Maseeh College of Engineering
and Computer Science
PORTLAND STATE UNIVERSITY

ECE 560: FALL 2023 - PROJECT
ASSERTION BASED VERIFICATION

ASSERTION-BASED VERIFICATION OF AN AHB2APB BRIDGE

FINAL REPORT

MOHAMED GHONIM
ALEXANDER MASO
PHANINDRA VEMIREDDY
HAYDEN GALANTE

12/06/2023

Table of Contents

Introduction

The objective of this project and the verification plan is to outline the strategies, methodologies, and findings of our efforts in verifying the efficiency and performance capabilities of the AHB-APB bridge design. This bridge serves as the critical intermediary facilitating seamless interaction between the high-performance Advanced High-performance Bus (AHB) and the power-efficient Advanced Peripheral Bus (APB).

As semiconductor technology continues to advance, the need for robust and accurate verification processes becomes increasingly imperative. The intricate dance between high-speed data transmission, memory mapping, and peripheral control mandates a comprehensive approach to bridge verification. This document encapsulates our comprehensive plan for assessing the AHB-APB bridge's functionality, ensuring it meets the requirements set forth by the ARM AMBA spec sheet.

Our journey through this verification process has been both challenging and enlightening. We tried to write specific assertions, assumptions and cover properties to verify the design.

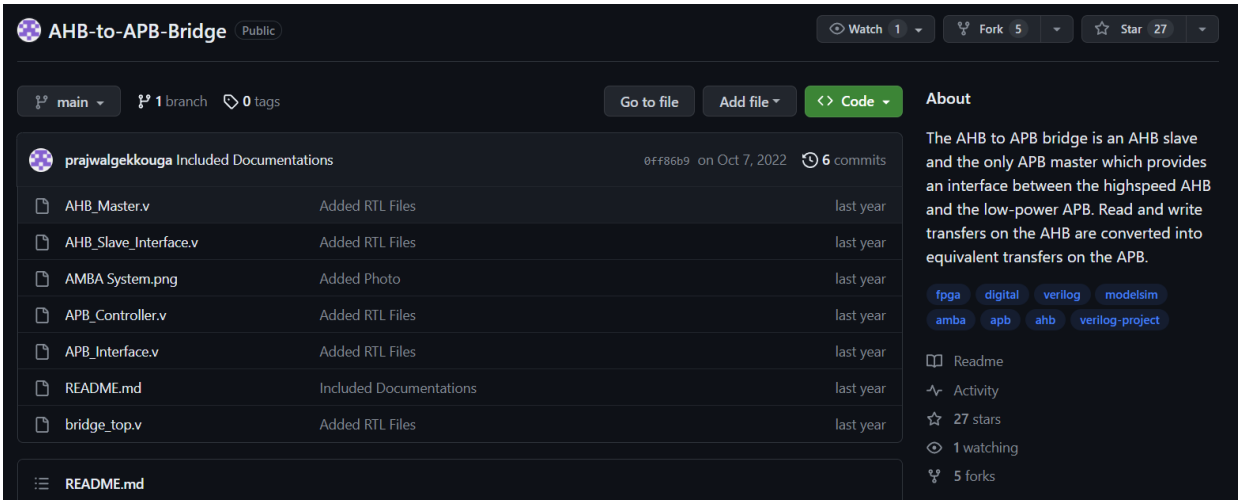
Project Proposal/Overview

Our final project, encompassing the principles of assertion-based verification, aimed to create and execute a robust validation plan for the functionality of the AHB-to-APB (AHB2APB) bridge design. The initiative was driven by our commitment to adhere to the official ARM AMBA standards, which govern the specifications of this critical interface.

During this project, our team embarked on the task of formulating a set of properties derived directly from the bridge's design specifications. We concurrently developed an array of assertions, assumptions, and cover properties designed to comprehensively evaluate the bridge's various capabilities. However, it is noteworthy that we encountered noteworthy challenges during the initial stages of this endeavor.

One significant challenge revolved around the availability of RTL (Register Transfer Level) code that fully adhered to the specified criteria. Specifically, we sought RTL that supported the diverse burst read and write transactions as outlined in the ARM AMBA standards. Despite these challenges, we made the strategic decision to proceed with the RTL code available for the AHB2APB bridge design, a crucial component of this validation process. This RTL code is included as an attachment in this report for reference.

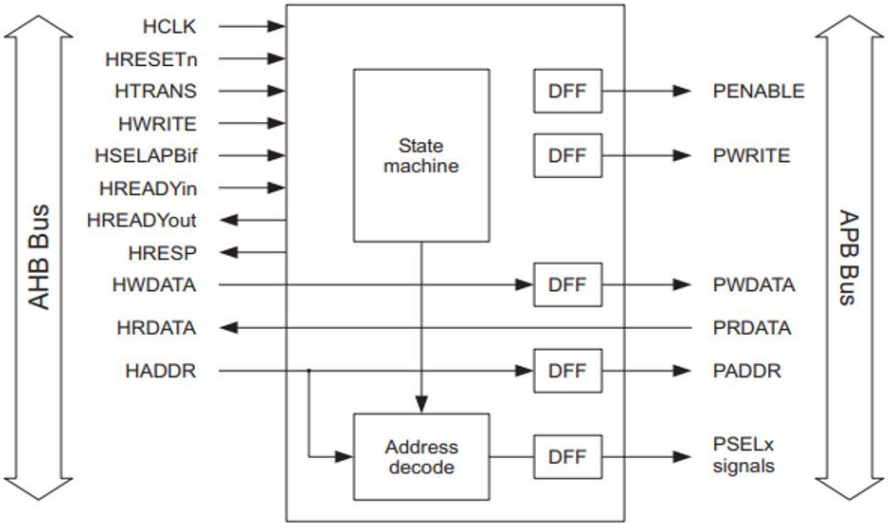
The chosen RTL code can also be found here: https://github.com/prajwalgekkouga/AHB-to-APB-Bridge.



Bus Bridge Description

The AHB2APB bridge, developed by ARM, serves as a crucial interface linking the high-speed Advanced High-performance Bus (AHB) with the low-power Advanced Peripheral Bus (APB). This bridge is designed to help facilitate various read and write operations between the two buses. Key components of this bridge include an AHB slave bus interface, an APB transfer state machine that operates independently of the device's memory map, and mechanisms for APB output signal generation.

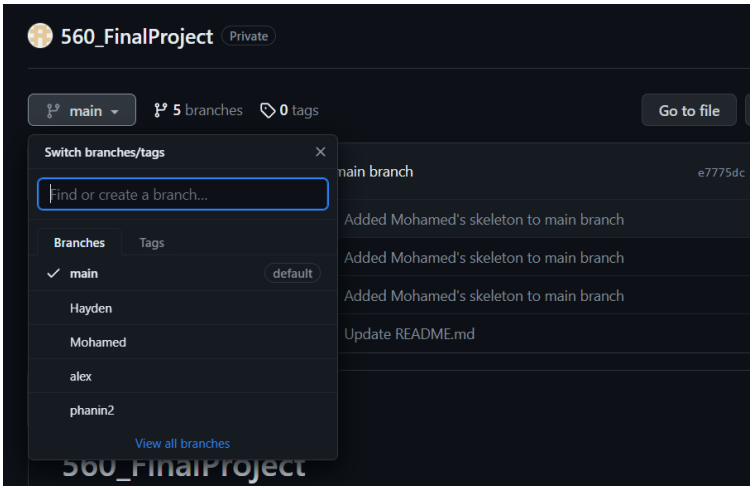
The AHB2APB bridge primarily functions to store addresses, control signals, and data from the AHB, subsequently directing these to the APB peripherals and relaying data along with a response signal back to the AHB. The bridge manages the APB data bus through two separate channels: the read data path (PRDATA) the write data path (PWDATA). Additionally, the bridge is capable of managing both sequential and non-sequential data transfers of different sizes (HSIZE). This flexibility allows for a versatile communication interface between the high-speed and low-speed buses.

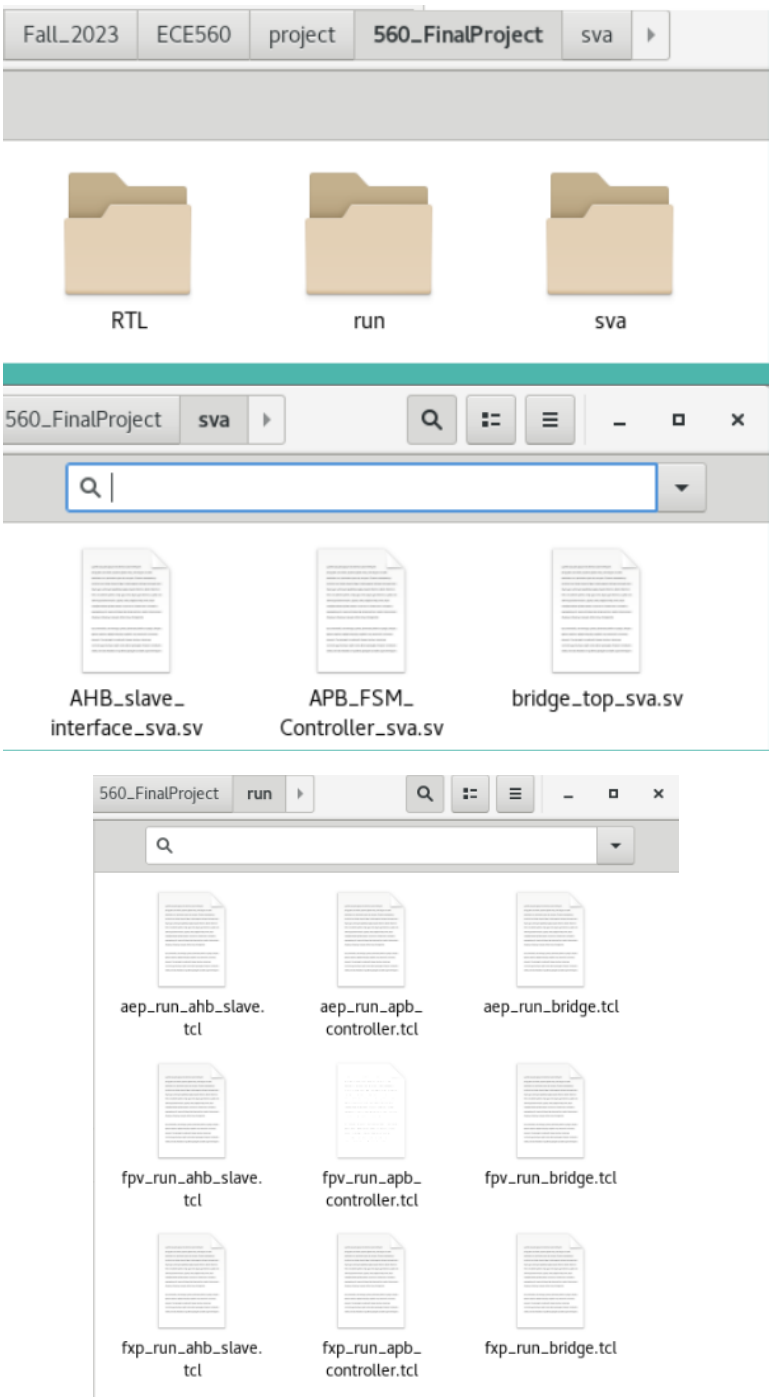


Project Collaboration and file structure

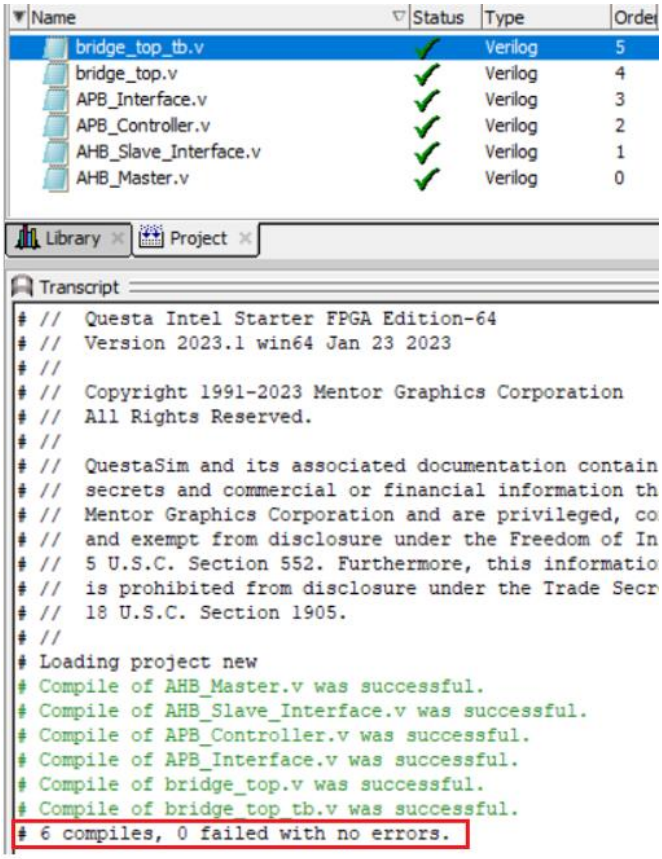
Our team used a collaborative approach and the following file structure we adopted for effective project management. We decided to use a GitHub repository to manage our RTL, as well as all our properties, assertions, assumptions, and cover points. In order to allow every team member to independently develop their properties while accessing the work of others, we created separate branches for each member of the team and populated them with template files.

We structured our repository into distinct folders for RTL, run, and SVA. Within the run directory, we created specific templates corresponding to each RTL file for our automated equivalence property (AEP), formal property verification (FPV), and formal equivalence point (FXP) TCL files. In the SVA directory, we used templates to house our SystemVerilog assertions, one for each RTL file.

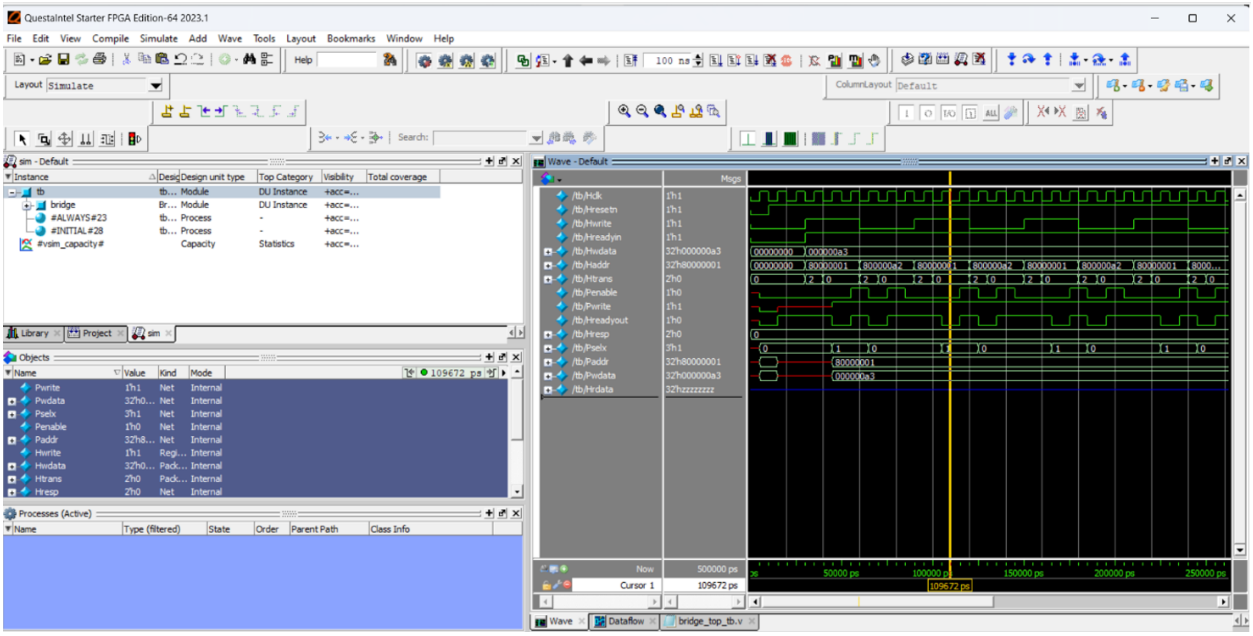


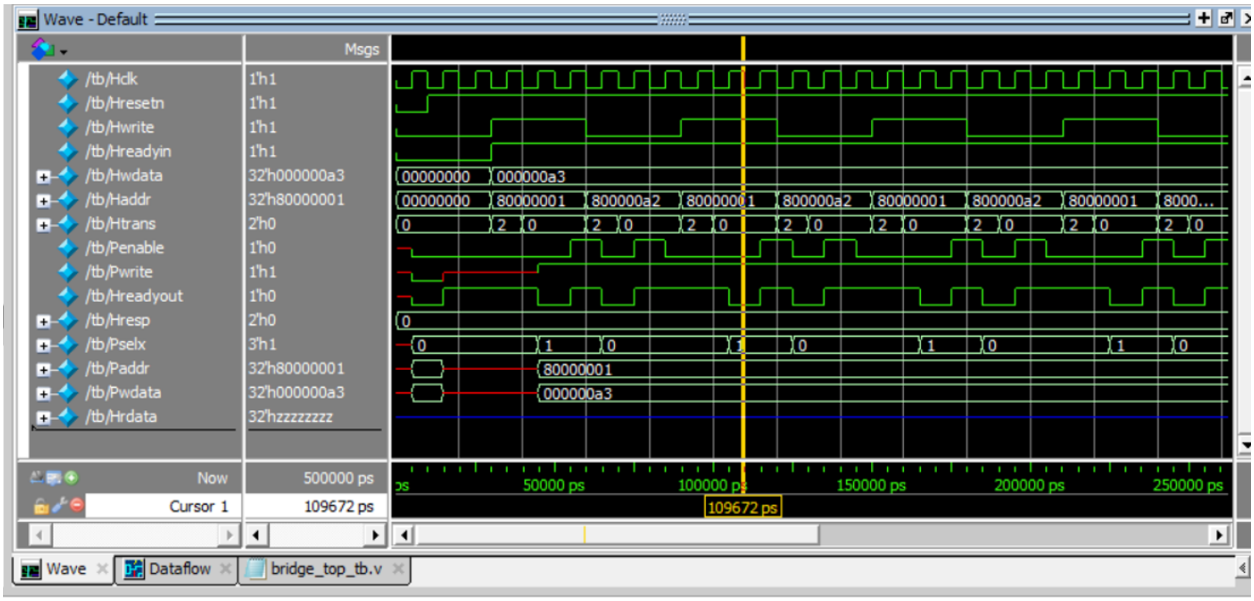


Before we got started, we made sure that the RTL design we are using compiles without any errors or warnings.



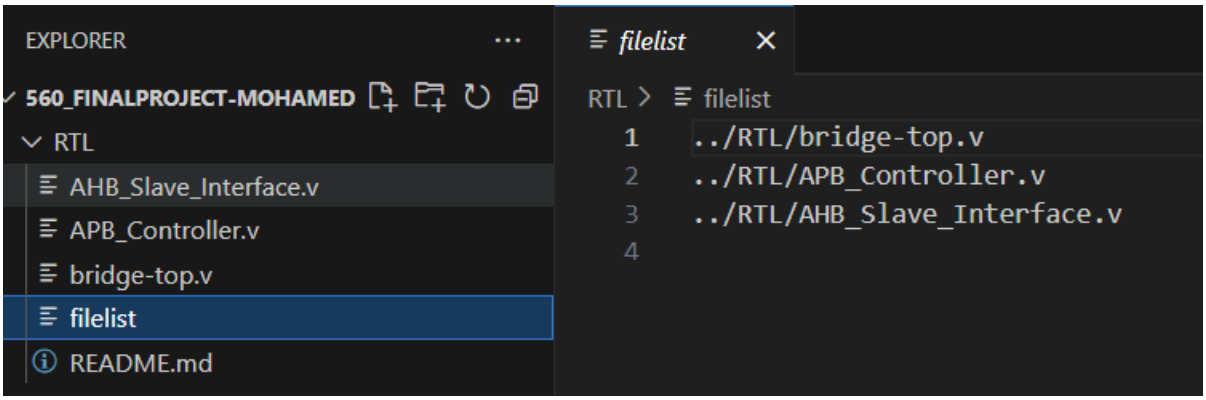
Since we used Questasim to compile the code, we also decided to make a very basic Testbench just to wiggle the signals and to possibly get a better understanding of the internal signal propagation in this specific design.





We then moved forward with our formal verification analysis.

Our filelist:



One of our TCL scripts:

fpv_run_bridge.tcl

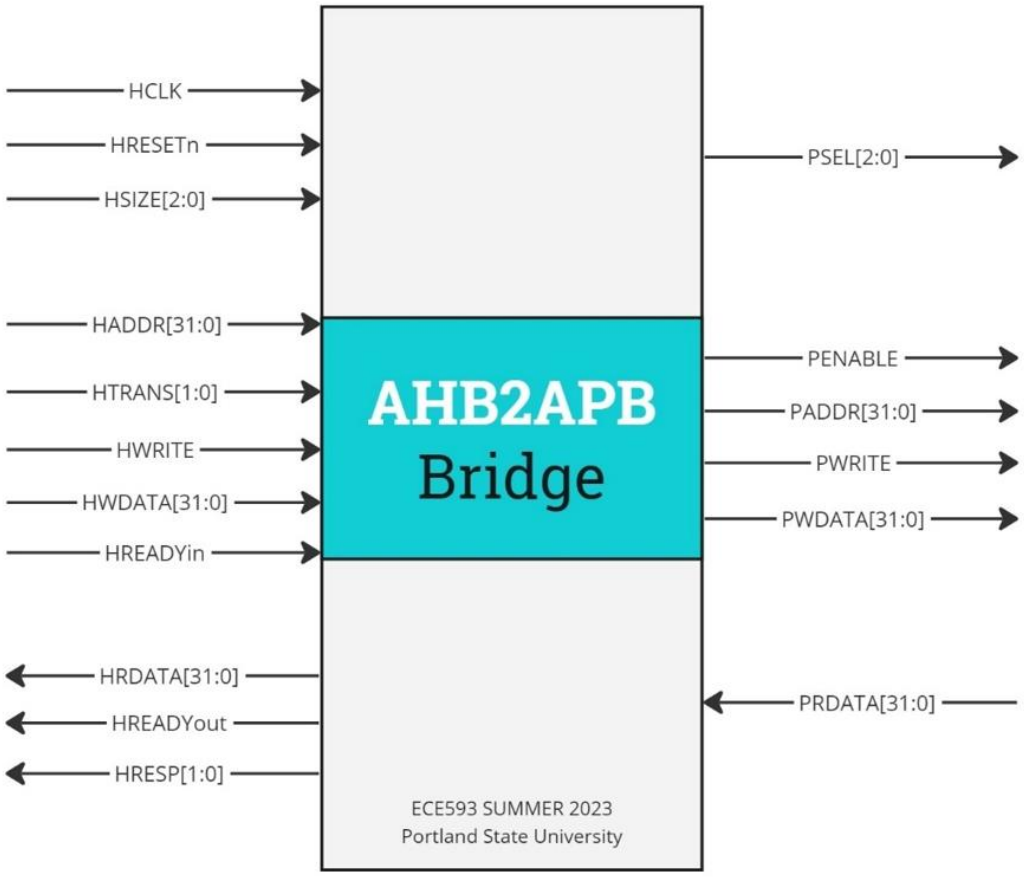
run > fpv_run_bridge.tcl

```tcl

# Select AEP as the VC Formal App mode
set_fml_appmode FPV
set design Bridge_Top

set_fml_var fml_aep_unique_name true
read_file -top $design -format sverilog -sva \
-aep all -vcs {-f ../RTL/filelist +define+INLINE_SVA \
   ../sva/bridge_top_sva.sv}
#read_waiver_file -elfiles aep.el

# Creating clock and reset signals
create_clock Hclk -period 100
create_reset Hresetn -sense low

# Runing a reset simulation
sim_run -stable
sim_save_reset

```

Specifications

Design Overview:

- Our customized AHB2APB bridge design provides a bridge between the Advanced High-performance Bus (AHB) and the Advanced Peripheral Bus (APB) interfaces, tailored to meet our specific requirements.
- The design supports AHB masters and APB slaves, allowing seamless transactions between the two interfaces while ensuring compliance with our design specifications.



The diagram above shows the input and output signals available in the AHB to APB Bridge, most (by not all of them) are visible 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 effective properties and to verify the correct 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

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

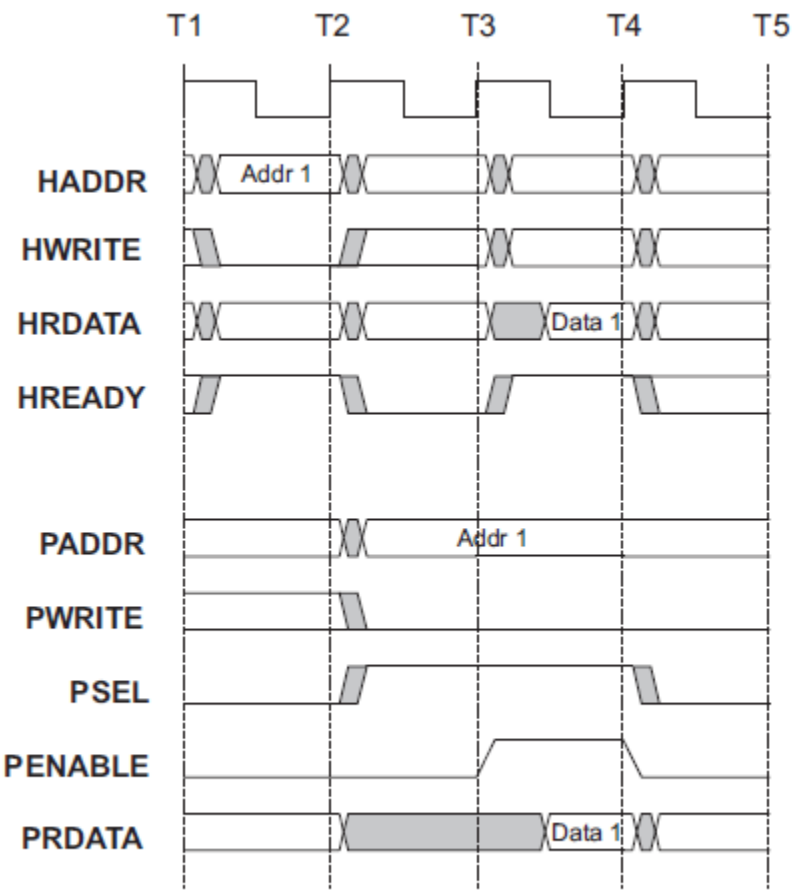


Figure 5-9 Read transfer to AHB

HADDR is sampled by APB at T2 which can be seen with the PADDR signal displaying “addr 1” at T2. At T2 PWRITE is pulled low and PSEL is asserted high. PSEL must also be high during the PENABLE signal, which must also be asserted high during the transfer for Data1 to be valid. HREADY Must also be high when DATA1 is on the HRDATA line. Each read requires a wait state for the transfer to occur.

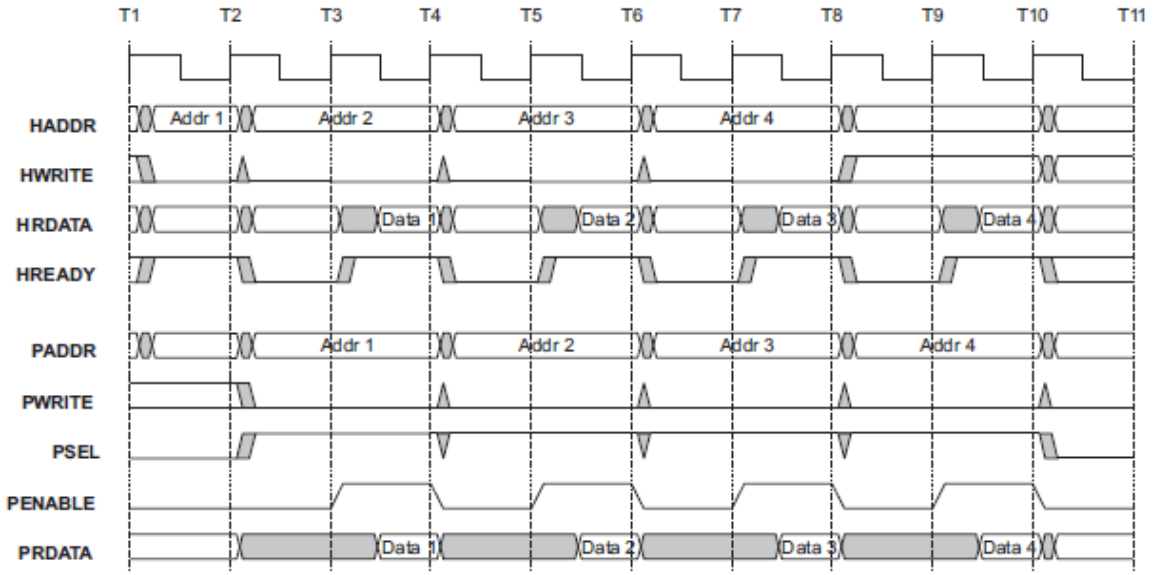


Figure 5-10 Burst of read transfers

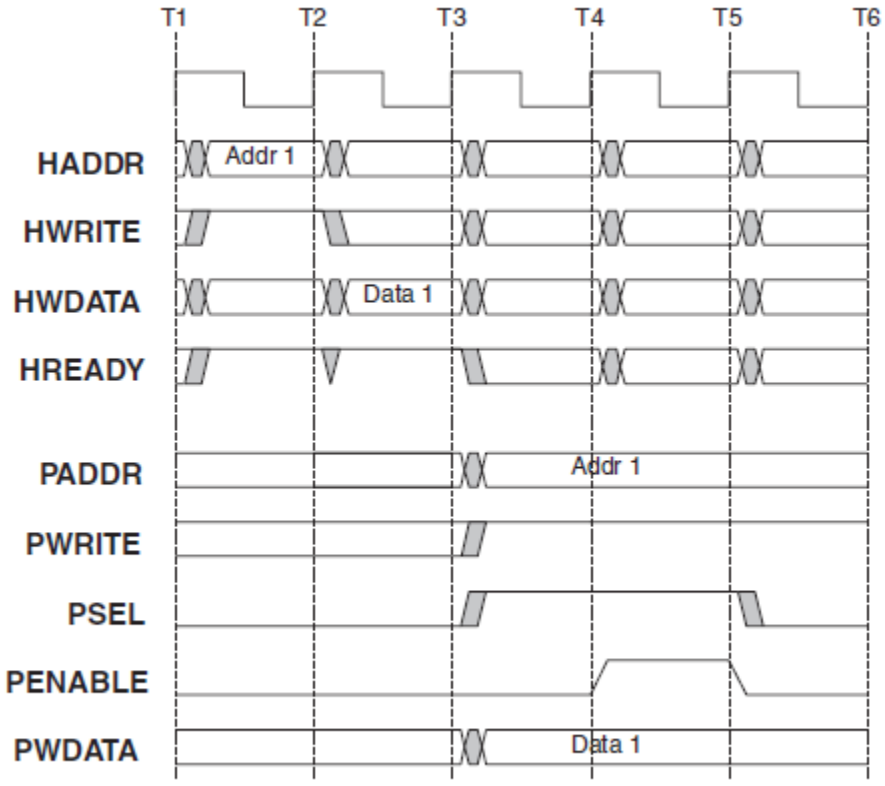


Figure 5-11 Write transfer from AHB

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.

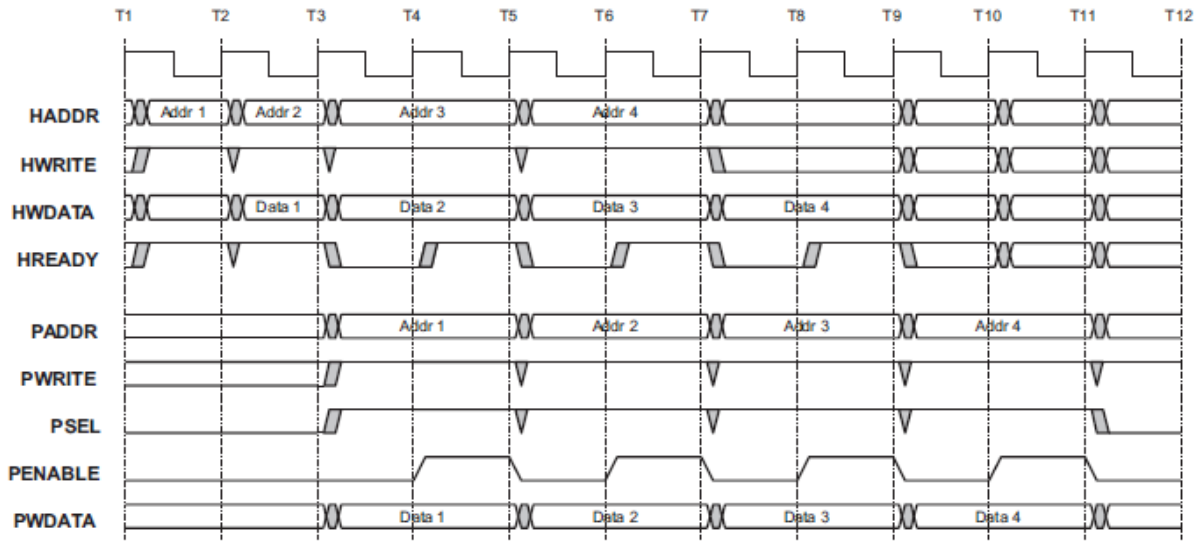


Figure 5-12 Burst of write transfers

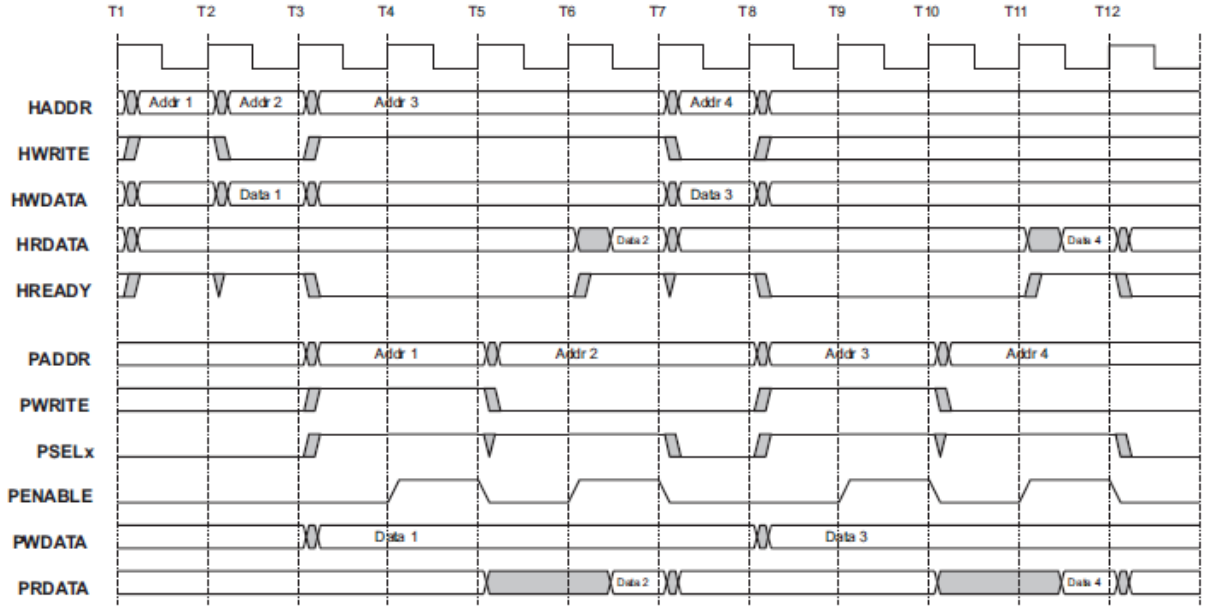


Figure 5-13 Back to back transfers

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 Execution

Function to be validated	Methodology Used	Further Description	Input signals	Output Signals
APB_Controller	AEP, FXP	Explore the high-level implementation of the bridge and controller		
Bridge Controller FSM	FPV	Verify the FSM states and transitions given the AMBA specifications		
Output logic based on the states	FPV	Verify that each state produces the expected output according to specs		
System functions (reset, and error handling)	FPV	Verify the behavior of the different parts of the design under the reset and error conditions.	HRESP = Error, HRESET	Observe affeced signals
Nonsequential (Read and Write) transactions	FPV	For different combinations of HSIZE (byte, halfword, and word) and HBURST (Single transfer, Incremental, 4-beat wrapping, 8 beat wrapping), different HRESP (Okay, Error) and different HTRANS (Nonsequential, Sequential)	HADDR, HWDATA, HRESET	PRDATA, HREADY, HRESP,
Sequential (Read and Write) transactions	FPV	For different combinations of HSIZE (byte, halfword, and word) and HBURST (Single transfer, Incremental, 4-beat wrapping, 8 beat wrapping), different HRESP (Okay, Error) and different HTRANS (Nonsequential, Sequential)	HADDR, HWDATA, HRESET	PRDATA, HREADY, HRESP,
Incremental Burst (Read and Write) transactions	FPV	For different combinations of HSIZE (byte, halfword, and word) and HBURST (Single transfer, Incremental, 4-beat wrapping, 8 beat wrapping), different HRESP (Okay, Error) and different HTRANS (Nonsequential, Sequential)	HADDR, HWDATA, HRESET	PRDATA, HREADY, HRESP,
Wrapping Burst (Read and Write) transactions	FPV	For different combinations of HSIZE (byte, halfword, and word) and HBURST (Signle transfer, Incremental, 4-beat wrapping, 8 beat wrapping), different HRESP (Okay, Error) and different HTRANS (Nonsequential, Sequential)	HADDR, HWDATA, HRESET	PRDATA, HREADY, HRESP,

Interface to be Validated	Methodology	
AHB Slave Interface	FPV	Verify the (Haddr1, Haddr2, Hwdata1, Hwdata2) signals, the Valid logic generation, and the Tempslex Logic

APB_Controller	
Use AEP and FXP to explore the design and find any obvious bugs	Automated VC Formal Check

Bridge Controller FSM
IDLE to IDLE
IDLE to READ
IDLE to WRITE-WAIT
READ to READ-ENABLE
READ-ENABLE to READ
READ-ENABLE to IDLE
READ-ENABLE to WRITE-WAIT
WRITE-WAIT to WRITE-P
WRITE-WAIT to WRITE
WRITE-P to WRITE-ENABLE-P
WRITE-ENABLE-P to WRITE-P
WRITE-ENABLE-P to WRITE
WRITE to WRITE-ENABLE-P
WRITE to WRITE-ENABLE
WRITE-ENABLE to WRITE-WAIT
WRITE-ENABLE to IDLE
WRITE-ENABLE to READ

Output logic based on the states
IDLE state with (valid &&~Hwrite)
IDLE state with (valid &&Hwrite)
WAIT state wih (not valid)
WAIT state wih (valid)
READ state
WRITE state (with ~valid)
WRITE state (with valid)
WRITEP (ABP) state
RENABLE state with (valid &&~Hwrite)
RENABLE state with (valid &&Hwrite)
RENABLE state with (not valid)
WENABLE state with (~valid && Hwritereg)
WENABLE state with (valid)
WENABLEP state with (~valid && Hwritereg)
WENABLEP state with (valid)

System functions	
If Hresetn is 0	reset everything to 0
HRESP = Error	

Read Transactions
Covered in the sequential/nonsequential transactions
HRDATA should be same as PRDATA when PENABLE(Enable cycle)
HREADYOUT and PENABLE should be high at end of transaction
PWRITE should be same as HWRITE

If HWRITE is low and HREADYIN is high next cycle should have PSEL high for next 2 clocks and should be onehot and PADDR should be same as HAADR If HWRITE is low and HREADYIN is high then PENABLE should be high after 2 clocks
If a read transfer immediately follows a write, then 3 wait states are required to complete the read PSEL should be a Onehot PENABLE shouldn't be high for 2 cycles continuously

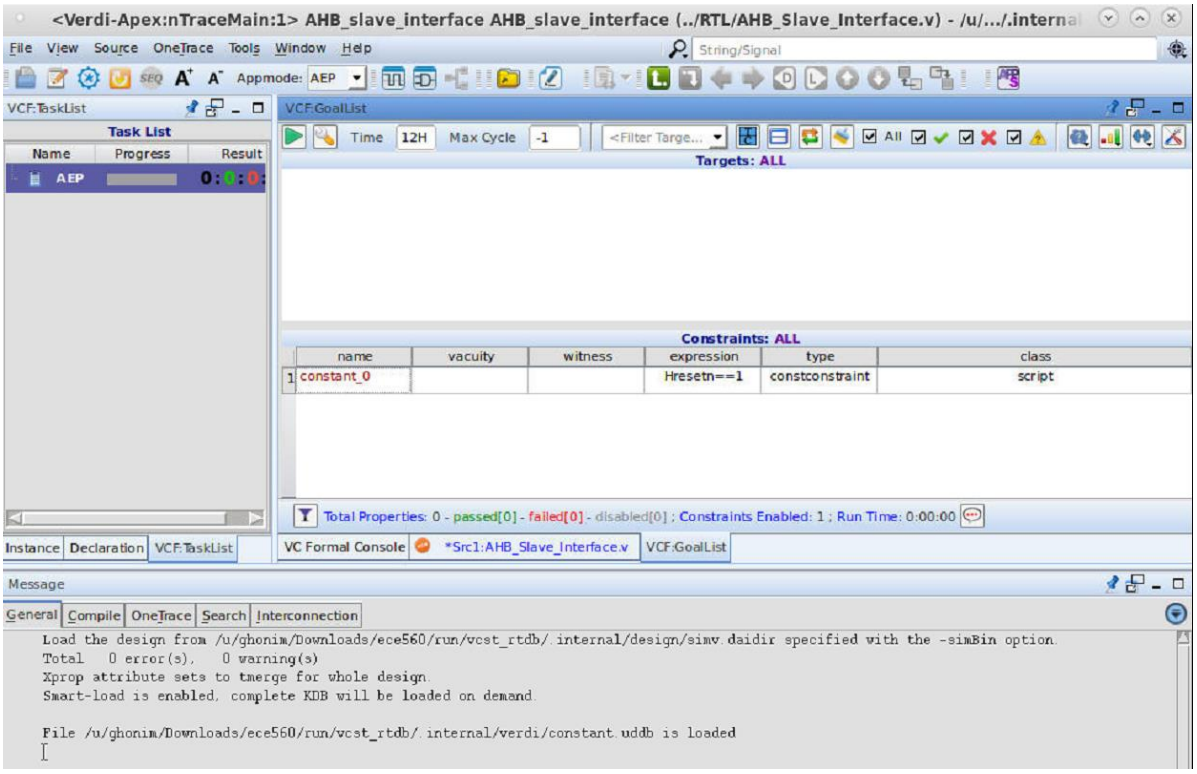
General Assumptions
HTRANS should be either 2'b10 or 2'b11 for SEQ, NONSEQ transfer type
when HWRITE is high next clock should have HWDATA as a valid value
HADDR should be in the range of peripherals address
when HWRITE is high HREADYIN should be high in that cycle and in following cycle
HREADYIN is high for 2 consecutive cycles later HREADYIN shouldn't be high until HREADYOUT[=2]

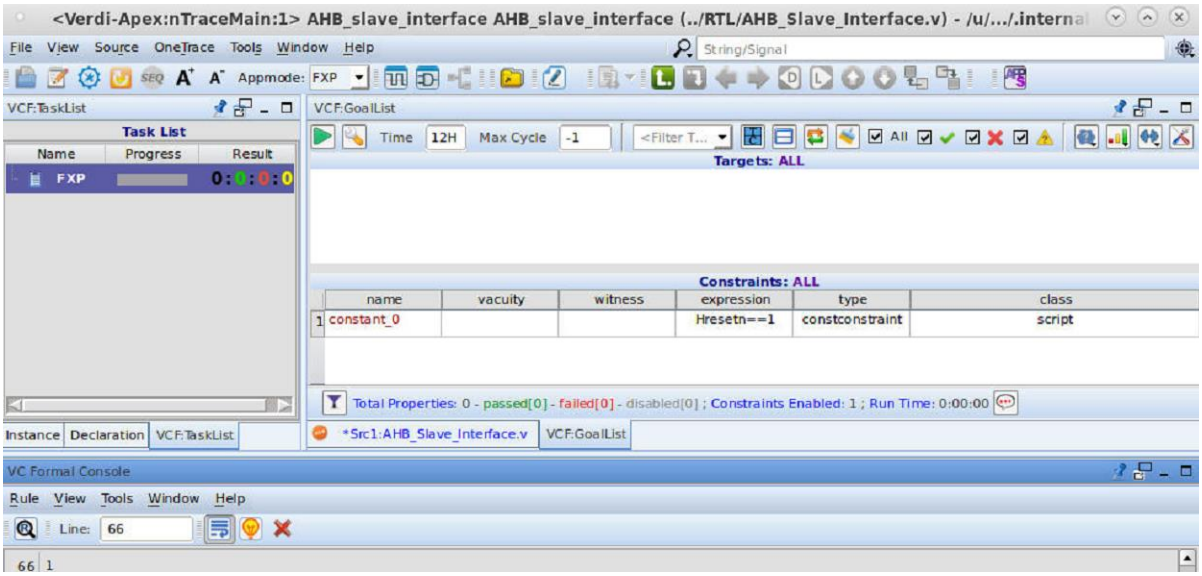
Newly Added Assumptions
For read transfers second HREADYIN shouldn't be high until HREADYOUT of 1st read is high

General Assertions
If HADDR is not in range of peripherals, PSEL should be 0
PADDR will hold its current value until the start of the next APB transfer

Cover Properties
Should support back-to-back writes for 2 different addresses
Should have Read after Read in 2 cycles (Burst of reads)
Should have back-to-back writes (Burst of writes)
Should have Read after Write (Back-to-back transfers)

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.

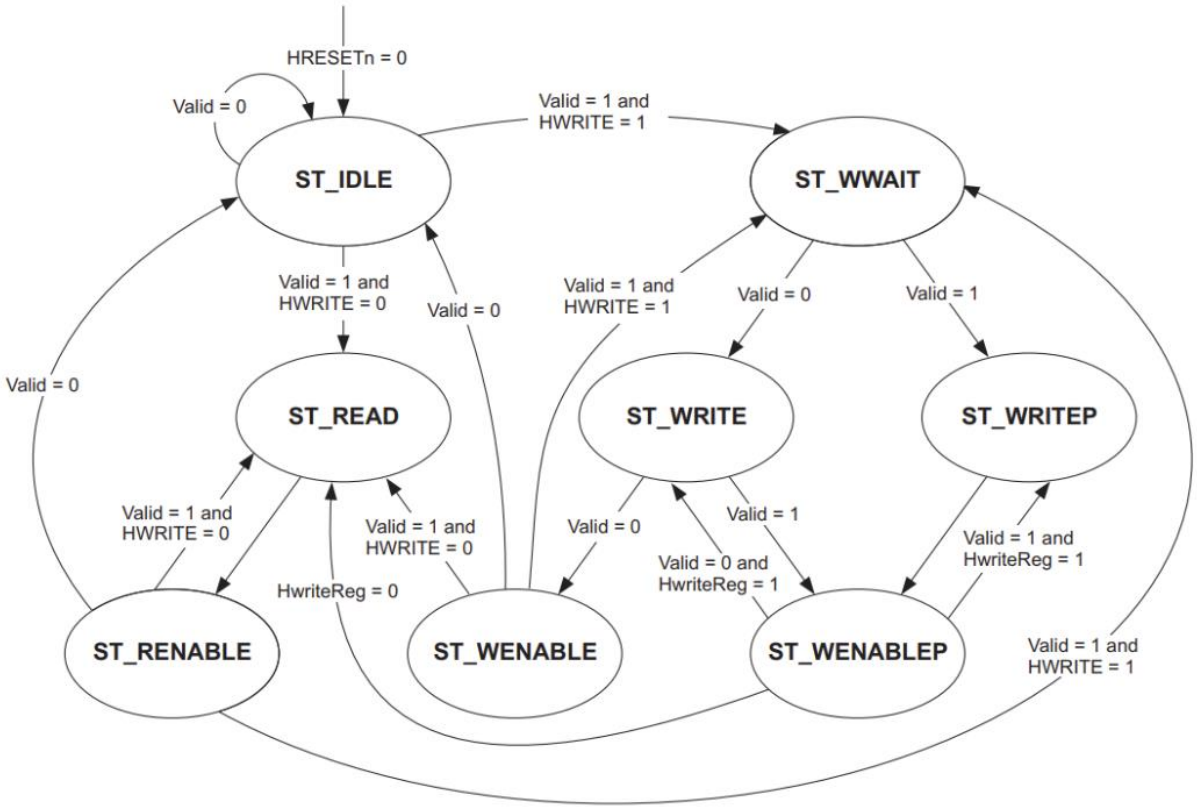




Ready and Write transfers.

State Machine for the AHB to APB interface.

We wanted to verify the state machine that controls the AHB to APB interface, and we wanted our assertions to be independent of the RTL code. We referenced the FSM from the AHB to APB ARM technical reference, and based on our signal names in the RTL as well as this FSM, we wrote 19 assertions covering every single transition.

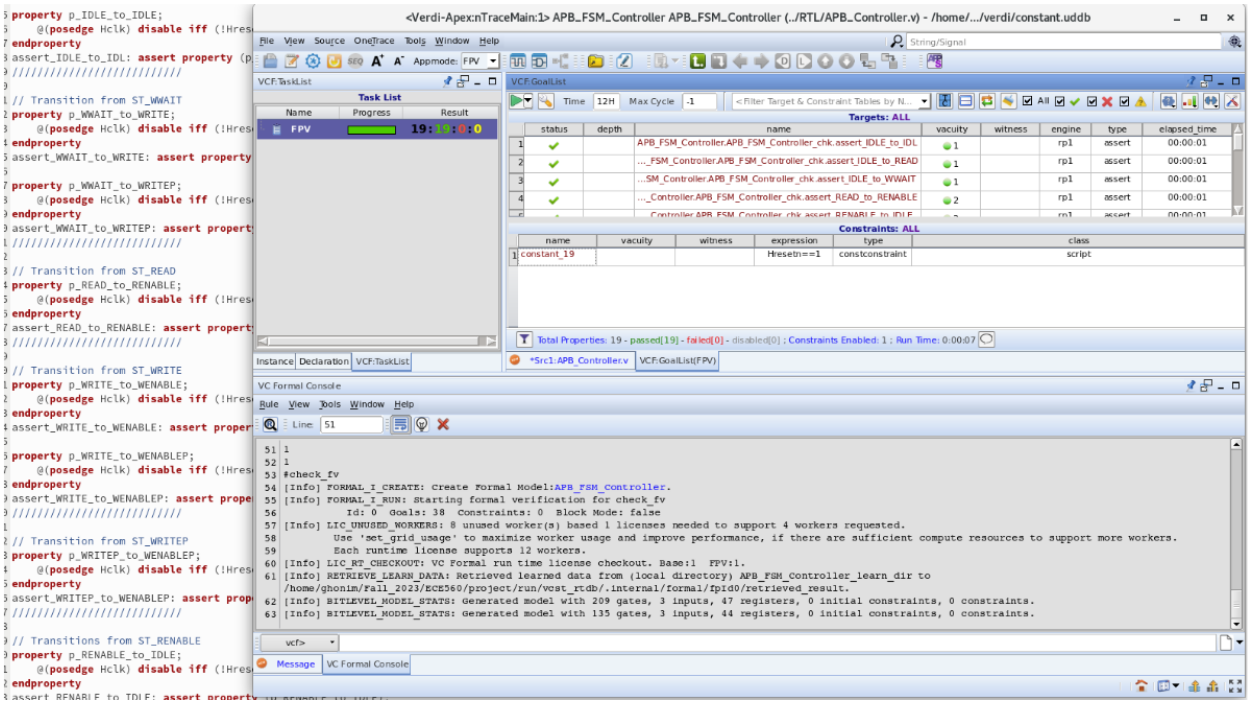



```
// Property to verify that PRESENT_STATE becomes NEXT_STATE one cycle later
property p_state_transition;
    @(posedge Hclk) disable iff (!Hresetn) 1 |-> ##2 PRESENT_STATE ==
$past(NEXT_STATE);
endproperty
assert_state_transition: assert property (p_state_transition);


// 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;
```

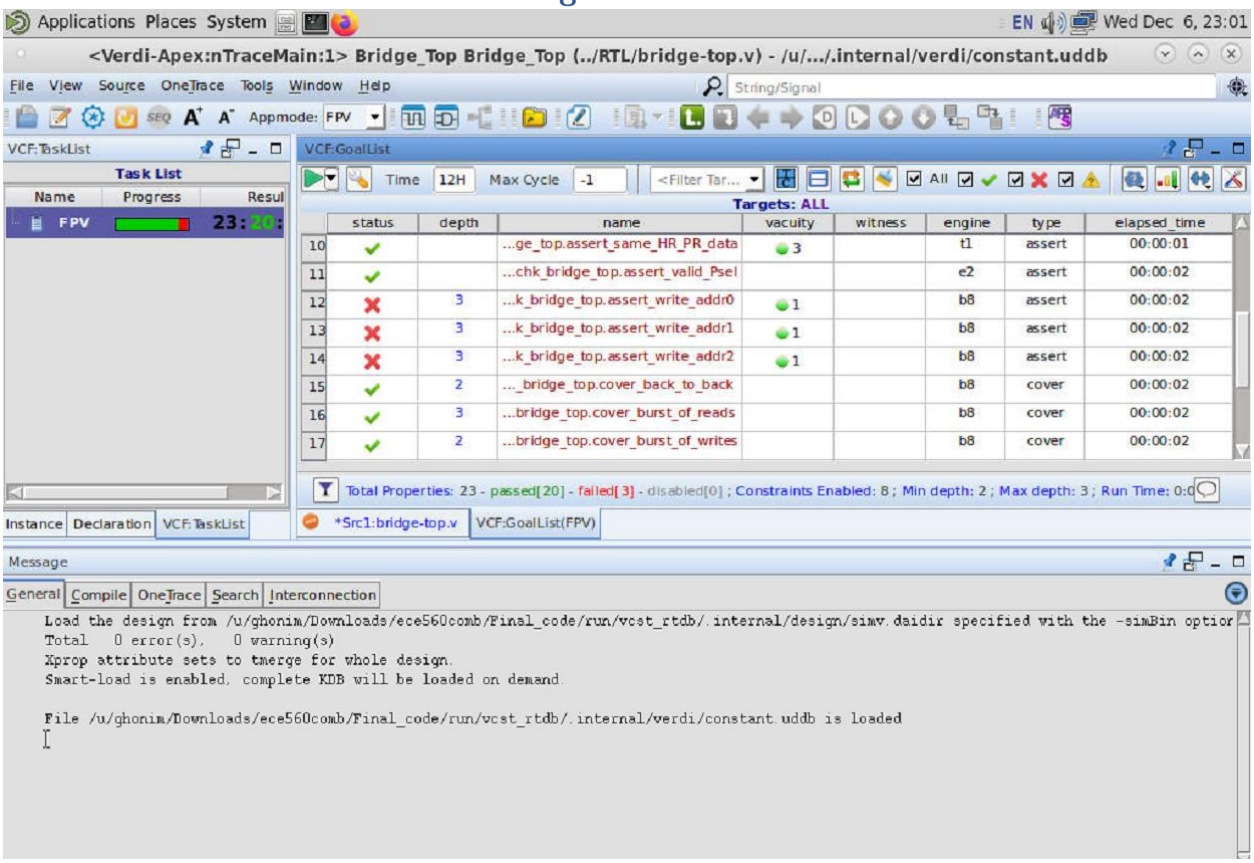
```
endproperty
assert_WRITE_to_WENABLE: assert property (p_WRITE_to_WENABLE);

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);

property p_RENABLE_to_WWAIT;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_RENABLE && valid
&& Hwrite |-> NEXT_STATE == ST_WWAIT;
endproperty
assert_RENABLE_to_WWAIT: assert property (p_RENABLE_to_WWAIT);

property p_RENABLE_to_READ;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_RENABLE && valid
&& !Hwrite |-> NEXT_STATE == ST_READ;
endproperty
assert_RENABLE_to_READ: assert property (p_RENABLE_to_READ);
////////////////////////////

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

```
property p_WENABLE_to_WWAIT;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WENABLE && valid
&& Hwrite |-> NEXT_STATE == ST_WWAIT;
endproperty
assert_WENABLE_to_WWAIT: assert property (p_WENABLE_to_WWAIT);

property p_WENABLE_to_READ;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WENABLE && valid
&& !Hwrite |-> NEXT_STATE == ST_READ;
endproperty
assert_WENABLE_to_READ: assert property (p_WENABLE_to_READ);
//////////////////////////

// Transitions from ST_WENABLEP
property p_WENABLEP_to_WRITE;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WENABLEP && !valid
&& Hwritereg |-> NEXT_STATE == ST_WRITE;
endproperty
assert_WENABLEP_to_WRITE: assert property (p_WENABLEP_to_WRITE);

property p_WENABLEP_to_WRITEP;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WENABLEP && valid
&& Hwritereg |-> NEXT_STATE == ST_WRITEP;
endproperty
assert_WENABLEP_to_WRITEP: assert property (p_WENABLEP_to_WRITEP);

property p_WENABLEP_to_READ;
    @(posedge Hclk) disable iff (!Hresetn) PRESENT_STATE == ST_WENABLEP &&
!Hwritereg |-> NEXT_STATE == ST_READ;
endproperty
assert_WENABLEP_to_READ: assert property (p_WENABLEP_to_READ);
```



All the 19 state transitions passed.

Read After Write and the PSLEX bug.



Actual Timing Diagram:

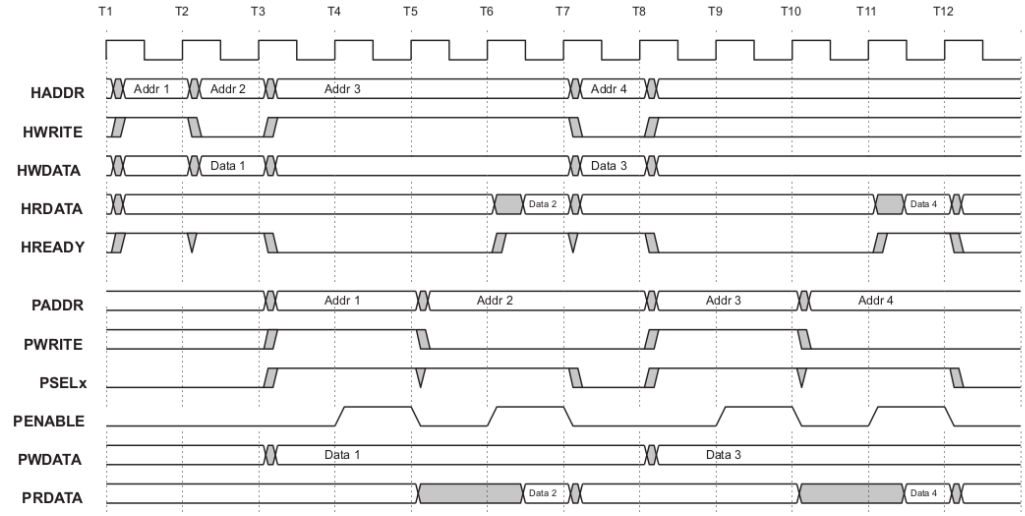
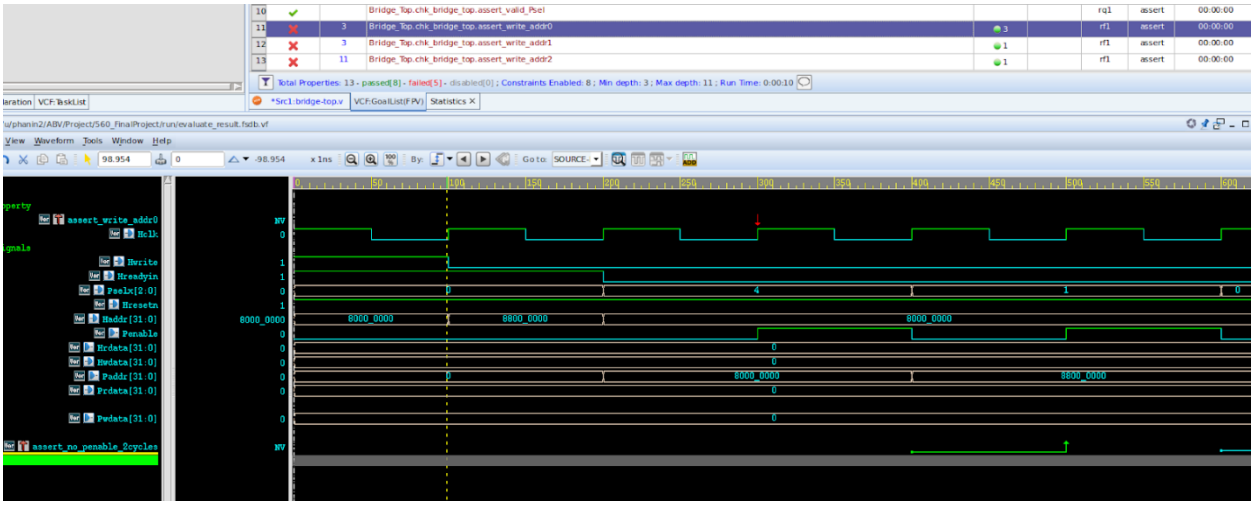


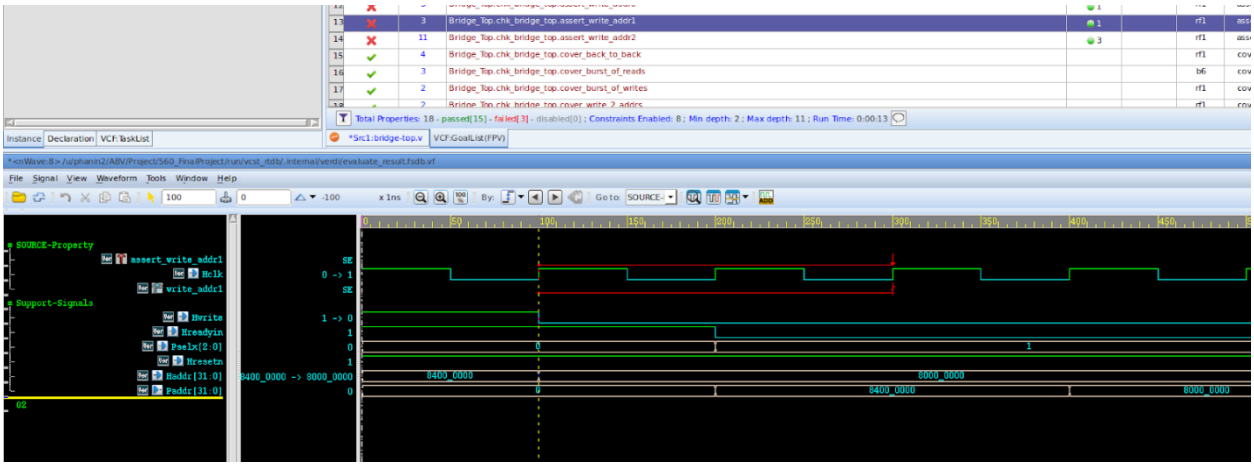
Figure 5-13 Back to back transfers

For Address range 8000_0000 to 8400_0000:



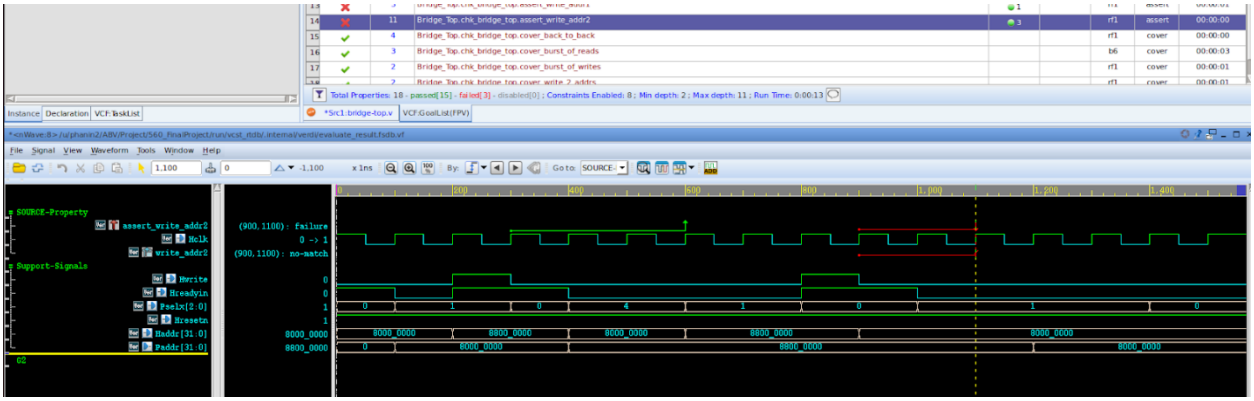
Here at Clock tick 100, we have a Write for address 8000_0000. At Clock tick 200 Read for address 8800_0000. And from clock tick 300 we have Paddr as 8000_0000 (correct) but Psel is 4 (wrong), which should have been 1 since address for write is in the range of 8000_0000 to 8400_0000.

For Address range 8400_0000 to 8800_0000:



Here at Clock tick 100, we have a Write for address 8400_0000. At Clock tick 200 Read for address 8000_0000. And from clock tick 300 we have Paddr as 8400_0000 (correct) but Psel is 1 (wrong), which should have been 2 since address for write is in the range of 8400_0000 to 8800_0000.

For Address range 8800_0000 to 8C00_0000:



Correct case: (When both Write and Read has same address)

At Clock tick 300, we have a Write for address 8800_0000. At Clock tick 200 Read for address 8800_0000. And at clock tick 1100 we have Paddr as 8800_0000 (correct) and Psel is 4(correct) since address for write is in the range of 8800_0000 to 8C00_0000.

Wrong case: (When both Write and Read has different addresses)

Here at Clock tick 900, we have a Write for address 8800_0000. At Clock tick 200 Read for address 8000_0000. And at clock tick 1100 we have Paddr as 8800_0000 (correct) but Psel is 1 (wrong), which should have been 4 since address for write is in the range of 8800_0000 to 8C00_0000.

To test for the read after write bug, we created an assertion that checked for a write sequence followed by a read sequence one cycle later. We then used the "past" function to verify that the RTL forced the address to stay the same for three cycles, as the specification noted that the read function requires three wait states when following a write signal. We were able to verify that this was the case with some difficulty. The hardest part was getting the address value from the previous clock cycles given that the sequences had a number of delays. After some work, we were able to make the assertion pass.

Additionally, through our investigation, we were able to identify a case where the Pselx signal did not conform to the AMBA specification. The output would occasionally transition to a value that was not one hot, resulting in the failure of the assertion.

Reset Signals

To check the reset signals, we remove the "disable iff" statements to allow the assertion to respond to resets. The antecedent is set to check if the reset signal rises and if the antecedent is triggered the code checks to make sure that the corresponding signal is set to 0, if appropriate.

Summary

This report outlines our verification efforts for the AHB-APB bridge, a critical link between AHB masters and slave interfaces. Our focus included ready and write transfers, the AHB to APB state machine, addressing Read After Write issues, and handling reset signals. We explored advanced verification methods and shared our findings of the PSELX signal. We used the AMBA specifications to write meaningful assumptions, assertions, and cover properties to cover the given RTL design as much as the RTL design and time allowed us.

Findings and future work

We found that the design was mostly bug free except for the PSELX bug which could actually cause a significant issue. We are glad that our assertions were able to catch that bug.

One of the difficulties we faced was finding a good, comprehensive RTL design to verify. The design we found implemented many of the AHB to APB Bridge functionalities and specifications but did not have a burst signal which would have allowed us to test the burst transactions. Future work would include finding, or even implementing, a bigger design with a burst signal which can be used to test the burst and wrap transactions.

Conclusion

In conclusion, the verification of the AHB-APB bridge design has provided us invaluable insights into the practical use of Formal Verification and SVA to use formal property verification.

Throughout the project, our team navigated through various challenges, from locating suitable RTL code that adhered to our verification criteria to crafting precise assertions and assumptions that rigorously evaluated the design's compliance with ARM AMBA standards. The collaborative approach we adopted, along with an organized file structure, facilitated effective teamwork and streamlined the development and verification processes.

As a result, we have gained a deeper understanding of the complexities and nuances associated with formal verification. The experience has not only honed our skills in creating and executing verification plans but has also underscored the importance of meticulous attention to detail in the pursuit of design integrity.