw3_, bw2_, bw1_} == 5'b11111) &&
 //               wdata_enb==1'b1});
 //
 // psl assert NoWriteToIdtMem;
 //   ---------------------------------------------------------
 // ******************* MEM_READS*******************
 //| # Pattern name #  MEM_READS
 //| # Problem #   The architecture with the IDT device provides the first
 //| data output in the third cycle after the master asserts the address
 //| during the NONSEQ htrans cycle. In addition, for burst, the IDT device
 //| allows for the advance mode to increment the address in a
 //| WRAP4 burst mode.  If the htrans==BUSY, then the IDT device needs to be put on hold.
 //| During the memory pipeline delay cycles, the slave must set the hready to 1'b0.
 //| Only HBurst of SINGLE and WRAP4 will be accepted because of limitation of memory device.
 //| # Motivation #  Efficiency.  Restriction for WRAP4 is set by memory device selection.
 //|               Other motivation is design simplicity of slave controller.
 //| # Context #  Reads in SINGLE or WRAP4 only.
 //| # Solution #
 //|  Reads in SINGLE or WRAP4 ONLY
 //|  1. The 2 cycles following htrans==NONSEQ, the hready shall be 1'b0.
 //|  2. If hburst == SINGLE then adv_ remains 1'b1 (no advance)
 //|    FSM cycles:  FSMR_IDLE |=>  FSMR_READ_P1 |=> FSMR_READ_P2 |=> FSMR_READ_P3
 //|  3. If hburst == WRAP4 then fsm_read will transition
 //|     FSMR_IDLE |=>  FSMR_READ_P1 |=> FSMR_READ_P2 |=> FSMR_READ_P3
 //|             |=>  FSMR_READ_P4 |=> FSMR_READ_P5 |=> FSMR_READ_P6
 //|     FSM many remain in FSMR_READ_P4, FSMR_READ_P5, or FSMR_READ_P6
 //|     if htrans==BUSY.
 //|
 //|    if htrans==SEQ |->
 //|           adv_ = 1'b0 in FSMR_READ_P2, FSMR_READ_P3, or FSMR_READ_P4
 //|
 //| # Considerations #   Powerdown ZZ signal is not modeled, and should be
 //|              in the deasserted dated, i.e. ZZ = 1'b0
 //
 // ********** Check all reads to be hburst SINGLE or WRAP4 ONLY **********
 // psl sequence qReadTransfer =
 //     {done_xfrn==1'b0 &&  hwrite==1'b0 && hsel_0n==1'b0 };
 //
 // psl property ReadInSingleOrWrap4 =
 //   always ({qReadTransfer} |-> {hburst==SINGLE | hburst== WRAP4});
 //
 // psl assume ReadInSingleOrWrap4;
 //
```

```
// ********** Check hResp to NONSEQ READ cycle **********
  // 7/15/03 need to differentiate between 1st xfr, and subsequent
  //  cycles of the read.  Add the done_xfrn
  // psl sequence qNonSeqReadTransfer =
  //          {htrans==NONSEQ && hwrite==1'b0 && hsel_0n==1'b0 && done_xfrn==1'b0};
  // 7/15/03 Correct error in design that was defined in Excel timing
  // psl property HrespOKHready1InNonSeqReadTransfer =
  //   always({qNonSeqReadTransfer} |->
  //         {hresp==OKAY && hready==1'b1;
  //          hresp==OKAY && hready==1'b0;
  //          hresp==OKAY && hready==1'b0;
  //          hresp==OKAY && hready==1'b1});
  // psl assert HrespOKHready1InNonSeqReadTransfer;
  //
  // ********** Check READ FSM and hready for read cycles **********
  // 7/15/03 insure single 1st read sequence
  // psl sequence qNonSeqReadTransferSingle =
  //       {hburst==SINGLE && htrans==NONSEQ && hwrite==1'b0 &&
  //                               hsel_0n==1'b0 && done_xfrn==1'b0};
  // 7/15/03 insure single 1st read sequence
  // psl sequence qNonSeqReadTransferWrap4 =
  //       {hburst==WRAP4 && htrans==NONSEQ && hwrite==1'b0 &&
  //                     hsel_0n==1'b0 && done_xfrn==1'b0};
  //
  // 7/21/03 Change due to error detected in compliance PSL
 //// psl property ControlMustBeConstantWhenSlaveNotReady = always (
 //// (hready == 0) && ((hresp == OKAY) || (hresp == ERROR)) -> next
 //// (htrans == prev_htrans) && (hsize == prev_hsize) &&
 //// (hprot === prev_hprot) && (hwrite == prev_hwrite) &&
 //// (haddr == prev_haddr) && (hburst == prev_hburst)
 //// abort (hresetn == 0));
 // 721 fail??
  // psl property ReadFsmPipelineSingle =
  //     always({qNonSeqReadTransferSingle} |=>
  //         {fsm_read==FSMR_READ_P1 && hready==1'b0;
  //          fsm_read==FSMR_READ_P2 && hready==1'b0;
  //          fsm_read==FSMR_READ_P3 && hready==1'b1;
  //          (fsm_read==FSMR_IDLE)});
  //was          (fsm_read==FSMR_READ_P1 || fsm_read==FSMR_IDLE)});
  //
  // psl assert ReadFsmPipelineSingle;
  // ----------
  // psl property ReadFsmPipelineWrap4 =
  //     always({qNonSeqReadTransferWrap4} |=>
  //         {fsm_read==FSMR_READ_P1 && hready==1'b0;
  //          fsm_read==FSMR_READ_P2 && hready==1'b0;
  //          fsm_read==FSMR_READ_P3 && hready==1'b1;
  //          ((fsm_read==FSMR_READ_P4 && htrans==SEQ && hready==1'b1) ||
  //            fsm_read==FSMR_READ_P3 && htrans==BUSY && hready==1'b1 ||
  //            fsm_read==FSMR_IDLE && htrans==IDLE  && hready==1'b1  ||
  //            fsm_read==FSMR_READ_P1 &&
  //               (htrans==NONSEQ && hwrite==1'b0 && hsel_0n==1'b0) ||
  //            fsm_read==FSMR_IDLE && hwrite==1'b1)});
  //
  // psl assert ReadFsmPipelineWrap4;
  // psl property FsmReadP4toP5 =
  //   always({fsm_read==FSMR_READ_P4 && htrans==SEQ} |=> {fsm_read==FSMR_READ_P5});
  //
  // psl assert FsmReadP4toP5;
  // ----------
  // psl property FsmReadP4toP4 =
  //   always({fsm_read==FSMR_READ_P4 && htrans==BUSY} |=> {fsm_read==FSMR_READ_P4});
  //
  // psl assert FsmReadP4toP4;
  // ----------
  // psl property FsmReadP5toP6 =
  //   always({fsm_read==FSMR_READ_P5 && htrans==SEQ} |=> {fsm_read==FSMR_READ_P6});
  //
  // psl assert FsmReadP5toP6;
  // ----------
  // psl property FsmReadP5toP5 =
  //   always({fsm_read==FSMR_READ_P5 && htrans==BUSY} |=> {fsm_read==FSMR_READ_P5});
  //
  // psl assert FsmReadP5toP5;
  // ----------
  // psl property FsmReadP6toNext =
  //   always({fsm_read==FSMR_READ_P6} |=>
  //            {(fsm_read==FSMR_READ_P6 && htrans==BUSY)   ||
  //             (fsm_read==FSMR_READ_P1 && htrans==NONSEQ
  //                 && hwrite==1'b0 &  hsel_0n==1'b0) ||
  //             (fsm_read==FSMR_IDLE &&
  //               (htrans==IDLE || hsel_0n==1'b1 ||
  //                htrans==NONSEQ && hwrite==1'b1) )});
  //
  // psl assert FsmReadP6toNext;
  // ----------
  // 7/11/03 Below assertion is incorrect and should be commented out
  //// psl property NeverReadP6andSEQ =
  ////    never ({fsm_read==FSMR_READ_P6 && htrans==SEQ});
  ////
  //// psl assert NeverReadP6andSEQ;
  //
  // ********** Control of idt71v433s10 signals **********
  // psl property MemReadAdvanceControl =
  //    always({(fsm_read==FSMR_READ_P2 && htrans==SEQ) ||
  //          (fsm_read==FSMR_READ_P3 && htrans==SEQ) ||
  //          (fsm_read==FSMR_READ_P4 && htrans==SEQ)} |-> {adv_==1'b0});
  //
  // psl assert MemReadAdvanceControl;
  // ----------
  // ** check load of idt71v433s10 during read, P1, P2, P3
  //  7/05/03 Error in 1st run ??
  //  condition for P2 and P3 needs to be added in RTL
  //  7/11/03 It turns out that it's OK to advance idt71v433s10 on single read
  //   This was done in the implementation, but not in the PSL
  //// psl property IdtReadControlSingle =
  ////    always({qNonSeqReadTransferSingle} |=>
  ////    {( adsp_ == 1'b1 && adsc_ == 1'b0 && adv_ == 1'b1 &&
  ////       lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  ////       cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b1);
  ////     ( adsp_ == 1'b1 && adsc_ == 1'b1 && adv_ == 1'b1 &&
  ////       lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  ////       cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b0);
  ////     ( adsp_ == 1'b1 && adsc_ == 1'b1 && adv_ == 1'b1 &&
  ////       lbo_  == 1'b0 && ce_   == 1'b1 && cs0   == 1'b1 &&
  ////       cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b1)
  ////    });
  // NOTE: this property works only when (htrans!=BUSY) during the read.
  //  This property was changed due to RTL design that took care of don't care conditions
  // 721 <= BUG !!
  // psl property IdtReadControlSingle =
  //    always({qNonSeqReadTransferSingle} |=>
  //    {( adsp_ == 1'b1 && adsc_ == 1'b0 && adv_ == 1'b1 &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b1);
  //     ( adsp_ == 1'b1 && adsc_ == 1'b1 && (adv_ == 1'b1 || adv_==1'b0) &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b0);
  //     ( adsp_ == 1'b1 && adsc_ == 1'b1 && (adv_ == 1'b1 || adv_==1'b0) &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b0)
  //    });
  //
  // psl assert IdtReadControlSingle;
  //
  // ** P1, P2
  // psl property IdtReadControlP1 =
  //    always({fsm_read==FSMR_READ_P1} |->
  //    {( adsp_ == 1'b1 && adsc_ == 1'b0 && adv_ == 1'b1 &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b1)
  //    });
  //
  // psl assert IdtReadControlP1;
  // ----------
  // psl property IdtReadControlP2_3_4 =
  //    always({(fsm_read==FSMR_READ_P2 ||
  //            fsm_read==FSMR_READ_P3 ||
  //            fsm_read==FSMR_READ_P4) && htrans==SEQ} |->
  //    {( adsp_ == 1'b1 && adsc_ == 1'b1 && adv_ == 1'b0 &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b0)
  //    });
  //
  // psl assert IdtReadControlP2_3_4;
  // ----------
  //
  // psl property IdtReadControlP5 =
  //    always({fsm_read==FSMR_READ_P5 && htrans==SEQ} |->
  //    {( adsp_ == 1'b1 && adsc_ == 1'b1 && adv_ == 1'b1 &&
  //      lbo_  == 1'b0 && ce_   == 1'b0 && cs0   == 1'b1 &&
  //      cs1_  == 1'b0 && gw_   == 1'b1 && bwe_ == 1'b1 && oe_==1'b0)
  //    });
  //
  // psl assert IdtReadControlP5;
```

Recommendation:
Use of constants or named sequences for the definition of combinations of signal states is preferred for readability.

```
// ----------
// 7/11/03 error in property IdtReadAdvance, the htrans is not needed.
// //    never ({(FSMR_READ_P2 || FSMR_READ_P3 || FSMR_READ_P4)
// //           htrans==SEQ} |-> {adv_==1'b1});
//
// psl property IdtReadAdvance =
//     always ({(fsm_read==FSMR_READ_P2 ||
//            fsm_read==FSMR_READ_P3 ||
//            fsm_read==FSMR_READ_P4)} |-> {adv_==1'b0});
// BAD   never ({(fsm_read==FSMR_READ_P2 ||
//            fsm_read==FSMR_READ_P3 ||
//            fsm_read==FSMR_READ_P4)} |-> {adv_==1'b1});
//
// psl assert IdtReadAdvance;
//--------------------------------------------------
//| # Pattern name # ERROR HANDLING
//| # Problem #  A slave may have 2 or more idt71v433s10 devices.
//|   However, in some configurations, a device may not be installed.
//|   It is important that the controller provide an ERROR response
//|   if a non-installed memory is addressed
//|   Design assumes that hburst shall be
//|      - SINGLE for writes, and
//|      = SINGLE or WRAP4 for reads
//|   Any other hburst modes should flag an error if device is addressed.
//| AHB LRM requires 2 cycles of Busy.  That needs to be verified.
//|
//| # Motivation # Allow growth in the controller design.
//|    Also, provide feedback to master about an ERROR condition.
//|
//| # Context # Installation of a controller
//|   with less idt71v433s10 devices than what the board can hold.
//|
//| # Solution #
//| - Detect thru a configuration pin or parameter for the number of idt71v433s10
//| devices installed in the board.
//| Provide the ERROR indication per ahb specification if an error occurs.
//| - Ensure when an error occurs:
//|    (hresp==ERROR  && hready==1'b0) followed by
//|    (hresp==ERROR  && hready==1'b1)
//| # Considerations #
//|
// **********
// // 0 -> full board, 1 -> 1 mem unit
// Error in sequence. ~addr[15]
//// psl sequence qAccessToNonPrimamryMemory =
////      {(number_idtmem == 1) && done_xfrn==1'b0 && htrans==NONSEQ && hsel_1n==1'b0 };
// psl sequence qAccessToNonPrimamryMemory =
//      {(number_idtmem == 1) && done_xfrn==1'b0 && htrans==NONSEQ &&
//                              hsel_0n==1'b0 && addr[15]==1'b1 };
//
// psl property MemCountErrorDetect =
//    always ({qAccessToNonPrimamryMemory} |->
//                          {hready==1'b0; hresp==ERROR && hready==1'b1});
//
// psl assert MemCountErrorDetect;
```

```
//----------
// psl sequence qNonSeqWriteTransferNonSingle =
//              {done_xfrn==1'b0 && htrans==NONSEQ &&
//               hburst!=SINGLE && hwrite==1'b1 && hsel_0n==1'b0 };
//
// psl property HrespErrorForNonSingleWriteTransfer =
//   always({qNonSeqWriteTransferNonSingle} |-> {hresp==ERROR && hready==1'b1});
//
// psl assert HrespErrorForNonSingleWriteTransfer;
// ----------
// psl sequence qNonSeqReadTransferNotSingleNotWrap4 =
//          {htrans==NONSEQ && hwrite==1'b0 && hsel_0n==1'b0 &&
//          !(hburst==SINGLE | hburst==WRAP4)};
//
// psl property NonSeqReadTransferNotSingleNotWrap4 =
//   always({qNonSeqReadTransferNotSingleNotWrap4} |-> {hresp==ERROR && hready==1'b1});
//
// psl assert NonSeqReadTransferNotSingleNotWrap4;
// ----------
//  psl property HrespToNonseqErrorIsTwoCycles=
//   always({(hresp==ERROR) && (htrans==NONSEQ)} |->
//           {hready==1'b0; (hready==1'b1) && (hresp==ERROR)});
//
//| # Pattern name # TAGGED TRANSACTION
//| # Problem #   Ensure that transactions consisting of multiple-out-of-order
//|   write responses match the appropriate read responses.
//|
//| # Motivation #  Ease of error detection during debug
//| # Context #  This pattern is useful because of potential memory
//|    write or read error in single or burst mode, particularly with rollover.
//| # Solution #
//|    1. A vector shall keep track if data was written into the memory
//|    2. For 1st idt71v433s10 memory:
//|       Data written to memory shall be {1'b0,idt_addr[14:0], 1'b0, idt_addr[14:0]}
//|      2nd idt71v433s10 memory data=
//|          {1'b1,idt_addr[14:0], 1'b1, idt_addr[14:0]}
//|    3. If data is written, then read data[idt_addr] must equal predefined pattern.
//|    4. Cycle timing indicates that at
//|       FSMR_READ_P3, FSMR_READ_P4, FSMR_READ_P5, FSMR_READ_P6 the
//|       hwdata is for address x located in register addr.
//|
//| # Considerations #  For formal verification, ignore this assumption.
//|
//  psl property ReadDataToHrData =
//   always ((hready==1'b1 &&
//          fsm_read==FSMR_READ_P3) -> datawritten[addr3]==1'b1 -> hrdata[7:0]==addr3[7:0]);
//
// 7/09/03 new 7/16, fixed range for datawritten address in lookup
// Need to add the case where datawritten[addr] != 1'b1
//  psl property ReadDataWithNothingWritten=
//   always ((hready==1'b1 &&
//          fsm_read==FSMR_READ_P3) -> datawritten[addr3[addr_msb+2:2]]!==1'b1 -> 1'b0);
//  BAD  never ((hready==1'b1 &&
//          fsm_read==FSMR_READ_P3) -> datawritten[addr3[addr_msb+2:2]]!==1'b1);
// psl assert ReadDataWithNothingWritten;
```

```
  // 7/08/03 Problem reading memory.  Need to add assertions that data read is not X
  // psl property ReadDataIsX =
  //   never {qNonSeqReadTransfer; true; idt_data !== 32'bx};
  // BAD  never({qNonSeqReadTransfer} |=> {true; idt_data == 32'bx});
  // assert ReadDataIsX;

// psl property AhbDataToIDT =
 //   always ({fsm_write==FSMW_WRITE} |-> {idt_data==hwdata});
 //  assert AhbDataToIDT;
```

The check can be generalized to catch any bit that is X with the use of the forall operator.
Also, a check for floating (Z) values could be added.

Figure 8.1.5-2 ***PSL*** **Description of Design Properties (ch8/ahb95/mem_slave.v)**

8.1.6 Write RTL

Once the *PSL* properties for the RTL design are written and reviewed, the RTL code can be written. Figures 8.1.6-1a and 8.1.6-1b represent a graphical view of the major RTL blocks as automatically drawn by Cadence *Incisive™ unified simulator*[6]. Figures 8.1.6-2 and 8.1.6-3 represent the FSM state machines automatically extracted by @HDL *atdesigner.*[7] The RTL code is available in the following files:

1. ch8/ahb01/mem_slave.v   // Verilog 2001
2. ch8/ahb95/mem_slave.v   // Verilog 1995

The RTL (ahb95/mem_slave.v) model was compiled with Synplicity *Synplify Pro*. Target technology was set to QuickLogic pASIC1. The following a short summary of the synthesis results:

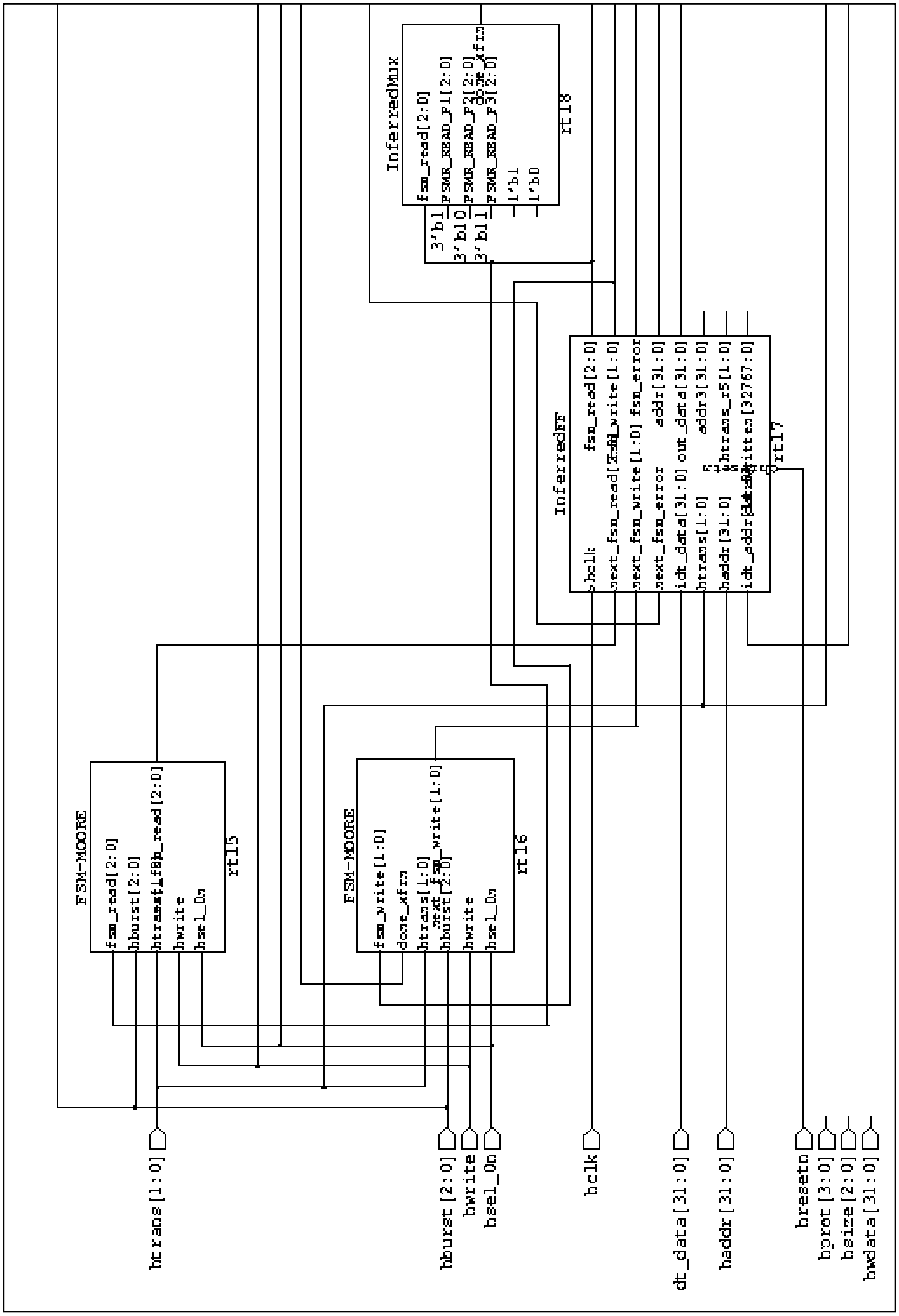
```
\ahb95\mem_slave.v":135:15:135:28|No assignment to
burst_advcount\ahb95\mem_slave.v":68:19:68:24|Input hwdata is unused
\ahb95\mem_slave.v":72:28:72:32|Input hsize is unused
\ahb95\mem_slave.v":74:18:74:22|Input hprot is unused
Performance Summary
*******************
Worst slack in design: -9.467
                 Requested   Estimated   Requested  Estimated          Clock      Clock
Starting Clock   Frequency   Frequency   Period     Period      Slack  Type       Group
------------------------------------------------------------------------------------------------
mem_slave|hclk   25.0 MHz    20.2 MHz    40.000     49.467      -9.467  inferred   default_clkgroup
================================================================================================
```

The schematic diagram demonstrates that the RTL includes two FSMs, one for the READ of memory data, and another one for the WRITE of data to the memory.

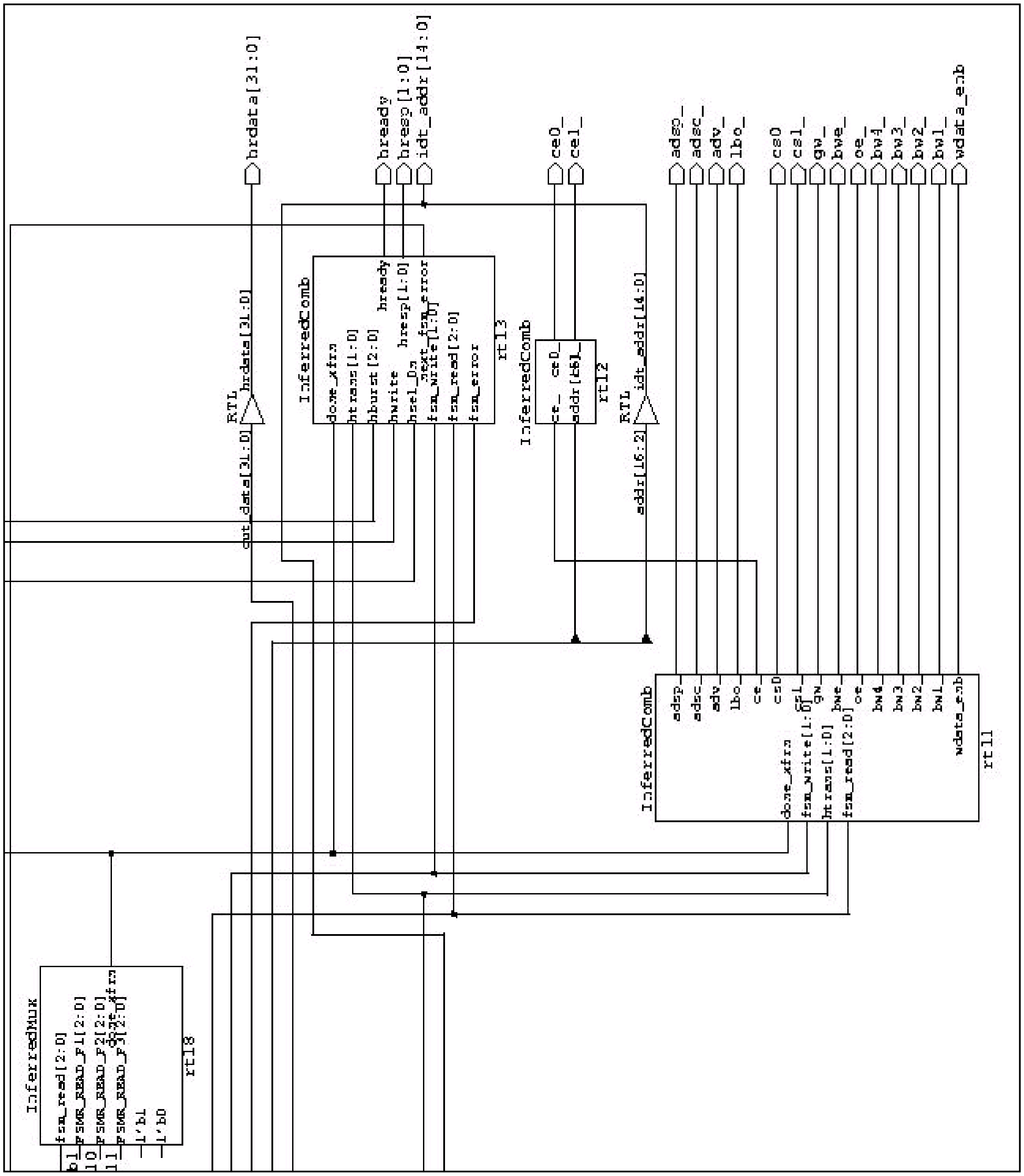
---

[6] Many tools can extract and draw architectural blocks from an RTL description. Cadence *Incisive™ unified simulator* was used because it was a tool available to the author. www.cadence.com

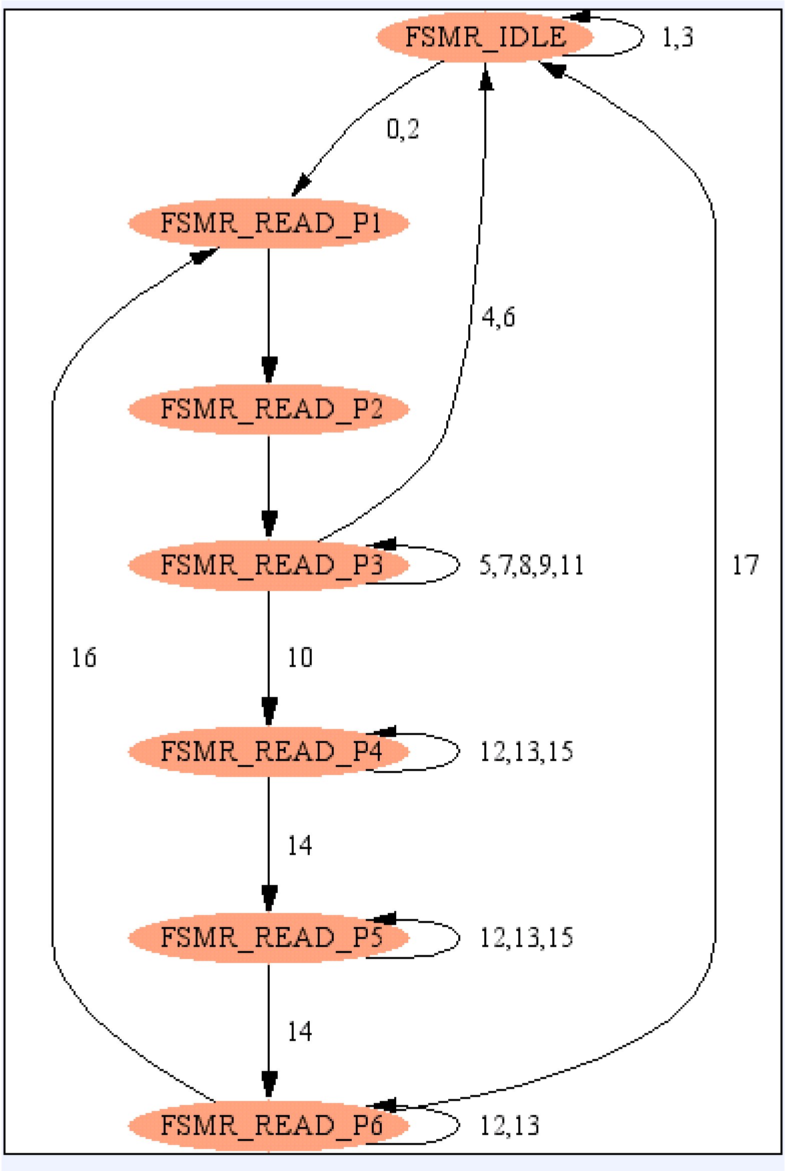
[7] Many tools can extract and draw FSMs from an RTL description. *atdesigner* was used because it was a tool available to the author. www.athdl.com



Figures 8.1.6-1a Memory Slave Interface Graphical View

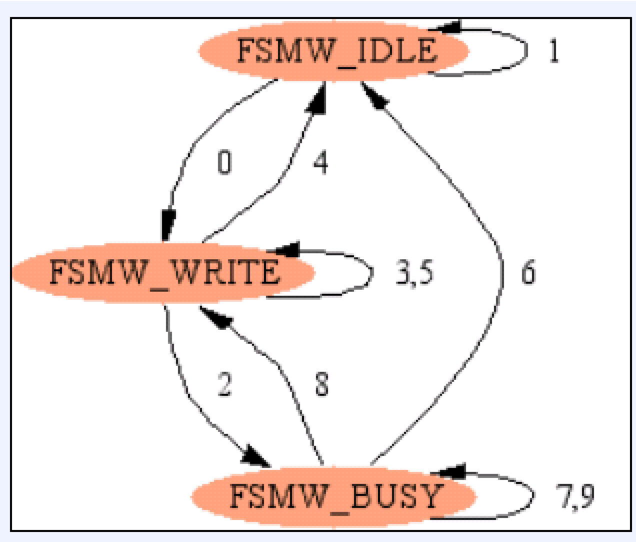


Figures 8.1.6-1b Memory Slave Interface Graphical View
(Drawn by Cadence *Incisive™ unified simulator*)



```
Table Fsm states for module mem_slave in File mem_slave.v
-----------------------
Name              Status
-----------------------
FSMR_IDLE         ---
FSMR_READ_P1      ---
FSMR_READ_P2      ---
FSMR_READ_P3      ---
FSMR_READ_P4      ---
FSMR_READ_P5      ---
FSMR_READ_P6      ---
```

```
Table Fsm transition for module mem_slave in File mem_slave.v
-----------------------------------------------------------------------------------------
--
State transition                 Transition ID     Condition
-----------------------------------------------------------------------------------------
--
FSMR_IDLE -> FSMR_READ_P1        0                 (((hburst == SINGLE) && (htrans ==
                                                   NONSEQ)) && (hwrite == 1'b0)) &&
                                                   (hsel_0n == 1'b0))
FSMR_IDLE -> FSMR_READ_P1        2                 (((hburst == WRAP4) && (htrans ==
                                                   NONSEQ)) && (hwrite == 1'b0)) &&
                                                   (hsel_0n == 1'b0))
FSMR_IDLE -> FSMR_IDLE           1                 !(((hburst == SINGLE) && (htrans ==
                                                   NONSEQ)) && (hwrite == 1'b0)) &&
                                                   (hsel_0n == 1'b0))
FSMR_IDLE -> FSMR_IDLE           3                 !(((hburst == WRAP4) && (htrans ==
                                                   NONSEQ)) && (hwrite == 1'b0)) &&
                                                   (hsel_0n == 1'b0))
FSMR_READ_P1 -> FSMR_READ_P2     -                 UnConditional
FSMR_READ_P2 -> FSMR_READ_P3     -                 UnConditional
FSMR_READ_P3 -> FSMR_IDLE        4                 ((htrans == NONSEQ) && (hwrite ==
                                                   1'b0)) && (hsel_0n == 1'b0))
FSMR_READ_P3 -> FSMR_IDLE        6                 ((htrans == IDLE) || (((hburst ==
                                                   SINGLE) && (htrans == NONSEQ)) &&
                                                   (hwrite  ==  1'b1))  &&  (hsel_0n  ==
1'b0)))
FSMR_READ_P3 -> FSMR_READ_P3     5                 !((htrans == NONSEQ) && (hwrite ==
                                                   1'b0)) && (hsel_0n == 1'b0))
FSMR_READ_P3 -> FSMR_READ_P3     7                 !((htrans == IDLE) || (((hburst ==
                                                   SINGLE) && (htrans == NONSEQ)) &&
                                                   (hwrite  ==  1'b1))  &&  (hsel_0n  ==
1'b0)))
FSMR_READ_P3 -> FSMR_READ_P3     8                 (((hburst == WRAP4) && (htrans ==
                                                   BUSY))  &&  (hwrite  ==  1'b0))  &&
(hsel_0n
                                                   == 1'b0))
FSMR_READ_P3 -> FSMR_READ_P3     9                 !(((hburst == WRAP4) && (htrans ==
                                                   BUSY))  &&  (hwrite  ==  1'b0))  &&
(hsel_0n
                                                   == 1'b0))
FSMR_READ_P3 -> FSMR_READ_P3     11                !(((hburst == WRAP4) && (htrans ==
                                                   SEQ)) && (hwrite == 1'b0)) && (hsel_0n
                                                   == 1'b0))
FSMR_READ_P3  ->  FSMR_READ_P4     10                  (((hburst  ==  WRAP4)  &&  (htrans  ==
SEQ))
                                                   && (hwrite == 1'b0)) && (hsel_0n ==
                                                   1'b0))
FSMR_READ_P4 -> FSMR_READ_P4     12                (htrans == BUSY)
FSMR_READ_P4 -> FSMR_READ_P4     13                !(htrans == BUSY)
FSMR_READ_P4 -> FSMR_READ_P4     15                !(htrans == SEQ)
FSMR_READ_P4 -> FSMR_READ_P5     14                (htrans == SEQ)
FSMR_READ_P5 -> FSMR_READ_P5     12                (htrans == BUSY)
FSMR_READ_P5 -> FSMR_READ_P5     13                !(htrans == BUSY)
FSMR_READ_P5 -> FSMR_READ_P5     15                !(htrans == SEQ)
FSMR_READ_P5 -> FSMR_READ_P6     14                (htrans == SEQ)
FSMR_READ_P6 -> FSMR_READ_P6     12                (htrans == BUSY)
FSMR_READ_P6 -> FSMR_READ_P6     13                !(htrans == BUSY)
FSMR_READ_P6  ->  FSMR_READ_P1     16                  ((htrans  ==  NONSEQ)  &&  ((hwrite  ==
1'b0)
                                                   & (hsel_0n == 1'b0)))
FSMR_READ_P6 -> FSMR_IDLE        17                !((htrans == NONSEQ) && ((hwrite ==
                                                   1'b0) & (hsel_0n == 1'b0)))
```



```
Table Fsm states for module mem_slave in File mem_slave.v
---------------------
Name           Status
---------------------
FSMW_IDLE      ---
FSMW_WRITE     ---
FSMW_BUSY      ---


Table Fsm transition for module mem_slave in File mem_slave.v
---------------------------------------------------------------------------------------
State transition             Transition ID     Condition
---------------------------------------------------------------------------------------
FSMW_IDLE -> FSMW_WRITE      0                 ((((done_xfrn == 1'b0) && (htrans ==
                                               NONSEQ)) && (hburst == SINGLE)) &&
                                               (hwrite == 1'b1)) && (hsel_0n == 1'b0))
FSMW_IDLE -> FSMW_IDLE       1                 !(((((done_xfrn == 1'b0) && (htrans ==
                                               NONSEQ)) && (hburst == SINGLE)) &&
                                               (hwrite == 1'b1)) && (hsel_0n == 1'b0))
FSMW_WRITE -> FSMW_BUSY      2                 ((fsm_write == FSMW_WRITE) && (htrans ==
                                               BUSY))
FSMW_WRITE -> FSMW_WRITE     3                 !((fsm_write == FSMW_WRITE) && (htrans
                                               == BUSY))
FSMW_WRITE -> FSMW_WRITE     5                 !((htrans == IDLE) || ((done_xfrn ==
                                               1'b0) && (hwrite == 1'b0)))
FSMW_WRITE -> FSMW_IDLE      4                 ((htrans == IDLE) || ((done_xfrn ==
                                               1'b0) && (hwrite == 1'b0)))
FSMW_BUSY -> FSMW_IDLE       6                 (htrans == IDLE)
FSMW_BUSY -> FSMW_BUSY       7                 !(htrans == IDLE)
FSMW_BUSY -> FSMW_BUSY       9                 !(((((done_xfrn == 1'b0) && (hwrite ==
                                               1'b1)) && (hsel_0n == 1'b0)) && (htrans
                                               == NONSEQ)) && (fsm_write == FSMW_BUSY))
FSMW_BUSY -> FSMW_WRITE      8                 ((((done_xfrn == 1'b0) && (hwrite ==
                                               1'b1)) && (hsel_0n == 1'b0)) && (htrans
                                               == NONSEQ)) && (fsm_write == FSMW_BUSY))
```

Figures 8.1.6-2 FSM READ Extracted State Machines, @HDL *@designer*.

8.1.7 Write Testbench

Figure 8.1.7-1 represents a high level view of the test environment for verification through simulation. Design verification techniques and testbench designs are described in several books.[8] The AHB testbench utilizes a transaction-based approach. Less emphasis is placed in this book on describing the testbench environment and on creating an extensive set of test vectors (directed and pseudo-random) because the purpose of this book is to teach *PSL* as a language, along with the methodologies and guidelines in using *PSL* for simulation and formal verification. This AHB model is just a vehicle to demonstrate the concepts in the application of *PSL*. This model is by no means a representation of an IP of a slave interface, and the accuracy of the model is not guaranteed.

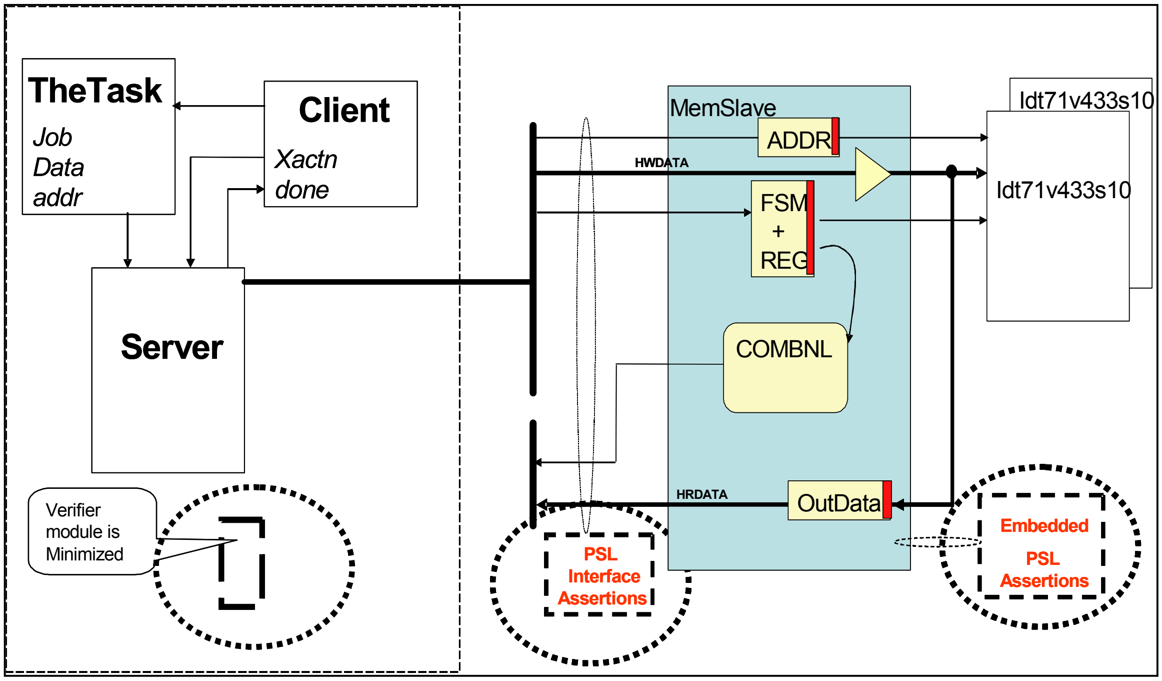


Figure 8.1.7-1 AMBA AHB High level View of the Testbench Environment

Figure 8.1.7-2 represents a more detailed view of the server interface connected to the *PSL* compliance model for verification of the AHB bus interface.

---

[8] * Writing Testbenches: Functional Verification of HDL Models, Second Edition, ISBN 1-4020-7401-8
* Real Chip Design and Verification Using Verilog and VHDL, 2002 isbn 0-9705394-2-8
* Component Design by Example ", 2001 isbn 0-9705394-0-1

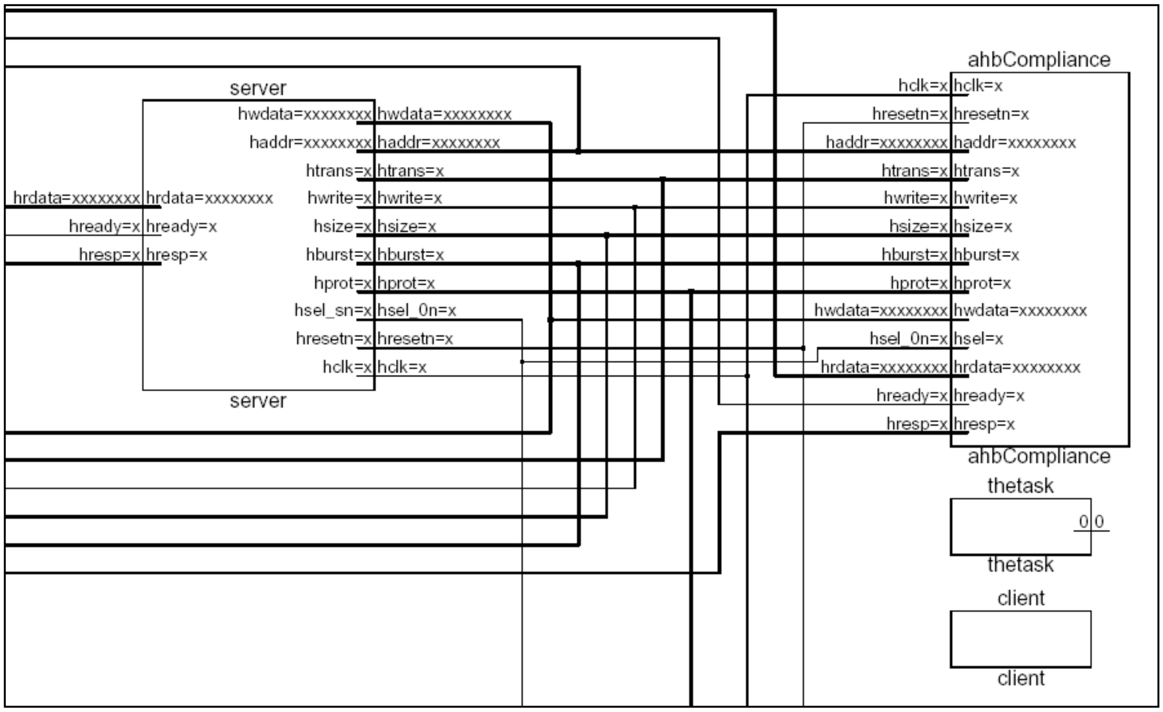


Figure 8.1.7-2 Detailed View of Server *PSL* AHB Bus Compliance Model

8.1.8 Simulate and Evaluate Design

One of the main advantages of *PSL* is the capacity to verify that the design meets the requirements, as defined by *PSL* properties. Several vendors have integrated *PSL* into the simulation environment either through close integration into the simulator, or through the translation of *PSL* into simulatable HDL modules. *PSL* is also used in formal verification, as addressed in Chapter 7.

Figures 8.1.8-1 and 8.1.8-2 demonstrate some simulation timing diagrams of the AHB interface. Figure 8.1.8-3 provides a sample of the assertion metrics as generated by Cadence Incisive™ unified simulator. Figure 8.1.8-4 demonstrates the detection of an actual design error (ch8/error/mem_slave.v) through *PSL*. Figure 8.1.8-5 provides an explanation of the RTL error. The error involves the return of the READ FSM to the IDLE state, as expressed in *PSL,* as one the possible states following the FSM READ P6 state. The RTL design was in error, as the FSM remained in the P6 state for an additional cycle.

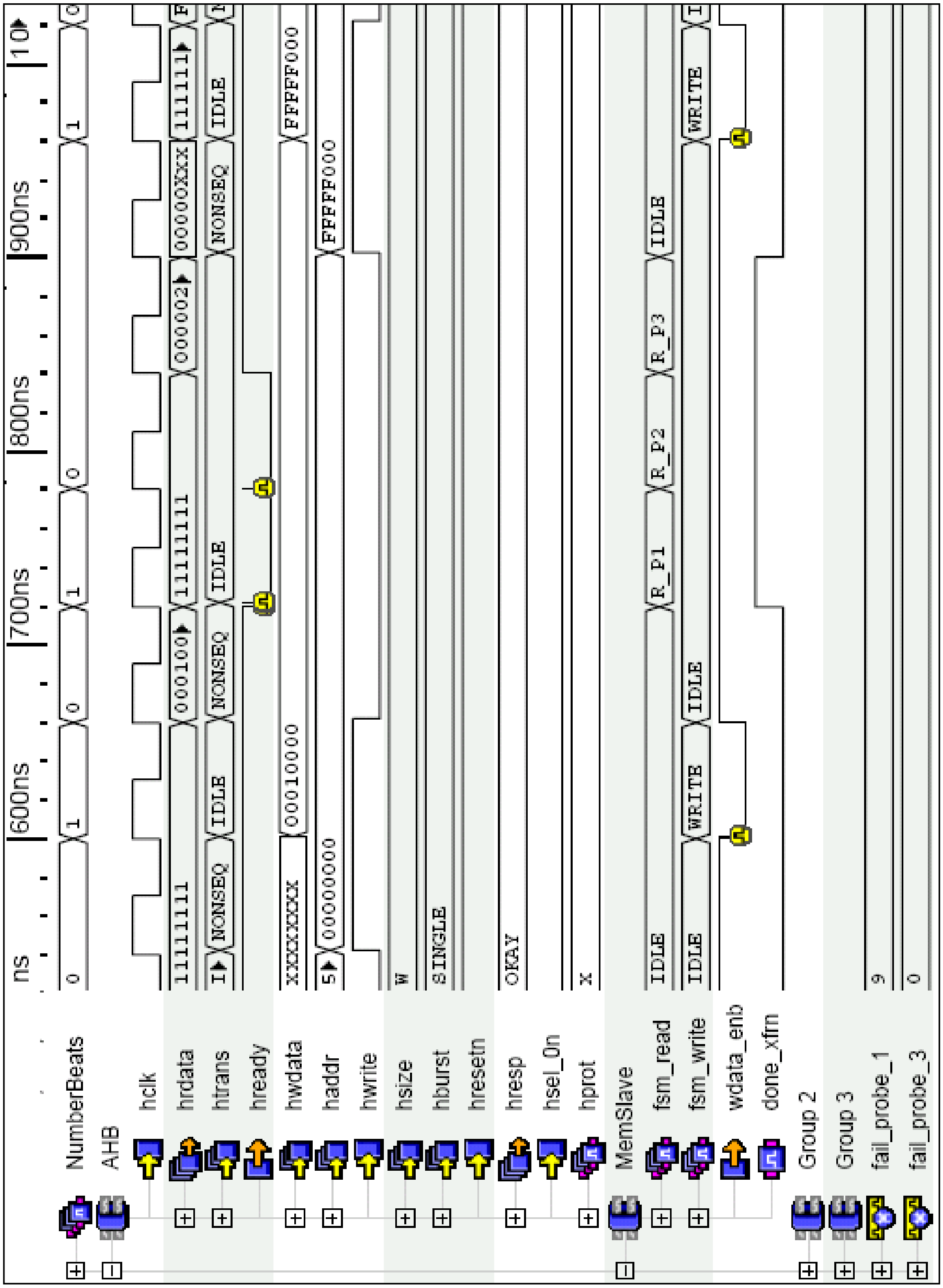


Figure 8.1.8-1 AHB Simulation, Read/Write One word, Cadence *Incisive™ unified simulator*

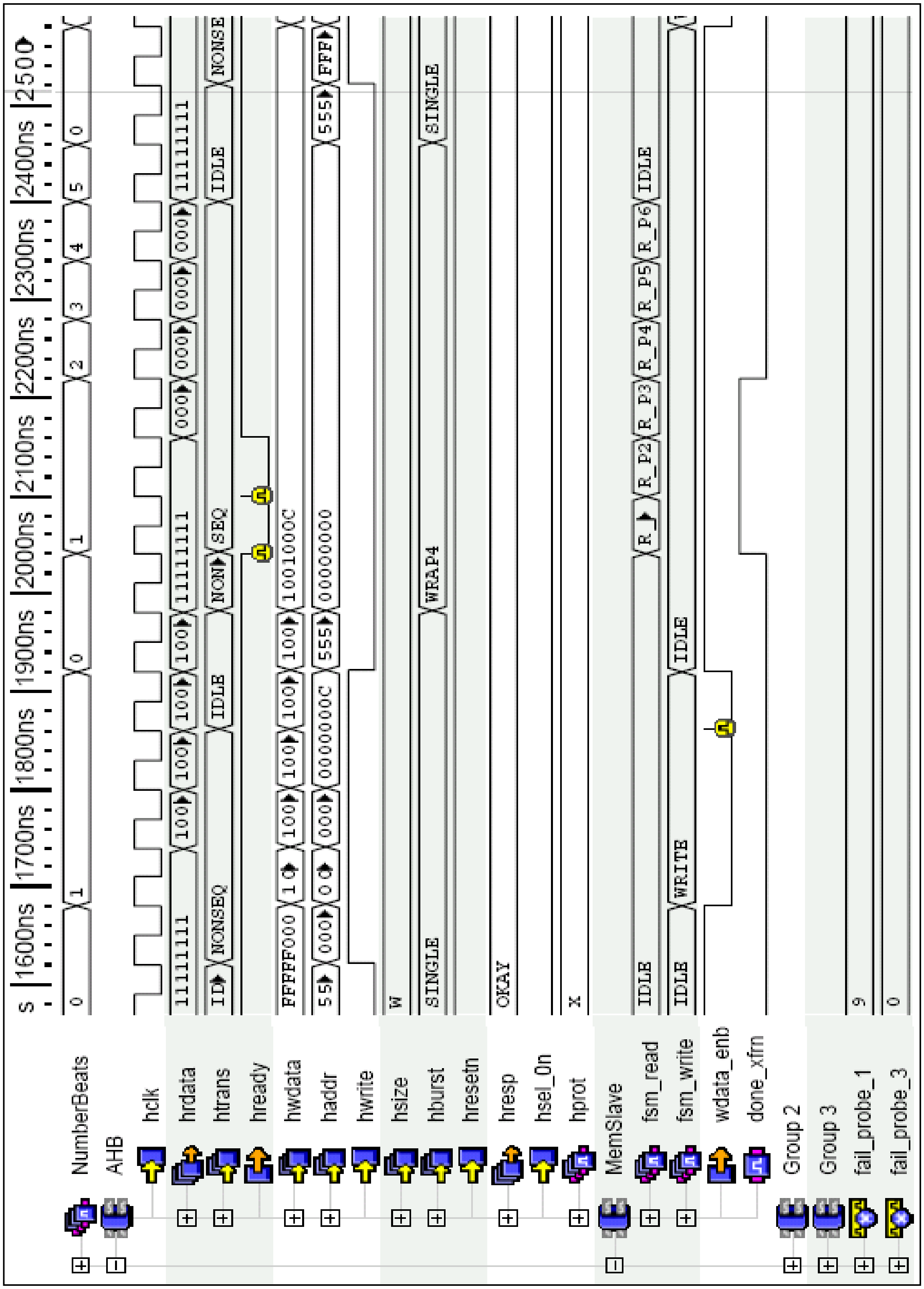
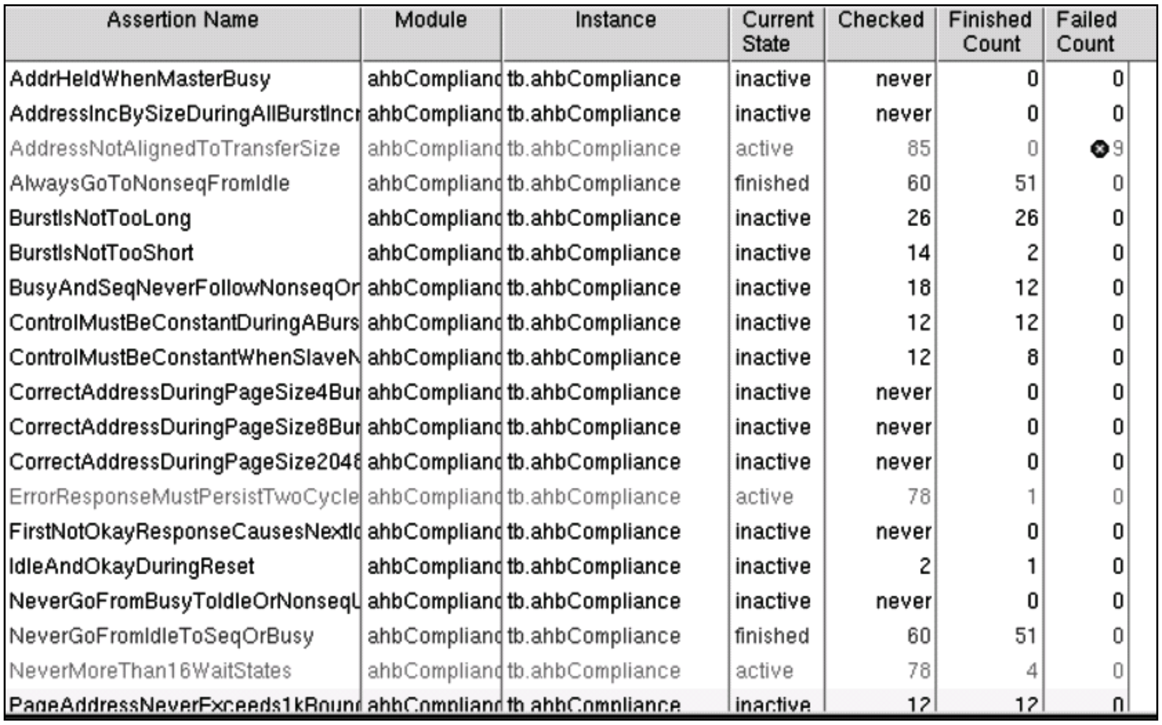
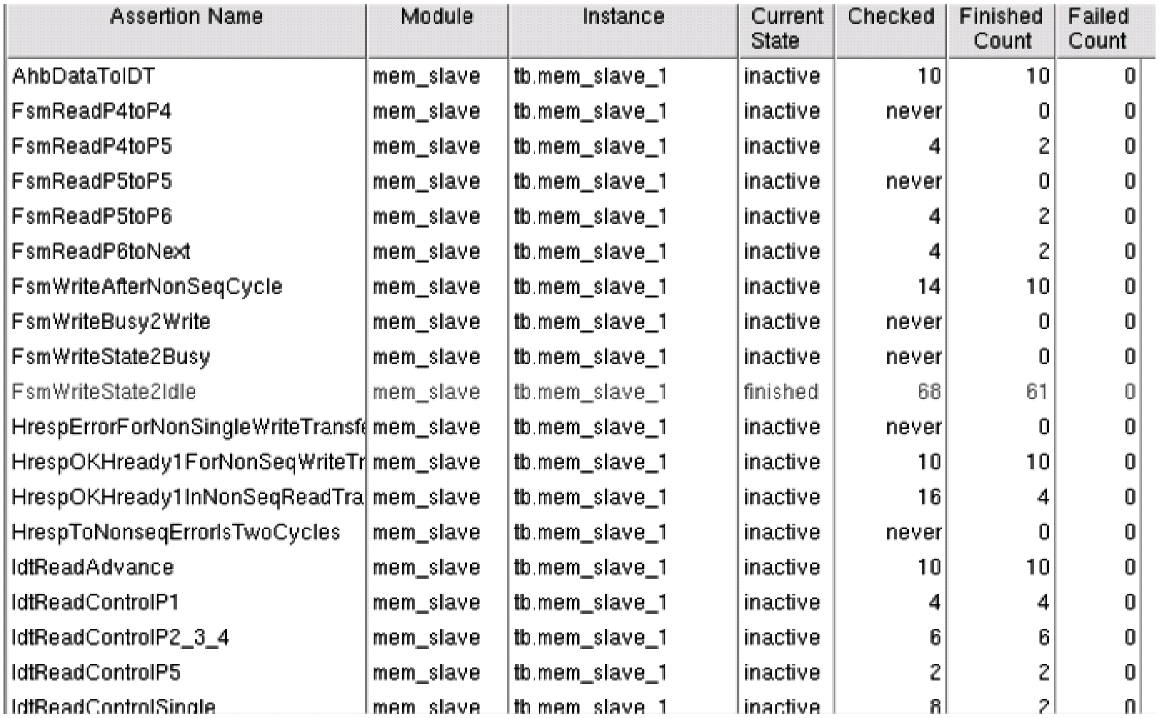


Figure 8.1.8-2 AHB Simulation for Read Four words, Cadence *Incisive™ unified simulator*



Assertion Name	Module	Instance	Current State	Checked	Finished Count	Failed Count
AddrHeldWhenMasterBusy	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
AddressIncBySizeDuringAllBurstIncr	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
AddressNotAlignedToTransferSize	ahbComplianc	tb.ahbCompliance	active	85	0	✖ 9
AlwaysGoToNonseqFromIdle	ahbComplianc	tb.ahbCompliance	finished	60	51	0
BurstIsNotTooLong	ahbComplianc	tb.ahbCompliance	inactive	26	26	0
BurstIsNotTooShort	ahbComplianc	tb.ahbCompliance	inactive	14	2	0
BusyAndSeqNeverFollowNonseqOr	ahbComplianc	tb.ahbCompliance	inactive	18	12	0
ControlMustBeConstantDuringABurs	ahbComplianc	tb.ahbCompliance	inactive	12	12	0
ControlMustBeConstantWhenSlaveN	ahbComplianc	tb.ahbCompliance	inactive	12	8	0
CorrectAddressDuringPageSize4Bur	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
CorrectAddressDuringPageSize8Bur	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
CorrectAddressDuringPageSize2048	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
ErrorResponseMustPersistTwoCycle	ahbComplianc	tb.ahbCompliance	active	78	1	0
FirstNotOkayResponseCausesNextId	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
IdleAndOkayDuringReset	ahbComplianc	tb.ahbCompliance	inactive	2	1	0
NeverGoFromBusyToIdleOrNonseqU	ahbComplianc	tb.ahbCompliance	inactive	never	0	0
NeverGoFromIdleToSeqOrBusy	ahbComplianc	tb.ahbCompliance	finished	60	51	0
NeverMoreThan16WaitStates	ahbComplianc	tb.ahbCompliance	active	78	4	0
PageAddressNeverExceeds1kBound	ahbComplianc	tb.ahbCompliance	inactive	12	12	0



Assertion Name	Module	Instance	Current State	Checked	Finished Count	Failed Count
AhbDataToIDT	mem_slave	tb.mem_slave_1	inactive	10	10	0
FsmReadP4toP4	mem_slave	tb.mem_slave_1	inactive	never	0	0
FsmReadP4toP5	mem_slave	tb.mem_slave_1	inactive	4	2	0
FsmReadP5toP5	mem_slave	tb.mem_slave_1	inactive	never	0	0
FsmReadP5toP6	mem_slave	tb.mem_slave_1	inactive	4	2	0
FsmReadP6toNext	mem_slave	tb.mem_slave_1	inactive	4	2	0
FsmWriteAfterNonSeqCycle	mem_slave	tb.mem_slave_1	inactive	14	10	0
FsmWriteBusy2Write	mem_slave	tb.mem_slave_1	inactive	never	0	0
FsmWriteState2Busy	mem_slave	tb.mem_slave_1	inactive	never	0	0
FsmWriteState2Idle	mem_slave	tb.mem_slave_1	finished	68	61	0
HrespErrorForNonSingleWriteTransfe	mem_slave	tb.mem_slave_1	inactive	never	0	0
HrespOKHready1ForNonSeqWriteTr	mem_slave	tb.mem_slave_1	inactive	10	10	0
HrespOKHready1InNonSeqReadTra	mem_slave	tb.mem_slave_1	inactive	16	4	0
HrespToNonseqErrorIsTwoCycles	mem_slave	tb.mem_slave_1	inactive	never	0	0
IdtReadAdvance	mem_slave	tb.mem_slave_1	inactive	10	10	0
IdtReadControlP1	mem_slave	tb.mem_slave_1	inactive	4	4	0
IdtReadControlP2_3_4	mem_slave	tb.mem_slave_1	inactive	6	6	0
IdtReadControlP5	mem_slave	tb.mem_slave_1	inactive	2	2	0
IdtReadControlSingle	mem_slave	tb.mem_slave_1	inactive	8	2	0

Figure 8.1.8-3 Sample Assertion Metrics,
Cadence *Incisive™ unified simulator*

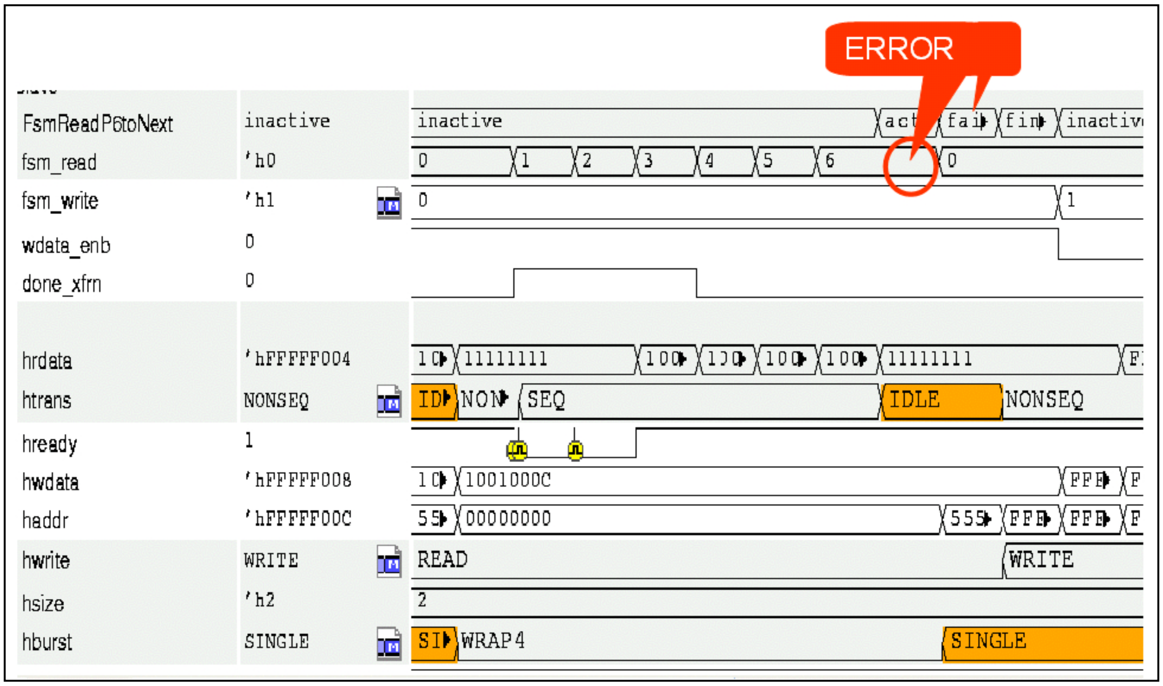
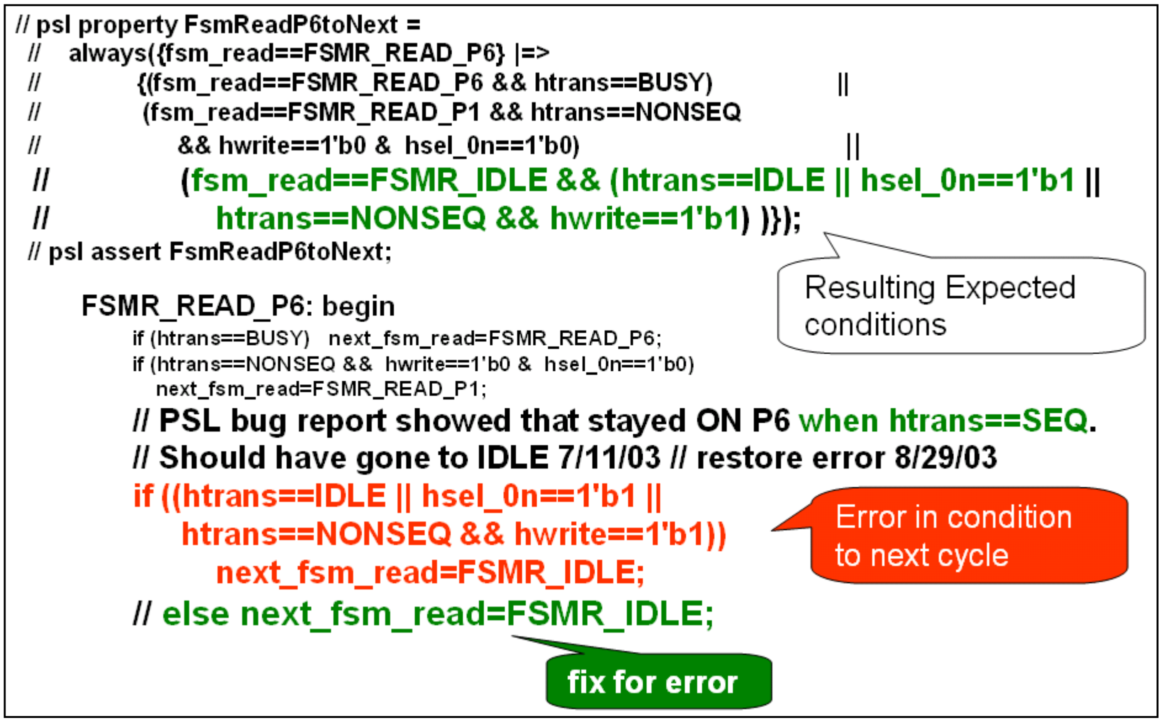


Figure 8.1.8-4 Error Detection, Cadence *Incisive™ unified simulator*



```
// psl property FsmReadP6toNext =
//    always({fsm_read==FSMR_READ_P6} |=>
//           {(fsm_read==FSMR_READ_P6 && htrans==BUSY)            ||
//            (fsm_read==FSMR_READ_P1 && htrans==NONSEQ
//               && hwrite==1'b0 &  hsel_0n==1'b0)                 ||
//            (fsm_read==FSMR_IDLE && (htrans==IDLE || hsel_0n==1'b1 ||
//               htrans==NONSEQ && hwrite==1'b1) )});
// psl assert FsmReadP6toNext;

     FSMR_READ_P6: begin
          if (htrans==BUSY)   next_fsm_read=FSMR_READ_P6;
          if (htrans==NONSEQ && hwrite==1'b0 & hsel_0n==1'b0)
            next_fsm_read=FSMR_READ_P1;
          // PSL bug report showed that stayed ON P6 when htrans==SEQ.
          // Should have gone to IDLE 7/11/03 // restore error 8/29/03
          if ((htrans==IDLE || hsel_0n==1'b1 ||
              htrans==NONSEQ && hwrite==1'b1))
                next_fsm_read=FSMR_IDLE;
          // else next_fsm_read=FSMR_IDLE;
```

Resulting Expected conditions

Error in condition to next cycle

fix for error

Figure 8.1.8-5 *PSL* Code versus RTL Code with Error

8.2 REFLECTION ON USING *PSL*

ABV with *PSL* demonstrates a paradigm shift in the design process. Specifically, ABV with *PSL* is moving the traditional design process from an RTL design approach with no executable documentation to a process that provides the following benefits:

1) *PSL* can be used to define a formal requirements specification. This was demonstrated in this chapter with the interface-level properties of the design.
2) Addresses and documents design decisions.
3) Documents design properties and assumptions using a formal property language.
4) Addresses solutions (e.g., interfaces, implied FSMs) to requirements prior to any RTL code.
5) Addresses verification assertions and functional coverage, which guide the design and verification.
6) Provides excellent basis for design and verification reviews.
7) Simplifies design of testbench verifier.
8) Guides testbench vectors for conditions to be addressed.
9) Detects errors at the white-box level during simulation.
10) Provides quick feedback as to the accuracy of the RTL model using formal verification, without access to a testbench (see Chapter 7).

It is important to note that *PSL* addresses “functional operation” rather than design. This is in contrast to RTL that addresses design, “drivers” of control signals, and FSM architectures. *PSL* is implementation independent. *PSL* presents a different viewpoint of the design. *PSL* may imply FSMs in the implementation. *PSL* does not necessarily show any design optimizations, such as the use of don't-care conditions. As the design matures, it may be necessary to revisit the *PSL* assertions, as they may be too restrictive. In addition, it may also be necessary to add assertions defined at the functional level. But this experience of tuning the assertions and the design is healthy because it forces users to delve into the requirements and implementation.

Using *PSL* for the front-end design definition work demonstrated that *PSL* is very powerful in the process of delving into design requirements, design architecture, and definition of restrictions imposed by the architecture. *PSL* is more expressive and precise than a natural language (e.g., English) for these tasks. The RTL and verification tasks are greatly simplified as a result of using this assertion-based methodology with *PSL* because it alleviated the need to write a complex verifier model prior to debugging the design. During simulation *PSL* immediately detected design and testbench errors. A possibly frustrating aspect in using *PSL* with dynamic and formal verification is the tool’s (e.g., simulation or FV) reluctance to provide approval of failed properties when the designer truly believes that the RTL is correct (when it really is an error).