ECE-560, Fall 2024
Assertion Based Verification

Asst. Prof of Practice: Venkatesh Patil

Maseeh College of Engineering and Computer Science

Final Project Report
AHB2APB BRIDGE

Team Number 3

Yashaswi Katne | katne@pdx.edu

Maheswar Reddy Kamalapuram | maheswar@pdx.edu

Naveen Kumar Reddy Thummala | naveenkt@pdx.edu

Prasanna Kumar Panchada | prapanch@pdx.edu

Table of Contents

Introduction:

The aim of this project and the associated verification plan is to outline the approaches, methodologies, and results derived from our efforts in assessing the effectiveness and performance attributes of the AHB-APB bridge design. This bridge serves as a vital intermediary, which enables seamless communication between the high-performance Advanced High-performance Bus (AHB) and the power-efficient Advanced Peripheral Bus (APB).

As semiconductor technology continues to evolve, there is a growing need for accuracy and reliable, the need for verification processes has become increasingly vital. In a design involving high-speed data transfer, memory mapping, and peripheral control, the verification of the bridge requires an integrated approach. This paper presents our step-by-step strategy for functional testing of the AHB-APB bridge for its conformance to the requirements specified by the ARM AMBA specification sheet. Our experience in this verification process has been challenging and enlightening. We tried to define assertions, assumptions, and coverage properties to check the design.

Project Proposal:

The goal of our project, which involved the principles of assertion-based verification, was to create and apply a set of comprehensive validation for the functionality of the AHB-to-APB (AHB2APB) bridge design. This was inspired by our commitment to adhere to the set ARM AMBA standards that define the nature of this critical interface. During this project, our group engaged in crafting a set of properties derived directly from the bridge's design specifications. We developed an array of assertions, assumptions, and cover properties in parallel to perform extensive testing of various functionalities provided by the bridge. This effort was not easy in the beginning.

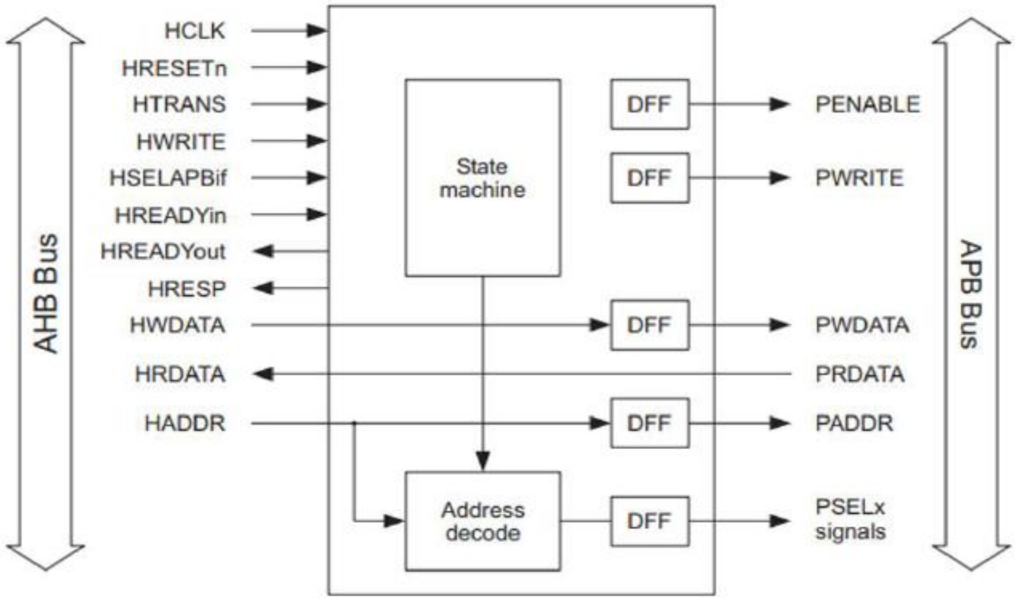
Specifically, we faced several major obstacles at the start of the above effort. The foremost of these issues involved obtaining a proper RTL (Register Transfer Level) code that met all our specified criteria. In other words, we were looking for RTL facilitated the burst read and write transactions as specified for the ARM AMBA standards. In the light of these challenges, we decided to proceed with the RTL code available for the AHB2APB bridge design, as an integral part of this process.

This has been presented as an annex to this report for quick reference. The RTL code selected can also be accessed from: https://github.com/prajwalgekkouga/AHB-to-APB-Bridge.

BUS BRIDGE DESCRIPTION:

The AHB2APB bridge developed by ARM acts as a crucial interface between the fast-paced AHB and the power-efficient APB. In simpler words, this can be called a bridge that offers compatibility to a wide range of read-write transactions in both buses. Amongst the key ingredients for such a bridge are a slave bus interface, an APB transfer state machine that operates independently of the memory architecture of a device, and methods for providing APB output signals are provided.

The AHB2APB bridge is basically designed to preserve the addresses, control signals, and data originating from the AHB and then routes this to the APB peripherals while, at the same time, sends data along with a response signal back to the AHB. The bridge manages the APB data bus using two different channels: the read data path PRDATA and the write data path. It is designed to handle sequential and nonsequential data transfers of various sizes: PWDATA and HSIZE. This enables a runtime communication interface between the high-speed and low-speed buses.



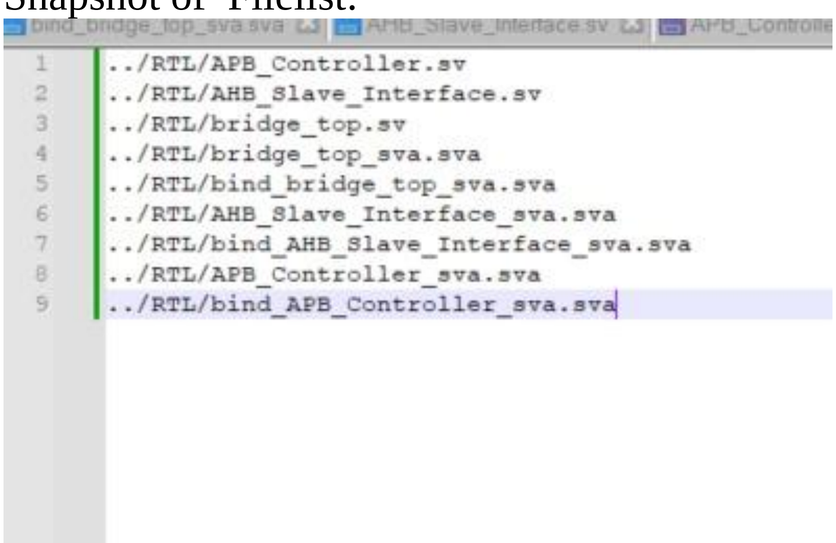
PROJECT COLLABORATION:

The team worked collaboratively, and the following file structure was set up to effectively manage the project. It was agreed that a GitHub repository would be used in order to manage our RTL, together with all associated properties, assertions, assumptions, and coverage points. This structure allowed each member of the team to work on their own properties independently yet still have access to.

Drawing on the work of colleagues, we created separate branches for each member of the team and populated these with template files.

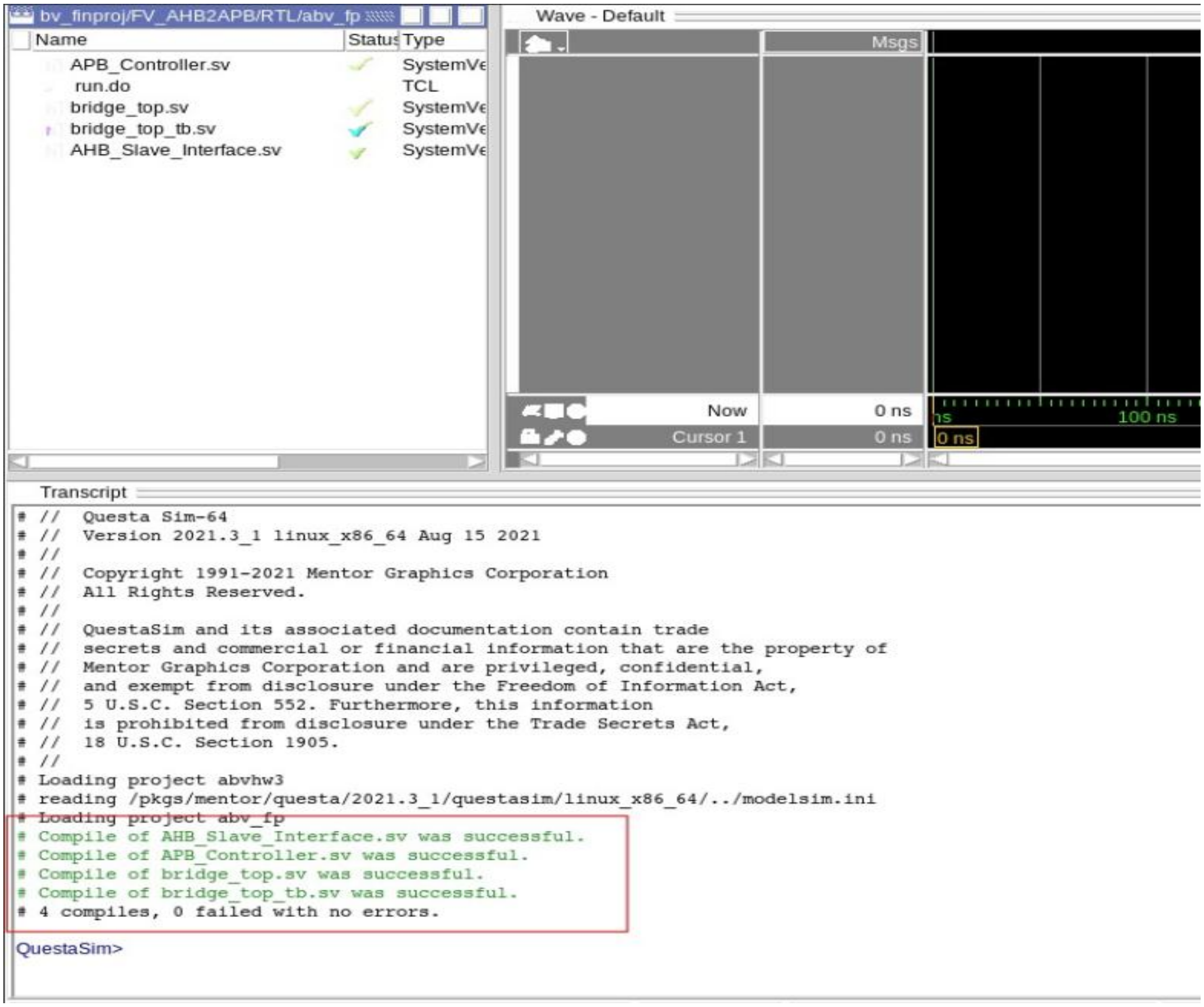
We organized our repository into top-level directories for RTL, run, and SVA. Inside the run directory, we had specific templates that correspond to each RTL file of interest for our AEP, FPV, and FXP TCL files. Inside the SVA directory, we used templates to contain our SystemVerilog assertions, where each assertion is a single RTL file.

Snapshot of Filelist:



```
../RTL/APB_Controller.sv
../RTL/AHB_Slave_Interface.sv
../RTL/bridge_top.sv
../RTL/bridge_top_sva.sva
../RTL/bind_bridge_top_sva.sva
../RTL/AHB_Slave_Interface_sva.sva
../RTL/bind_AHB_Slave_Interface_sva.sva
../RTL/APB_Controller_sva.sva
../RTL/bind_APB_Controller_sva.sva
```

Snapshot of Successful compilation:



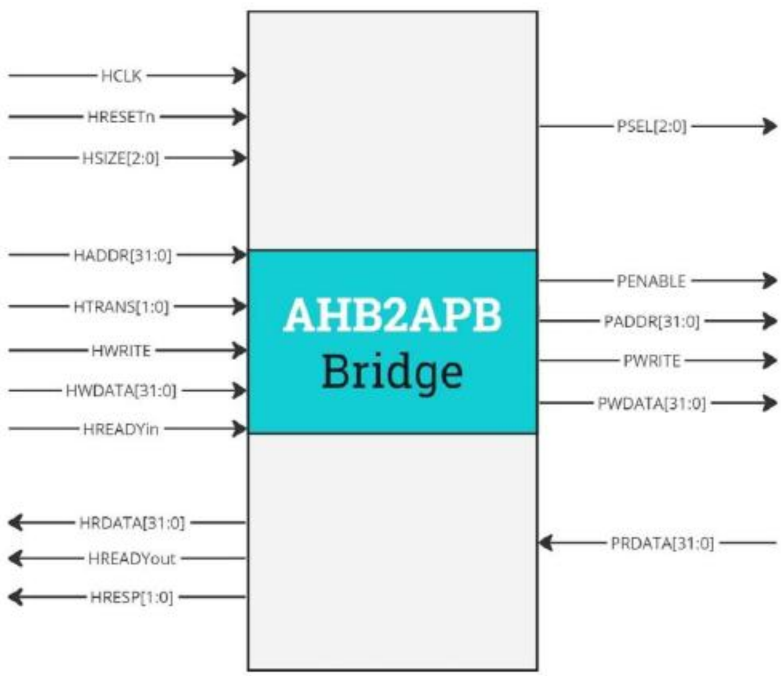
TCL:

one of the tcl file

Specifications:

Design Overview:
• The custom-designed AHB2APB bridge provides the interface between the Advanced High-performance Bus and the Advanced Peripheral Bus, which is designed to meet our needs.
• It allows for AHB masters and APB slaves to be supported in this configuration for smooth transactions between the two interfaces while ensuring that our design specifications are met.

The below diagram illustrates the input and output signals available in the AHB to APB Bridge, most (but not all of them) are exposed as input and output pins in the RTL design we are using. Since this is a bridge protocol, understanding the specifications of those signals is crucial for us to write good properties and to check the right functionality of the bridge.



Design Input Signals (AHB side):
• HCLK: Clock signal for the AHB interface.
• HRESETn: Active-low reset signal for the AHB interface.
• HSIZE[2:0]: Size of the transfer on the AHB interface.
• HADDR[31:0]: Address for the AHB transfer.
• HTRANS[1:0]: Transfer type on the AHB interface.
• HWRITE: Write enable signal for the AHB interface.
• HWDATA[31:0]: Write data on the AHB interface.
• HREADYin: Ready signal indicating the availability of the AHB interface.
Design Output Signals (AHB side):
• HRDATA[31:0]: Read data from the AHB interface.
• HREADYout: Ready signal indicating the readiness of the AHB interface.
• HRESP[1:0]: Response indicating the status of the AHB transfer.

Design Input Signals (APB side):
• PRDATA[31:0]: Read data on the APB interface.
Design Output Signals (APB side):
• PSEL[2:0]: Slave select signal on the APB interface.
• PENABLE: Enable signal for the APB interface.
• PADDR[31:0]: Address for the APB transfer.
• PWRITE: Write enable signal for the APB interface.
• PWDATA[31:0]: Write data on the APB interface.

These input and output signals form the communication interface between the AHB and APB sides of the bridge design. They facilitate the transfer of data and control signals between the two interfaces, ensuring the proper functioning of the AHB2APB bridge.

AHB Master:

We have extracted the following specs from the AMBA reference manual.

HTRANS[1:0] Transfer type	Master	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.

HTRANS[1:0]	Type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer should be ignored by the slave.
01	BUSY	The BUSY transfer type allows bus masters to insert IDLE cycles in the middle of bursts of transfers. This transfer type indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. The transfer should be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
11	SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8 or 16).

HSIZE[2:0] Transfer size	Master	Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.

3.7.1 Transfer direction

When **HWRITE** is HIGH, this signal indicates a write transfer and the master will broadcast data on the write data bus, **HWDATA[31:0]**. When LOW a read transfer will be performed and the slave must generate the data on the read data bus **HRDATA[31:0]**.

3.7.2 Transfer size

HSIZE[2:0] indicates the size of the transfer, as shown in Table 3-3.

Table 3-3 Size encoding

HSIZE[2]	HSIZE[1]	HSIZE[0]	Size	Description
0	0	0	8 bits	Byte
0	0	1	16 bits	Halfword
0	1	0	32 bits	Word
0	1	1	64 bits	-
1	0	0	128 bits	4-word line
1	0	1	256 bits	8-word line
1	1	0	512 bits	-
1	1	1	1024 bits	-

The size is used in conjunction with the **HBURST[2:0]** signals to determine the address boundary for wrapping bursts.

HBURST[2:0] Burst type	Master	Indicates if the transfer forms part of a burst. Four, eight and sixteen beat bursts are supported and the burst may be either incrementing or wrapping.

Table 3-2 Burst signal encoding

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-beat wrapping burst
011	INCR4	4-beat incrementing burst
100	WRAP8	8-beat wrapping burst
101	INCR8	8-beat incrementing burst
110	WRAP16	16-beat wrapping burst
111	INCR16	16-beat incrementing burst

AHB SLAVE INTERFACING:

HRESP[1:0] Transfer response	Slave	The transfer response provides additional information on the status of a transfer. Four different responses are provided, OKAY, ERROR, RETRY and SPLIT.

During a transfer the slave shows the status using the response signals, **HRESP[1:0]**:

OKAY The OKAY response is used to indicate that the transfer is progressing normally and when **HREADY** goes HIGH this shows the transfer has completed successfully.

ERROR The ERROR response indicates that a transfer error has occurred and the transfer has been unsuccessful.

RETRY and SPLIT Both the RETRY and SPLIT transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

In normal operation a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies it is possible for the arbiter to break up a burst and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst.

Table 3-5 Response encoding

HRESP[1]	HRESP[0]	Response	Description
0	0	OKAY	When **HREADY** is HIGH this shows the transfer has completed successfully. The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses.
0	1	ERROR	This response shows an error has occurred. The error condition should be signalled to the bus master so that it is aware the transfer has been unsuccessful. A two-cycle response is required for an error condition.
1	0	RETRY	The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.
1	1	SPLIT	The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete. A two-cycle SPLIT response is required.

TIMING DIAGRAMS:

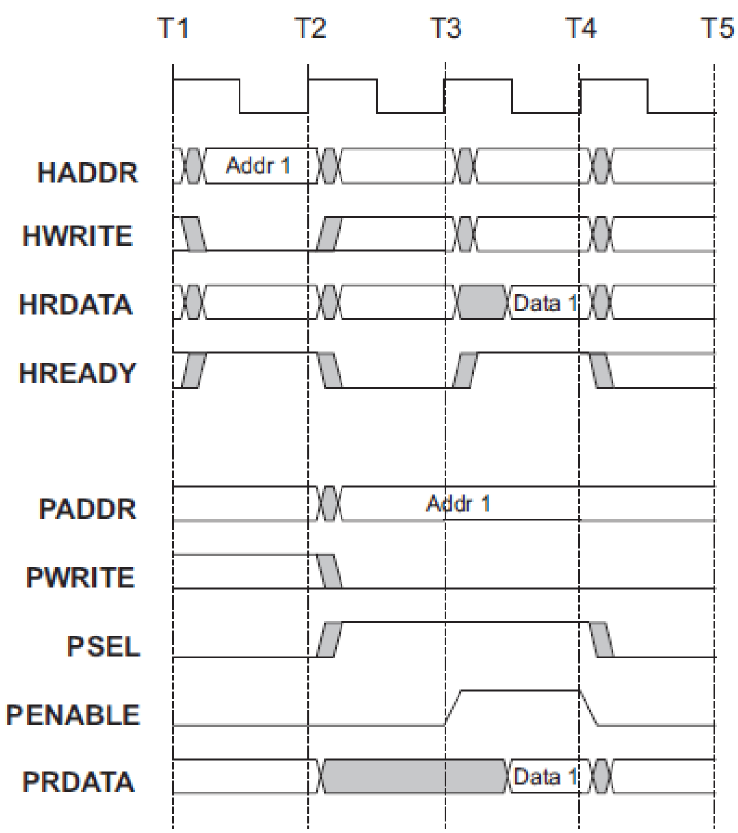


Figure 5-9 Read transfer to AHB

The APB samples the HADDR signal at time T2, as indicated by the PADDR signal being "addr 1" at that time. At T2, the PWRITE signal is driven low while PSEL is asserted high. It is important that PSEL should be high during the whole of the PENABLE signal which must also be asserted high during the transfer in order to make Data1 valid. Also, the HREADY signal should be high when DATA1 is on the HRDATA line. Each read needs a wait state for the transfer to happen.

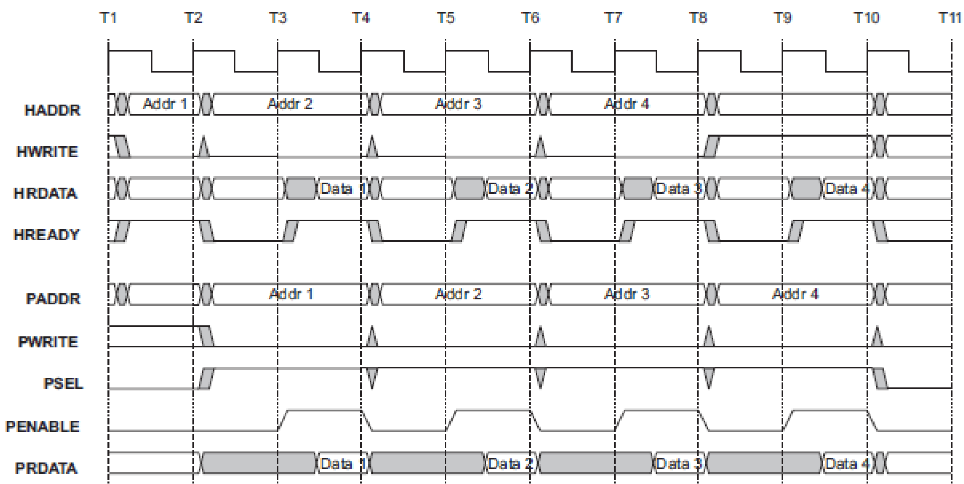


Figure 5-10 Burst of read transfers

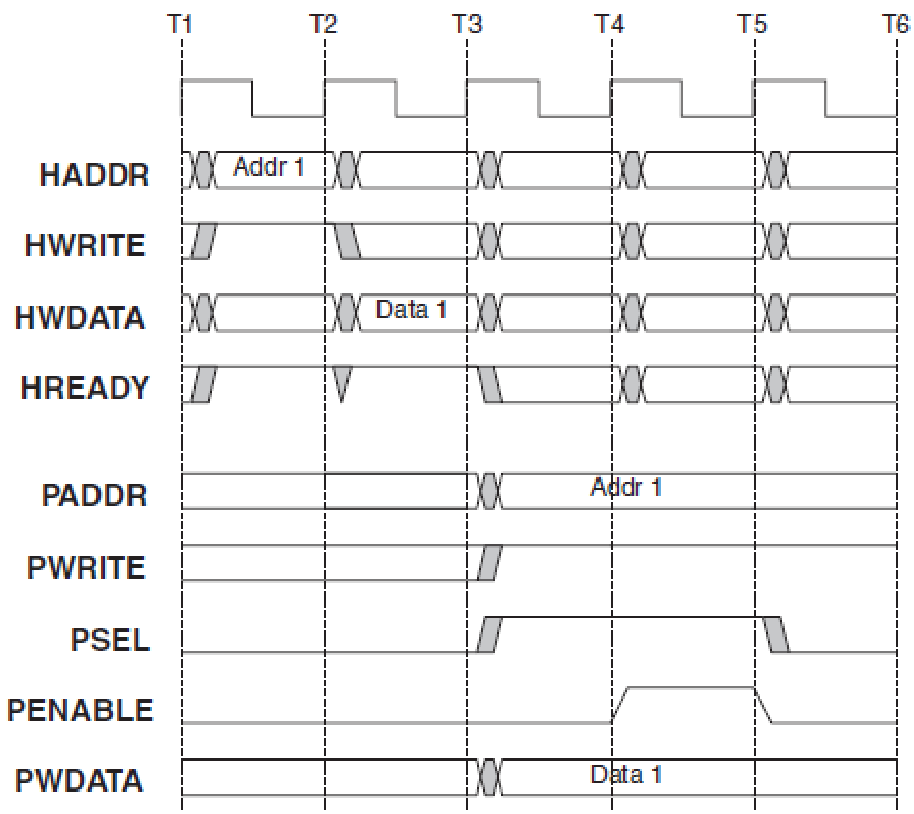


Figure 5-11 Write transfer from AHB

Write transfers can occur with zero transfers. The bridge must sample the address and data of the transfer and must hold these values for the duration of the write transfer.

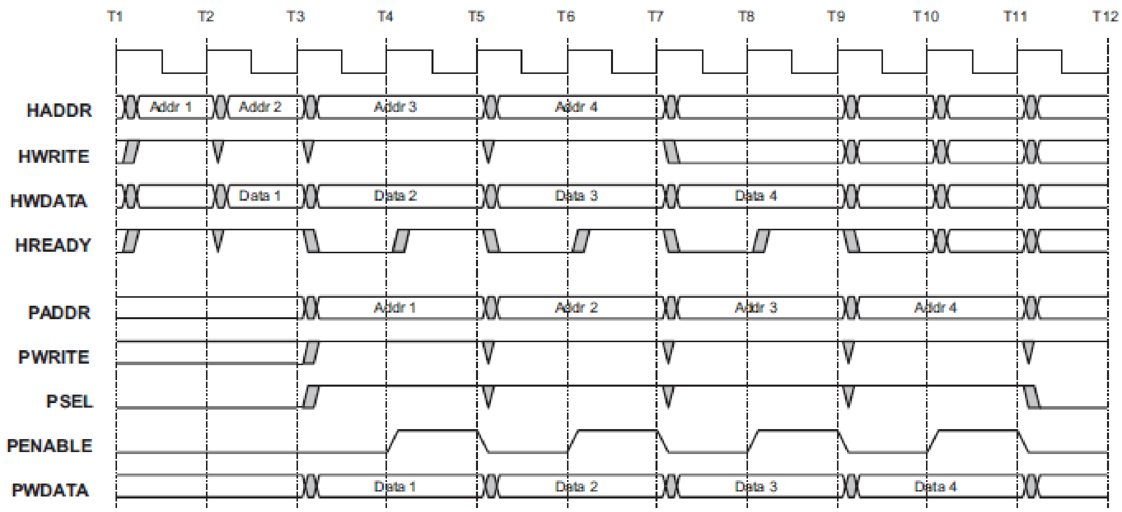


Figure 5-12 Burst of write transfers

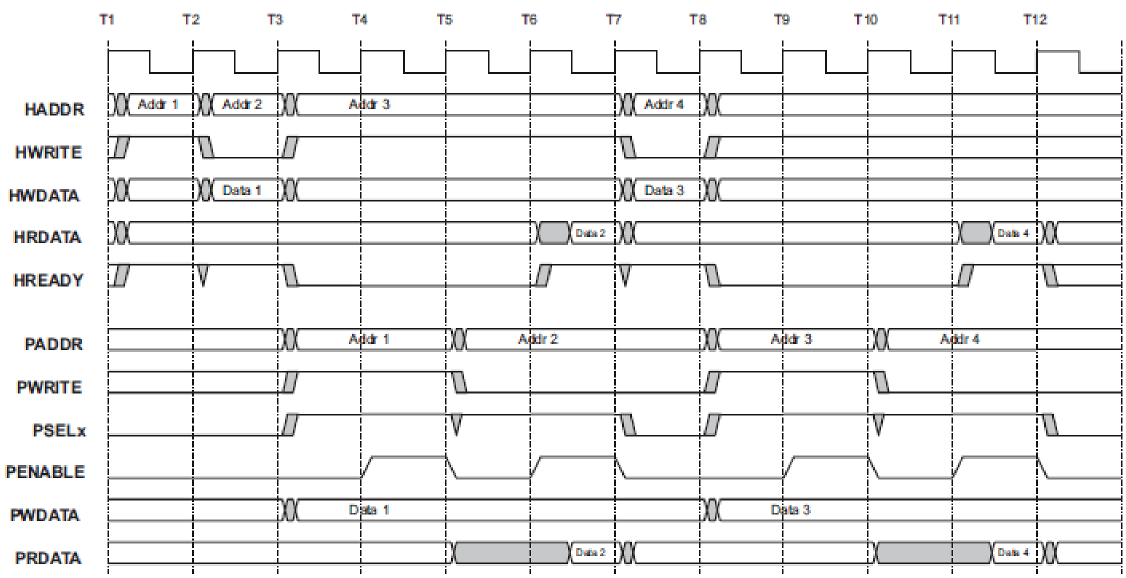


Figure 5-13 Back to back transfers

When a read follows a write, there must be 3 wait states to complete the read. The three wait states can be seen on HREADY and PWRITE.

VERIFICATION PLAN:

The assertions in these files are used for formal verification and dynamic simulation to ensure the correctness of the design under various conditions. Below is an outline of the verification methodology:

1. Functional Properties Verification
Each property in the file corresponds to a functional requirement of the design.

For example:
AHB_slave_interface_sva:

Valid signal generation: Ensures the valid signal is asserted under the correct conditions (e.g., Hreadyin is high, Haddr is within range, and Htrans is valid). Address-based decoding (tempselx): Checks if tempselx is correctly set based on the Haddr range.
Invalid conditions handling: Ensures that the module behaves predictably when Haddr is out of range or invalid.

APB_Controller_sva:
State transitions: Verifies that the controller transitions between states (PRESENT_STATE to NEXT_STATE) correctly based on the design specification.
Read/Write sequencing: Ensures the APB controller follows the correct sequence for read and write transactions (e.g., enabling Penable after valid Pselx).

Bridge_Top_sva:
Protocol compliance: Assumes and checks the behavior of AMBA AHB and APB protocols, such as:
Hwrite and Hreadyin behavior. Valid address ranges for AHB transactions, Synchronization between AHB (Hwdata, Haddr) and APB (Pwdata, Paddr).
Data consistency: Ensures that Hrdata matches Prdata during enable cycles.

2. Temporal Behavior Verification
Assertions include temporal logic to validate timing relationships:

@posedge Hclk: Properties are verified at each clock cycle.
Delay operators (##, |->): Used to verify that signals and state transitions occur in the correct sequence and timing:
Example: write_s ##1 read_s ensures a read occurs one cycle after a write.
Example: PRESENT_STATE == ST_IDLE |-> NEXT_STATE == ST_READ ensures a specific state transition.

3. Protocol Compliance Verification
The files ensure compliance with the AHB and APB protocol requirements:

One-hot encoding: Checks that Pselx is always one-hot or zero-hot.
Penable behavior: Ensures Penable is not high for consecutive cycles.
Address decoding: Verifies address mapping for peripherals in the APB subsystem.
Write and read waits: Ensures that the bridge respects wait states during read-after-write scenarios.

4. Corner Case Verification
The assertions also cover edge cases:

Reset conditions: Ensures all signals are reset to zero after a reset event.

Back-to-back transactions: Verifies behavior during bursts of reads or writes.
Read-after-write timing: Ensures correct timing and data consistency when a read transaction immediately follows a write.

5. Coverage Collection
Coverage properties (e.g., cover_burst_of_reads, cover_burst_of_writes) are used to:

Ensure the verification environment exercises key design scenarios, such as bursts of reads, back-to-back writes, and transitions between states. Identify untested portions of the design.

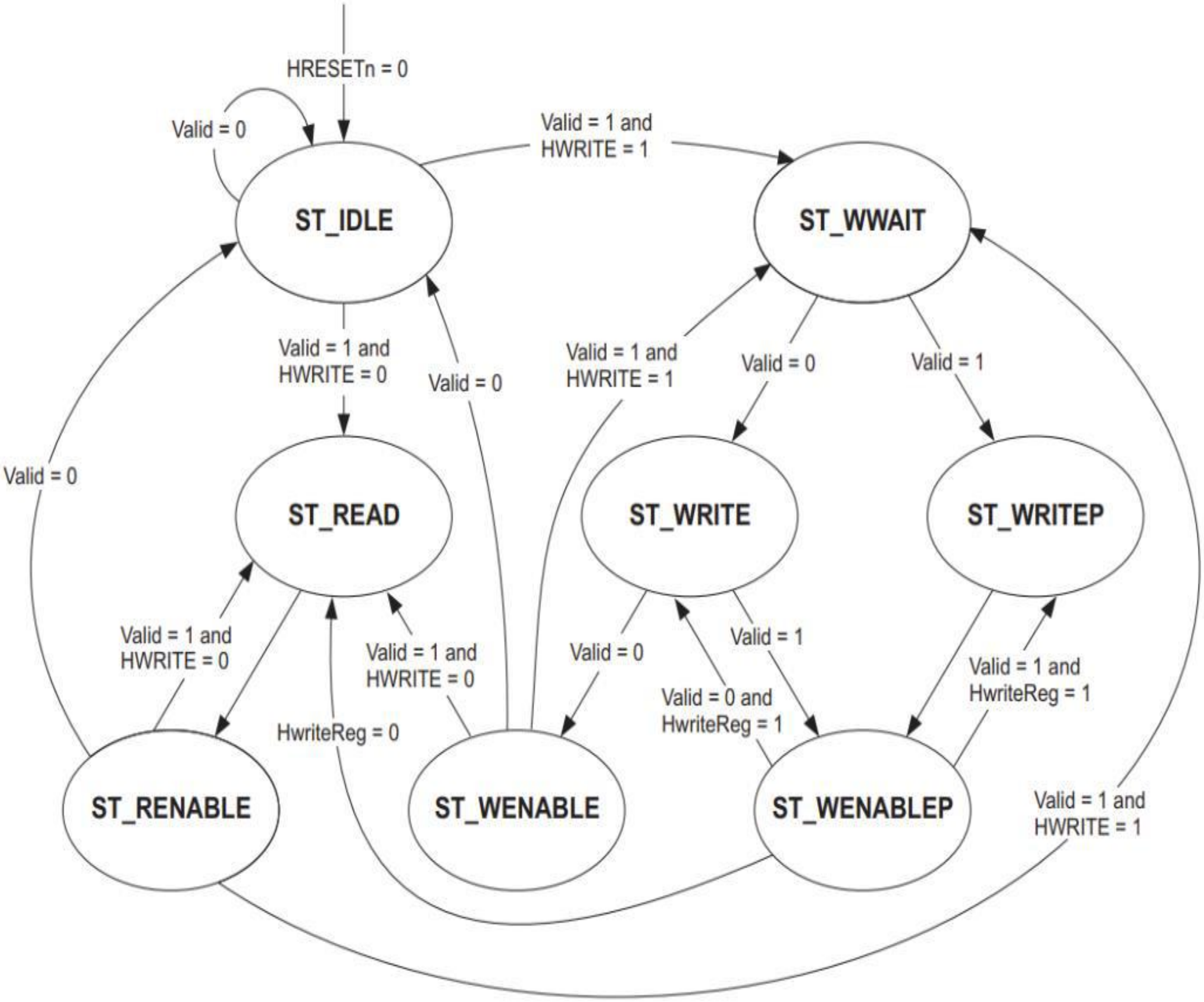
Assertions/Assumptions/Cover:

```
//For Address range 8400_0000 to 8800_0000
sequence psel_s1;
$onehot(Pselx) ##0 (Pselx[1] ##1 Pselx[1]);
endsequence

//For Address range 8800_0000 to 8C00_0000
sequence psel_s2;
$onehot(Pselx) ##0 (Pselx[2] ##1 Pselx[2]);
endsequence

// HRDATA should be same as PRDATA  when PENABLE(Enable cycle)
property same_HR_PR_data;
@(posedge Hclk) disable iff(!Hresetn)
Penable |-> Hrdata == Prdata;
endproperty
assert_same_HR_PR_data: assert property (same_HR_PR_data);

//PWRITE should be same as HWRITE for Read transfer
property same_HPwrite_read;
@(posedge Hclk) disable iff(!Hresetn)
!Hwrite && Hreadyin && !($past(Hwrite) && $past(Hreadyin)) |=> (Pwrite == 0);
endproperty
assert_same_HPwrite_read: assert property (same_HPwrite_read);

////PWRITE should be same as HWRITE for Write transfer
property same_HPwrite_write;
@(posedge Hclk) disable iff(!Hresetn)
write_s |=> ##1 (Pwrite == 1);
endproperty
assert_same_HPwrite_write: assert property (same_HPwrite_write);

//HREADYOUT and PENABLE should be high at end of transaction for Read transfer
property Hreadyout_penable_read;
@(posedge Hclk) disable iff(!Hresetn)
read_s |-> ##2 Penable && Hreadyout;
endproperty
assert_Hreadyout_penable_read: assert property (Hreadyout_penable_read);

//HREADYOUT and PENABLE should be high at end of transaction for write transfer
property Hreadyout_penable_write;
@(posedge Hclk) disable iff(!Hresetn)
write_s |-> ##3 Penable && Hreadyout;
endproperty
assert_Hreadyout_penable_write: assert property (Hreadyout_penable_write);

//PENABLE shouldn't be high for 2 cycles continously
property no_penable_2cycles;
@(posedge Hclk) disable iff(!Hresetn)
Penable |=> !Penable;
endproperty
assert_no_penable_2cycles: assert property (no_penable_2cycles);

// Valid PSEL
property valid_Psel;
@(posedge Hclk) disable iff(!Hresetn)
$onehot0(Pselx);
endproperty
assert_valid_Psel: assert property (valid_Psel);

// For Read transfer next cycle should have corresponding PSEL high for next 2 clocks and should be onehot
```

```
// Transition from ST_IDLE
property p_IDLE_to_WWAIT;
    @(posedge Hclk) disable iff (!Hresetn)
    (PRESENT_STATE == ST_IDLE && valid && Hwrite) |-> (NEXT_STATE == ST_WWAIT);
endproperty
assert_IDLE_to_WWAIT: assert property (p_IDLE_to_WWAIT);

property p_IDLE_to_READ;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_IDLE && valid && !Hwrite |-> NEXT_STATE == ST_READ;
endproperty
assert_IDLE_to_READ: assert property (p_IDLE_to_READ);

property p_IDLE_to_IDLE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_IDLE && !valid |-> NEXT_STATE == ST_IDLE;
endproperty
assert_IDLE_to_IDL: assert property (p_IDLE_to_IDLE);


// Transition from ST_WWAIT
property p_WWAIT_to_WRITE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WWAIT && !valid |-> NEXT_STATE == ST_WRITE;
endproperty
assert_WWAIT_to_WRITE: assert property (p_WWAIT_to_WRITE);

property p_WWAIT_to_WRITEP;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WWAIT && valid |-> NEXT_STATE == ST_WRITEP;
endproperty
assert_WWAIT_to_WRITEP: assert property (p_WWAIT_to_WRITEP);


// Transition from ST_READ
property p_READ_to_RENABLE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_READ |-> NEXT_STATE == ST_RENABLE;
endproperty
assert_READ_to_RENABLE: assert property (p_READ_to_RENABLE);


// Transition from ST_WRITE
property p_WRITE_to_WENABLE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WRITE && !valid |-> NEXT_STATE == ST_WENABLE;
endproperty
assert_WRITE_to_WENABLE: assert property (p_WRITE_to_WENABLE);

property p_WRITE_to_WENABLEP;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WRITE && valid |-> NEXT_STATE == ST_WENABLEP;
endproperty
assert_WRITE_to_WENABLEP: assert property (p_WRITE_to_WENABLEP);


// Transition from ST_WRITEP
property p_WRITEP_to_WENABLEP;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WRITEP |-> NEXT_STATE == ST_WENABLEP;
endproperty
assert_WRITEP_to_WENABLEP: assert property (p_WRITEP_to_WENABLEP);


// Transitions from ST_RENABLE
property p_RENABLE_to_IDLE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_RENABLE && !valid |-> NEXT_STATE == ST_IDLE;
endproperty
assert_RENABLE_to_IDLE: assert property (p_RENABLE_to_IDLE);
```

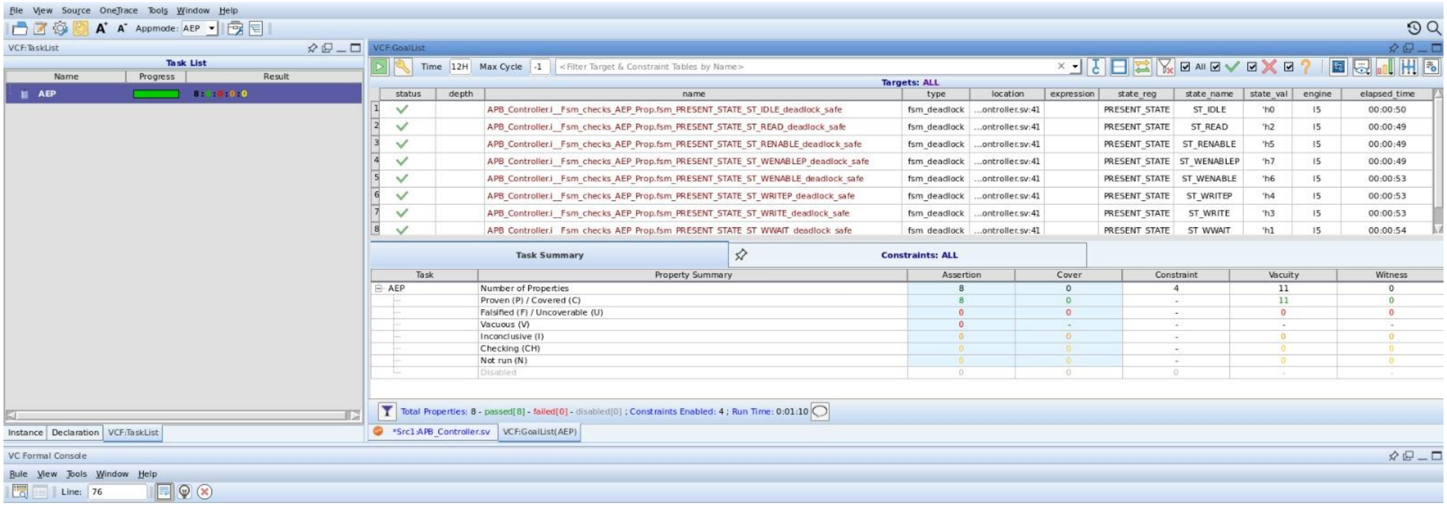
```
// Assert valid signal generation logic
assert property (@(posedge Hclk)
    (Hresetn && Hreadyin && (Haddr >= 32'h8000_0000 && Haddr < 32'h8C00_0000) && (Htrans == 2'b10 || Htrans == 2'b11))|-> (valid == 1'b1));

// Assert that valid should not be high when conditions are not met
assert property (@(posedge Hclk)
    (!(Hresetn && Hreadyin && (Haddr >= 32'h8000_0000 && Haddr < 32'h8C00_0000) && (Htrans == 2'b10 || Htrans == 2'b11)))|-> (valid == 1'b0));


// Assert tempselx logic for different ranges of Haddr
assert property (@(posedge Hclk)
    (Hresetn && (Haddr >= 32'h8000_0000 && Haddr < 32'h8400_0000)) |-> (tempselx == 3'b001));

assert property (@(posedge Hclk)
    (Hresetn && (Haddr >= 32'h8400_0000 && Haddr < 32'h8800_0000)) |-> (tempselx == 3'b010));

assert property (@(posedge Hclk)
    (Hresetn && (Haddr >= 32'h8800_0000 && Haddr < 32'h8C00_0000)) |-> (tempselx == 3'b100));



// Assert tempselx should be 000 for invalid ranges of Haddr
assert property (@(posedge Hclk)
    (Hresetn && !(Haddr >= 32'h8000_0000 && Haddr < 32'h8C00_0000)) |-> (tempselx == 3'b000));
```

FSM FOR AHB TO APB INTERFACE:

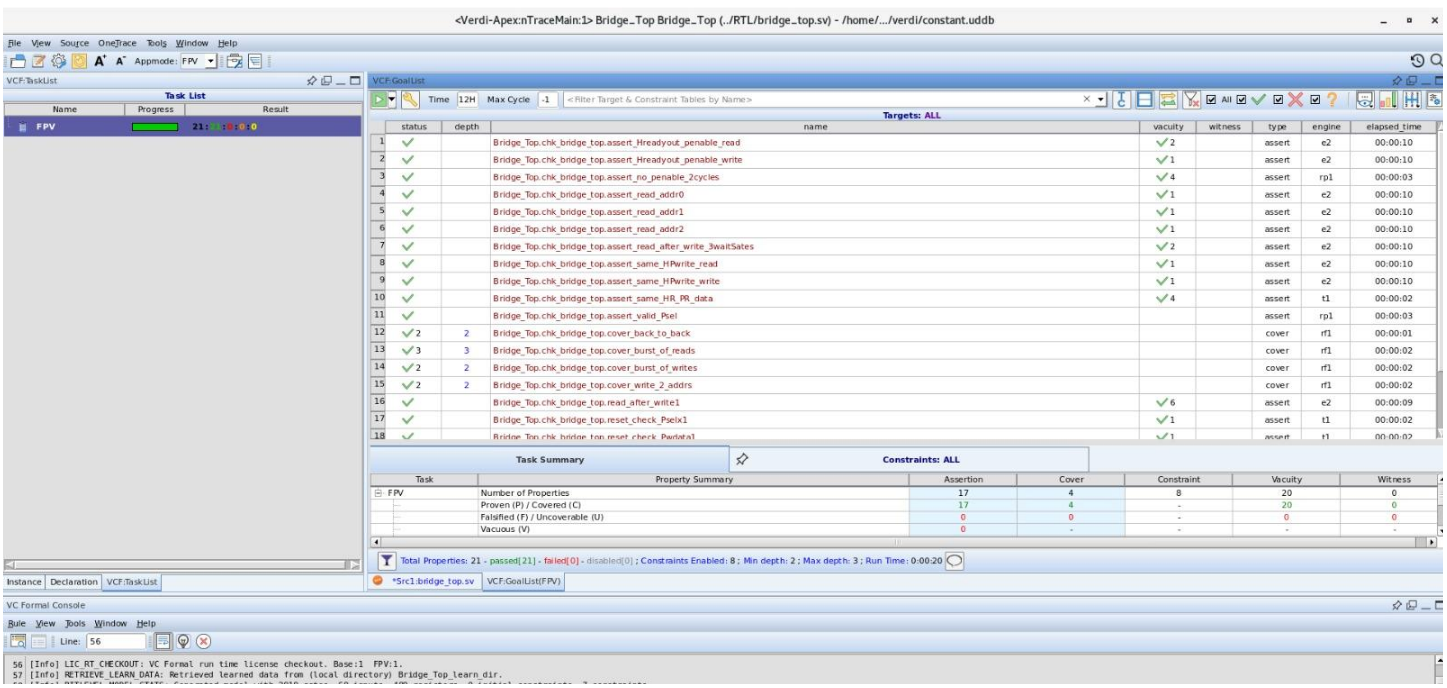


First, we started exploring the design by running the completely automated apps, AEP and FXP, we ran them on the top module with the filelist, as well as on the individual modules for better signal visibility. In both cases, we didn't get much information using those automated techniques.

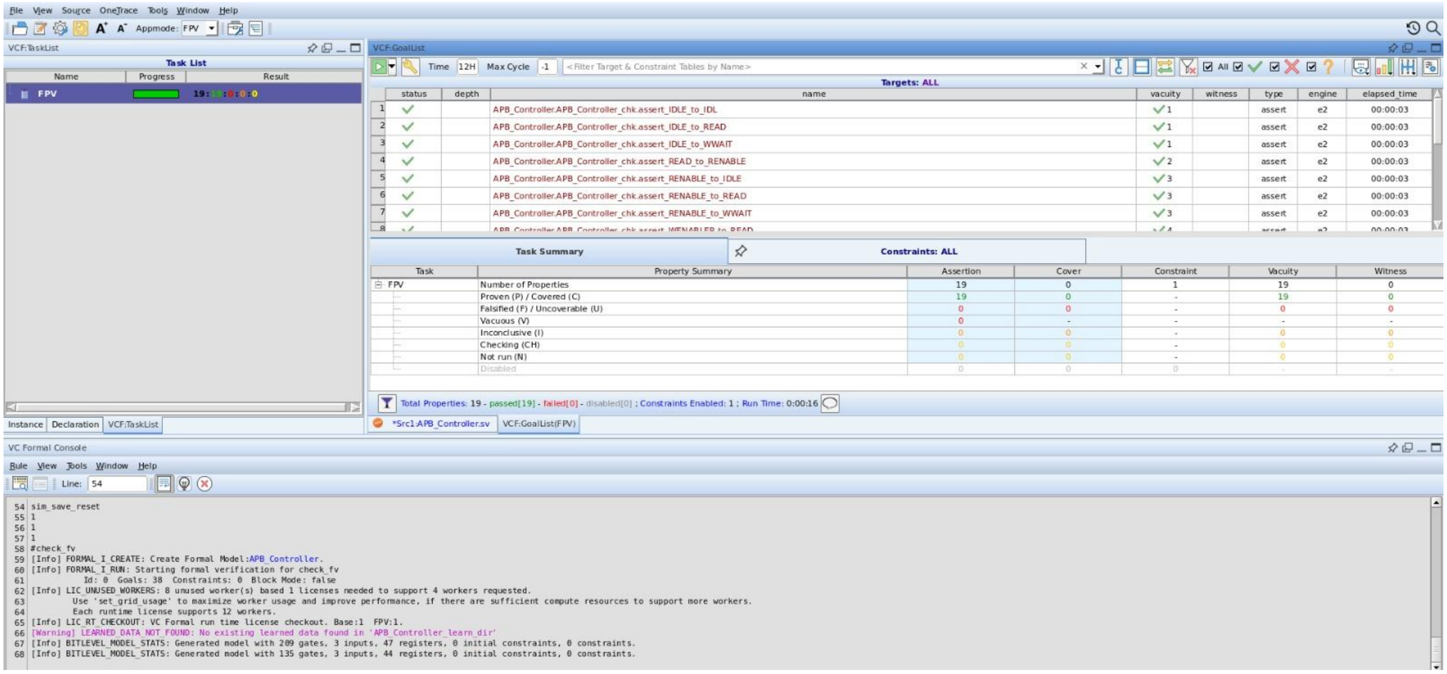
AEP:



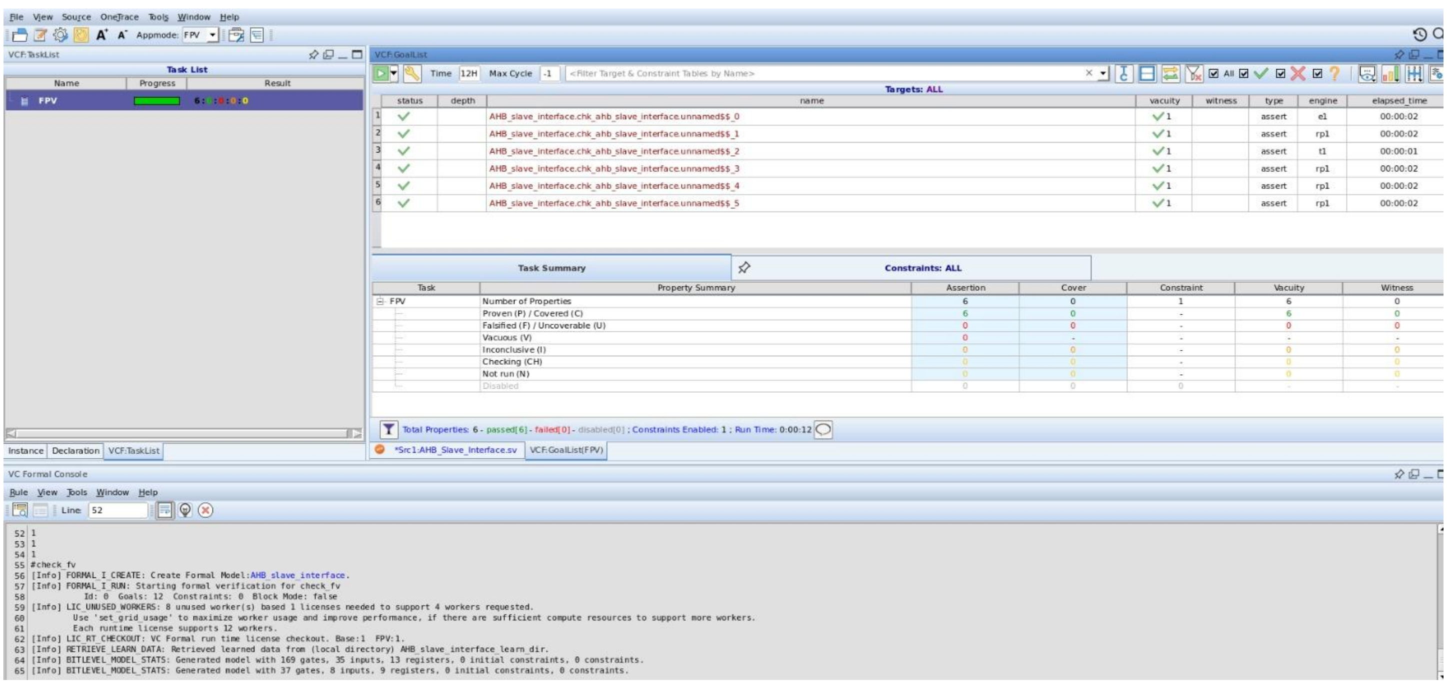
FPV_Bridge:



FPV_APB_CONTROLLER:



FPV_AHB:



Challenges:

Specifically, we faced several major obstacles at the start of the above effort. The foremost of these issues involved obtaining a proper RTL (Register Transfer Level) code that met all our specified criteria. In other words, we were looking for RTL facilitated the burst read and write transactions as specified for the ARM AMBA standards. In the light of these challenges, we decided to proceed with the RTL code available for the AHB2APB bridge design, as an integral part of this process.

Summary:

This paper outlines our verification efforts on the AHB-APB bridge, which is a critical link between AHB masters and slave interfaces. Our focus was on read and write transfers, the AHB to APB state machine, handling Read After Write issues, and reset signal handling. We analyzed advanced verification methodologies and shared our experience on the PSELX signal. We used the AMBA specifications to create meaningful assumptions, assertions, and cover properties to cover the given RTL design as much as the RTL design and time allowed us.

References:

1) Github
2) Lecture recordings
3) ARM SPEC DOCUMENT
4) Udemy