Portland State
UNIVERSITY

ECE560-ASSERTION BASED VERIFICATION

AHB2APB BRIDGE MODEL VERIFICATION

FINAL REPORT

NIKHITHA VADNALA
KEERTHANAA BHOOPATHY
SAI ROHITH REDDY YERRAM
SUHAIL AHAMED SUBAIRUDEEN

TEAM - 14

12/06/2024

TABLE OF CONTENTS

INTRODUCTION

This project and verification plan detail the strategies, methodologies, and results of our efforts to validate the functionality and performance of the AHB-APB bridge design. This bridge is pivotal as the interface enables seamless communication between the high-performance Advanced High-performance Bus (AHB) and the power-efficient Advanced Peripheral Bus (APB).

As semiconductor technology evolves, the demand for precise and reliable verification processes has become essential. Verifying the interaction of high-speed data transmission, memory mapping, and peripheral control requires a thorough and systematic approach. This document presents our comprehensive plan to evaluate the functionality of the AHB-APB bridge, ensuring its compliance with the specifications outlined in the ARM AMBA standard.

The verification process has been challenging and insightful. It involved creating targeted assertions, assumptions, and coverage properties to test the design rigorously. Our efforts are focused on ensuring the bridge operates as intended, meeting performance and efficiency expectations.

PROJECT OVERVIEW

Our final project revolved around assertion-based verification. We designed and executed a detailed validation plan to verify the functionality of the AHB-to-APB (AHB2APB) bridge design. This bridge serves as a critical interface, enabling seamless communication between the high-performance AHB and the power-efficient APB. To ensure adherence to industry standards, our approach was firmly rooted in the ARM AMBA specifications, which define the expected behaviors and operational requirements of the bridge.

The project involved analyzing the design specifications of the AHB2APB bridge and deriving a comprehensive set of properties to validate its functionality. These properties served as the foundation for crafting a suite of assertions, assumptions, and coverage properties aimed at evaluating key aspects of the bridge’s capabilities. These included handling burst transactions, proper signal propagation, memory mapping, and

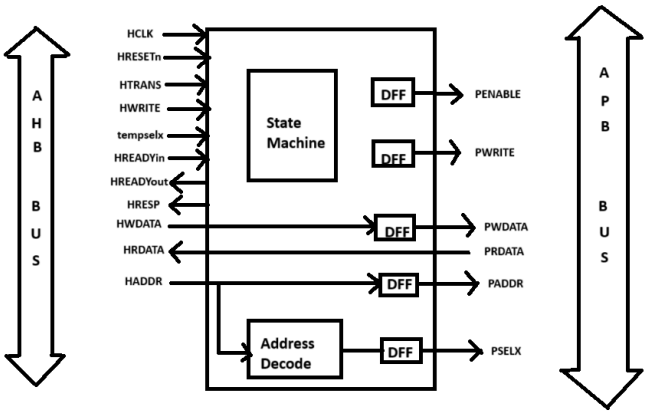
peripheral control. Despite our systematic approach, we encountered several challenges early in the project.

A significant challenge stemmed from the availability of Register Transfer Level (RTL) code that fully aligned with the ARM AMBA standards, particularly for supporting various burst read and write transactions. Ideally, the RTL code should encompass all the design features required for thorough validation. However, we worked with the RTL code that was available for the AHB2APB bridge design, which covered the core functionality of the bridge. This decision allowed us to move forward with the verification process, albeit with certain limitations. The RTL code utilized for this project is included as an attachment to this report for reference.

Through this project, we gained a deeper understanding of assertion-based verification, property-based validation, and the challenges of verifying complex bridge designs. This experience highlighted the importance of comprehensive RTL design and robust verification strategies in ensuring the reliability of semiconductor systems.

DESIGN DESCRIPTION

The AHB-to-APB (AHB2APB) bridge, developed by ARM, acts as a vital intermediary enabling seamless communication between the high-speed Advanced High-performance Bus (AHB) and the low-power Advanced Peripheral Bus (APB). Its primary role is to ensure efficient read and write operations between these two buses, which differ significantly in their performance and operational characteristics. This bridge is integral in designs where high-performance cores and memory systems interact with power-efficient peripheral devices.



Key Components:
AHB Slave Interface:

The AHB2APB bridge includes an AHB slave interface that receives addresses, control signals, and data from the AHB master. This interface handles all AHB-compliant transactions, such as burst, single, sequential, and non-sequential data transfers, and ensures that these operations are properly interpreted for the APB side.

APB Transfer State Machine:

At the core of the bridge is the APB transfer state machine, which is responsible for controlling the transfer of signals to and from APB peripherals. This state machine operates independently of the device's memory map, enabling efficient management of peripheral communication without being constrained by specific address spaces.

Signal Generation Mechanisms:

The bridge includes mechanisms for generating APB-compatible output signals such as PADDR (address), PWDATA (write data), and PRDATA (read data). It also ensures proper timing and synchronization for signals like PENABLE, PWRITE, and PSELx.

Functionality:
Address and Control Signal Management:

The bridge stores incoming addresses and control signals from the AHB, converting them into the simplified protocol of the APB. This includes translating complex burst and pipelined AHB transactions into single, sequential transactions as per APB specifications.

Data Path Management:

The data flow between the two buses is managed via two distinct channels:
Read Data Path (PRDATA): Transfers data read from APB peripherals back to the AHB master.
Write Data Path (PWDATA): Handles data written from the AHB master to APB peripherals.

Transfer Flexibility:

The bridge is designed to handle both sequential and non-sequential transfers of varying sizes (defined by HSIZE). This adaptability ensures compatibility with a wide range of peripherals, regardless of their specific data transfer requirements.

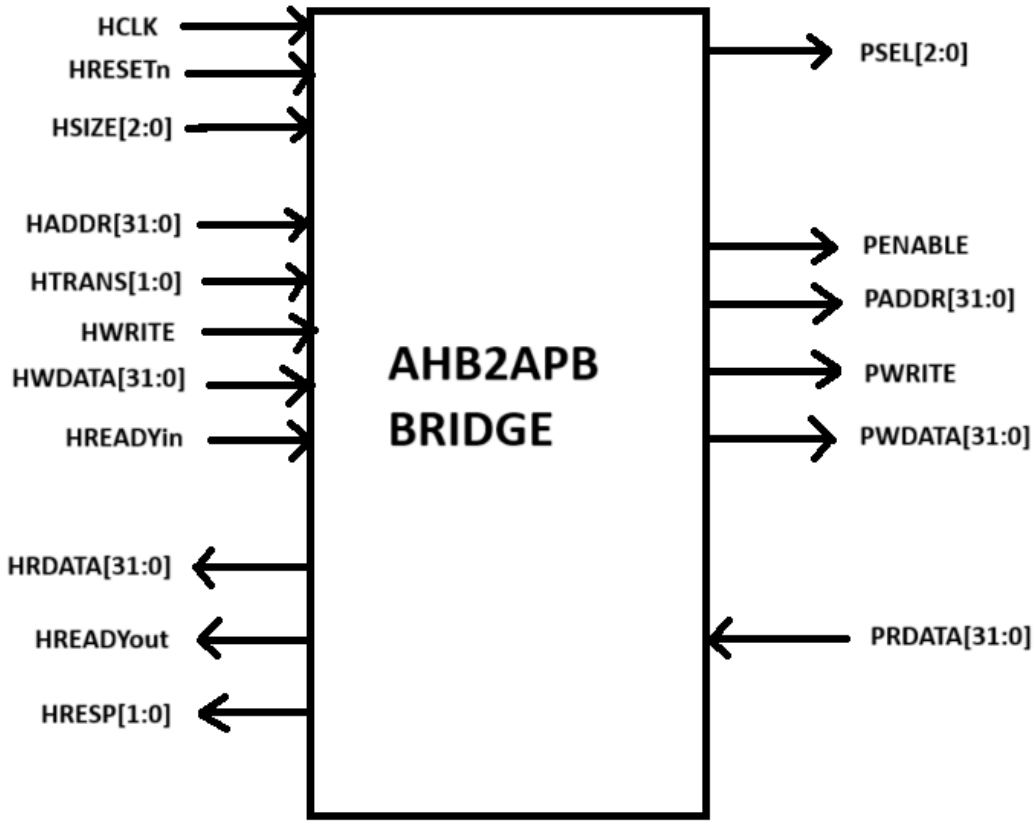
Advantages:

The AHB2APB bridge bridges the gap between high-speed and low-speed buses, providing a versatile and efficient communication pathway. By supporting a broad range of transaction types and data sizes while adhering to the simplified APB protocol, the bridge ensures low latency and high reliability. It also minimizes power consumption for APB peripherals, making it a critical component in embedded systems, SoCs, and other designs requiring efficient inter-bus communication.

SPECIFICATIONS
DESIGN OVERVIEW

Our customized AHB-to-APB (AHB2APB) bridge design facilitates seamless communication between the Advanced High-performance Bus (AHB) and the Advanced Peripheral Bus (APB). This design has been tailored to meet specific requirements, ensuring compatibility and efficiency in transactions between AHB masters and APB slaves while adhering to our design specifications.

The accompanying diagram illustrates the input and output signals involved in the AHB-to-APB bridge. Many of these signals are implemented as input and output pins in the RTL design used for this project. Given the protocol's nature as a bridge, understanding the functionality and specifications of these signals is vital for developing effective properties and verifying the bridge's correct operation.



Signal Overview
AHB Side - Input Signals:

HCLK: The clock signal drives the AHB interface.

HRESETn: Active-low reset signal for the AHB interface.

HSIZE[2:0]: Defines the size of the data transfer on the AHB interface.

HADDR[31:0]: Provides the address for AHB transfers.

HTRANS[1:0]: Specifies the type of transfer occurring on the AHB interface.

HWRITE: Indicates whether the AHB operation is a written transaction.

HWDATA[31:0]: Contains the data to be written during a write operation.
HREADYin: Signals the availability of the AHB interface for new transactions.

AHB Side - Output Signals:

HRDATA[31:0]: Outputs the read data from the AHB interface.

HREADYout: Indicates that the AHB interface is ready for the next operation.

HRESP[1:0]: Conveys the status of the AHB transfer (e.g., success or error).

APB Side - Input Signals:

PRDATA[31:0]: Provides read data from the APB interface.

APB Side - Output Signals:

PSEL[2:0]: Selects the APB slave for the current transaction.

PENABLE: Enables data transfer on the APB interface.

PADDR[31:0]: Provides the address for the APB transfer.

PWRITE: Indicates whether the APB operation is a write transaction.

PWDATA[31:0]: Contains the data to be written to the APB interface.

Functionality

These signals create the communication pathways between the AHB and APB interfaces, enabling the transfer of data and control information. The design ensures proper synchronization and functionality, allowing the AHB2APB bridge to facilitate efficient and reliable interactions between the high-performance AHB and the power-efficient APB.

AHB Signals

Signal Name	Source	Description
HCLK Bus clock	Clock Source	This clock times all bus transfers. All signal timings are related to the rising edge of **HCLK.**
HRESETn Reset	Reset Controller	The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW signal.
HADDR[31:0] Address bus	Master	The 32-bit system address bus.
HTRANS[1:0] Transfer type	Master	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE Transfer direction	Master	When HIGH this signal indicates a write transfer and when LOW a read transfer.
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.
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.
HWDATA[31:0] Write data bus	Master	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HRDATA[31:0] Read data bus	Slave	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HREADY Transfer done	Slave	When HIGH the **HREADY signal indicates that a** transfer has finished on the bus. This signal may be driven LOW to extend a transfer. Note: Slaves on the bus require **HREADY as both** an input and an output signal.
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.

APB Signals

Signal Name	Description
PCLK Bus Clock	This clock times all bus transfers. Both the LOW phase and HIGH phase of PCLK are used to control transfers.
PRESETn APB Reset	The bus reset signal is active LOW and is used to reset the system.
PENABLE APB Strobe	This strobe signal is used to time all accesses on the peripheral bus. The enable signal is used to indicate the second cycle of anAPB transfer. The rising edge of PENABLE occurs in the middle of the APB transfer.
PADDR[31:0] APB address bus	This is the APB address bus, which may be up to 32-bits wide and is driven by the peripheral bus bridge unit.
PWRITE APB transfer direction	When HIGH this signal indicates an APB write access and when LOW a read access.
PRDATA APB read data bus	The read data bus is driven by the selected slave during read cycles (when PWRITE is LOW). The read data bus can be up to 32-bits wide.
PWDATA APB write data bus	The write data bus is driven by the peripheral bus bridge unit during write cycles (when PWRITE is HIGH). The write data bus can be up to 32-bits wide.
PSELx APB select	A signal from the secondary decoder, within the peripheral bus bridge unit, to each peripheral bus slave x. This signal indicates that the slave device is selected and a data transfer is required. There is a PSELx signal for each bus slave.

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).

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.

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.

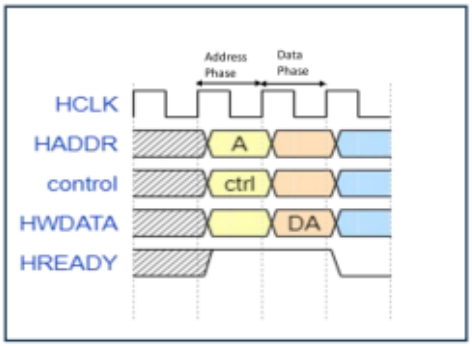
When it is necessary for a slave to insert a number of wait states prior to deciding what response will be given then it must drive the response to OKAY.

TIMING DIAGRAMS:

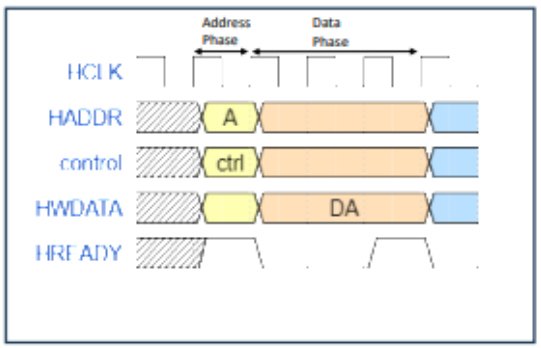
Read after Write

	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2		Cycle1	Cycle2	Cycle3	Cycle4	Cycle5	Cycle6	Cycle7	Cycle8	Cycle9	Cycle10	Cycle11	Cycle12
3	HADDR	32'bz	32'b80000002	32'b80000003	32'b80000004	32'b80000004	32'b80000004	32'b80000004	32'b80000017				
4	HWRITE	1'bx	1'b1	1'b0	1'b1	1'b1	1'b1	1'b1	1'b0	1'b1	1'b1	1'b1	1'b1
5	HWDATA	32'bz	32'bz	32'b00000032	32'bz	32'bz	32'bz	32'bz	32'b00000065	32'bz	32'bz	32'bz	32'bz
6	HRDATA	32'bz	32'bz	32'bz	32'bz	32'bz	32'bz	32'b000000a7	32'bz	32'bz	32'bz	32'bz	32'b000000B2
7	HREADY	1'bx	1'b1	1'b1	1'b0	1'b0	1'b0	1'b1	1'b1				
8	PADDR	32'bx	32'bx	32'bx	32'b80000002	32'b80000002	32'b80000003	32'b80000003	32'b80000003	32'b80000004	32'b80000004	32'b80000017	32'b80000017
9	PWRITE	1'bx	1'bx	1'bx	1'b1	1'b1	1'b0	1'b0	1'b0	1'b1	1'b1		
10	PSELX	1'bx	1'bx	1'bx	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
11	PENABLE	1'bx	1'bx	1'bx	1'b0	1'b1	1'b0	1'b1	1'b0	1'b0	1'b1		
12	PWDATA	32'bx	32'bx	32'bx	32'b00000032	32'b00000032	32'b00000032	32'b00000032	32'b000000032	32'b00000065	32'b00000065	32'b00000065	32'b00000065
13	PRDATA	32'bx	32'bx	32'bx	32'bx	32'bx	32'bx	32'b000000a7	32'bx	32'bx	32'bx	32'bx	32'b000000B2
14													
15													

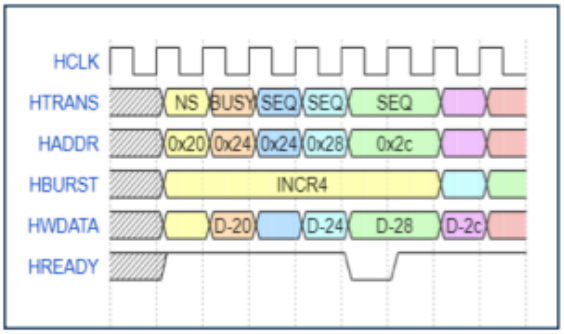
AHB:



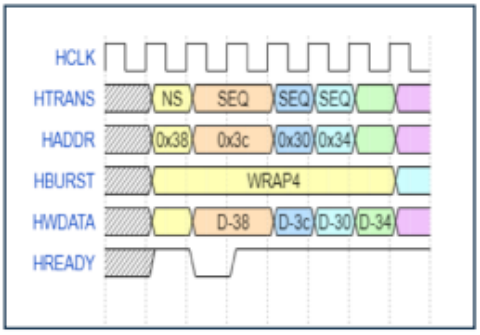
Basic AHB Transfer



Transfer with wait state

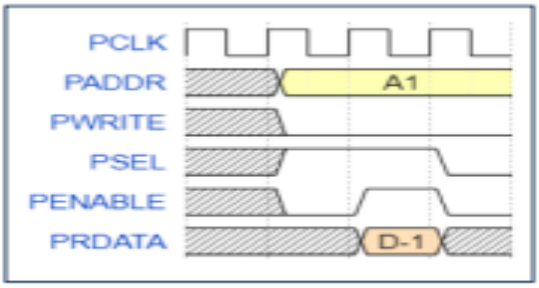


Burst Transfer(Increment)- Pipelined Opereration

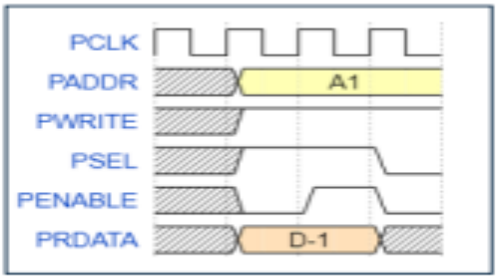


Burst Transfer(Wrapping) – Pipelined Operation

APB:

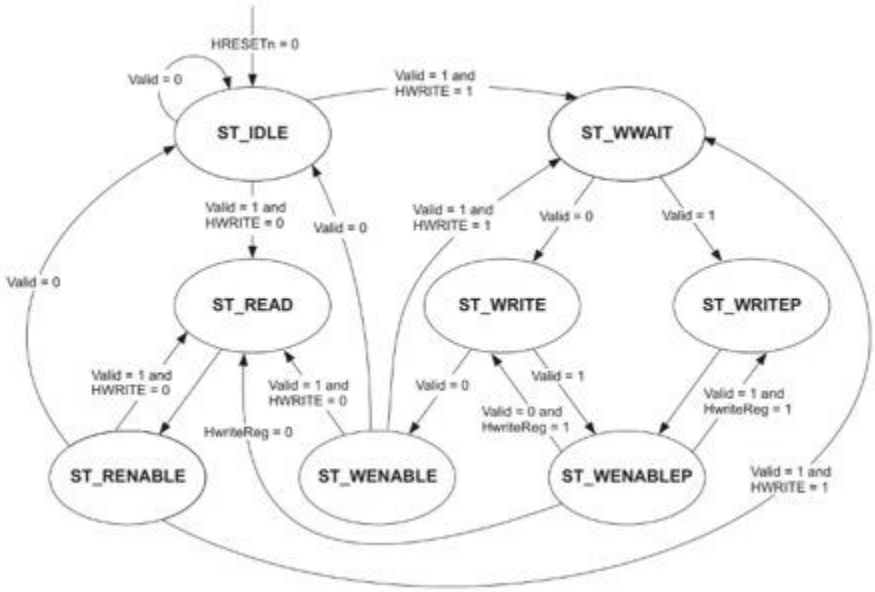


Read Transfer



Write Transfer

FINITE STATE MACHINE:



SIMULATION:

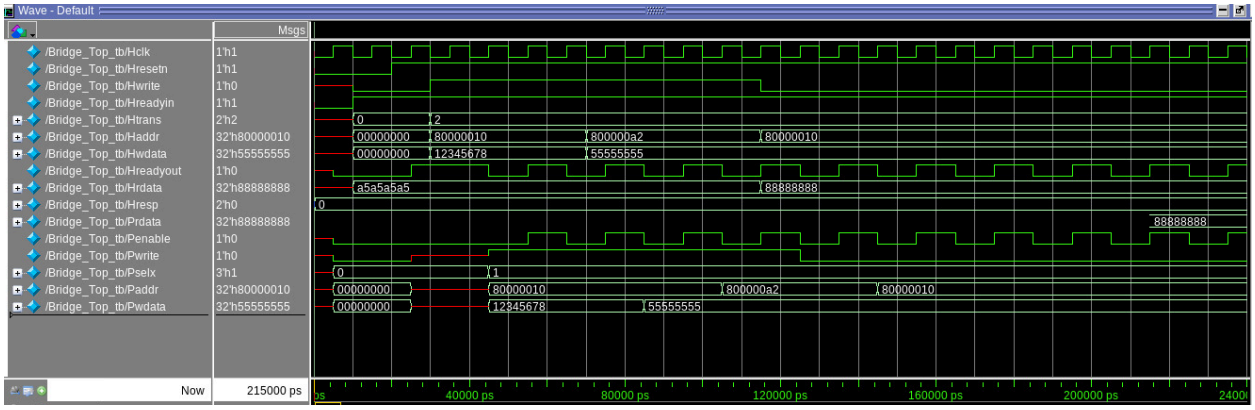
We have compiled all the design files and simulated them with a simple testbench to make sure that write and read transactions happen between AHB and APB.

TRANSCRIPT:

```
# Hresetn = 0,Hreadyin = 0, Hwrite = x, Htrans = x, Haddr = xxxxxxxx, Hwdata = xxxxxxxx, Paddr = xxxxxxxx, Pwdata = xxxxxxxx, Prdata = xxxxxxxx, Hrdata = xxxxxxxx
# Hresetn = 0,Hreadyin = 0, Hwrite = x, Htrans = x, Haddr = xxxxxxxx, Hwdata = xxxxxxxx, Paddr = 00000000, Pwdata = 00000000, Prdata = xxxxxxxx, Hrdata = xxxxxxxx
# Hresetn = 0,Hreadyin = 1, Hwrite = 0, Htrans = 0, Haddr = 00000000, Hwdata = 00000000, Paddr = 00000000, Pwdata = 00000000, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 0, Htrans = 0, Haddr = 00000000, Hwdata = 00000000, Paddr = 00000000, Pwdata = 00000000, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 0, Htrans = 0, Haddr = 00000000, Hwdata = 00000000, Paddr = xxxxxxxx, Pwdata = xxxxxxxx, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 1, Htrans = 2, Haddr = 80000010, Hwdata = 12345678, Paddr = xxxxxxxx, Pwdata = xxxxxxxx, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 1, Htrans = 2, Haddr = 80000010, Hwdata = 12345678, Paddr = 80000010, Pwdata = 12345678, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 1, Htrans = 2, Haddr = 800000a2, Hwdata = 55555555, Paddr = 80000010, Pwdata = 12345678, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 1, Htrans = 2, Haddr = 800000a2, Hwdata = 55555555, Paddr = 80000010, Pwdata = 55555555, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 1, Htrans = 2, Haddr = 800000a2, Hwdata = 55555555, Paddr = 800000a2, Pwdata = 55555555, Prdata = a5a5a5a5, Hrdata = a5a5a5a5
# Hresetn = 1,Hreadyin = 1, Hwrite = 0, Htrans = 2, Haddr = 80000010, Hwdata = 55555555, Paddr = 800000a2, Pwdata = 55555555, Prdata = 88888888, Hrdata = 88888888
# Hresetn = 1,Hreadyin = 1, Hwrite = 0, Htrans = 2, Haddr = 80000010, Hwdata = 55555555, Paddr = 80000010, Pwdata = 55555555, Prdata = 88888888, Hrdata = 88888888
```

We are monitoring all the top module important signals to make sure transactions are happening as defined.

WAVEFORM:



We are sending data from AHB to a valid address using non-sequential bus transactions and we can observe the data at the APB side. After some delay, we are trying to read the data from the slave (APB) by using PRdata and HRdata signals. From the simulated output, we can tell that the AHB2APB bridge model's basic functionality is working fine. Now we are going to verify this design formally by running the design in 3 different modes including AEP, FXP, and FPV.

FORMAL VERIFICATION:

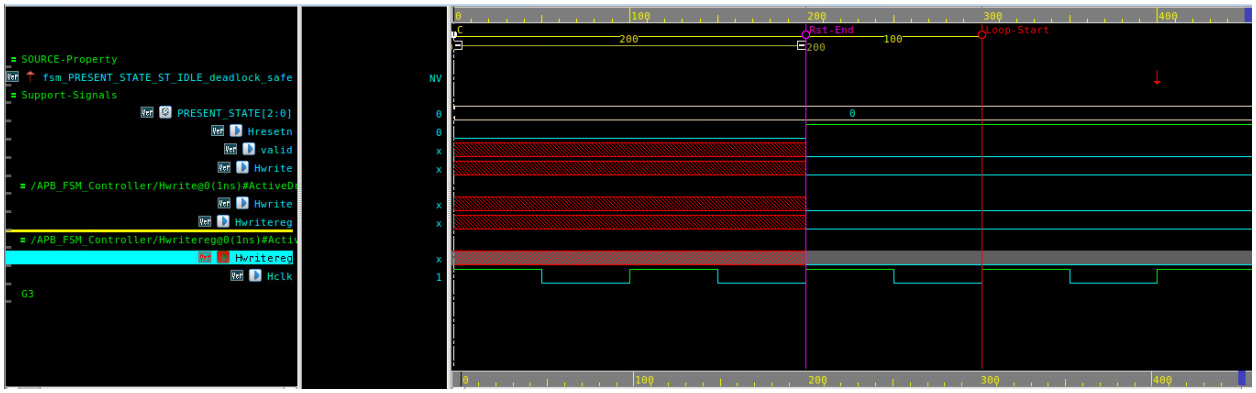
Automatically Extracted Properties (AEP) in formal verification are properties inferred by verification tools directly from the design under test (DUT) without manual input. These properties include safety checks, liveness conditions, state machine integrity, and signal stability, ensuring basic correctness, such as avoiding illegal states or unreachable conditions. AEP simplifies the verification process by reducing manual effort, providing comprehensive design coverage, and identifying potential issues through counterexamples. However, it complements rather than replaces manually written properties, as it may not fully capture the design's functional intent.

AEP for Bridge Top Module:

	status	depth	name	type	location	expression	state_reg	state_name	state_val	engine	elapsed_time
1	✗ 2	2	...T_STATE_ST_IDLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_IDLE	'h0	I8	00:00:05
2	✓		..._STATE_ST_READ_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_READ	'h2	t1	00:00:07
3	✓		...ATE_ST_RENABLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_RENABLE	'h5	t1	00:00:07
4	✓		...E_ST_WENABLEP_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WENABLEP	'h7	t1	00:00:07
5	✓		...TE_ST_WENABLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WENABLE	'h6	t1	00:00:07
6	✓		...TATE_ST_WRITEP_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WRITEP	'h4	t1	00:00:07
7	✓		...STATE_ST_WRITE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WRITE	'h3	t1	00:00:07
8	✓		...STATE_ST_WWAIT_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WWAIT	'h1	t1	00:00:07

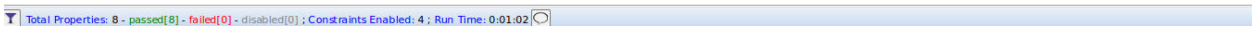
AEP generates 8 properties out of which 7 are passing and 1 is getting failed due to deadlock conditions. After adding **aep_generate -type fsm_fairness** command in .tcl file our deadlock condition is resolved.

CEX Waveform:



Now we can see all the properties are getting passed dut to fsm_fairness command.

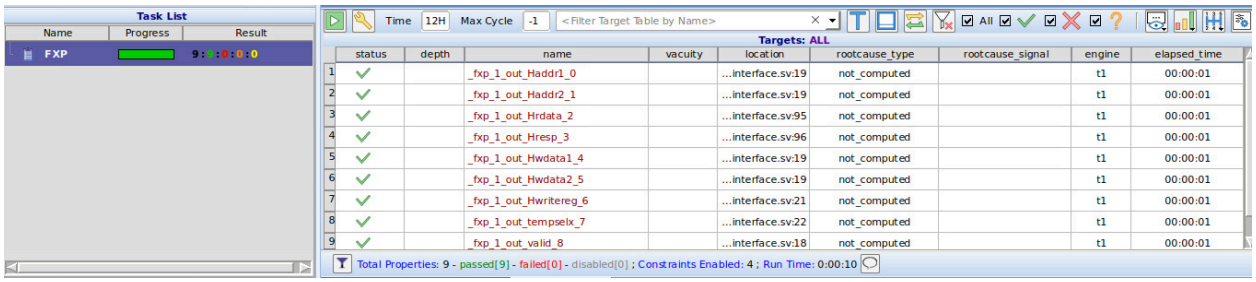
status	depth	name	type	location	expression	state_reg	state_name	state_val	engine	elapsed_time
✓		...T_STATE_ST_IDLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_IDLE	'h0	I5	00:00:48
✓		..._STATE_ST_READ_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_READ	'h2	I5	00:00:46
✓		...ATE_ST_RENABLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_RENABLE	'h5	I5	00:00:47
✓		...E_ST_WENABLEP_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WENABLEP	'h7	I5	00:00:47
✓		...TE_ST_WENABLE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WENABLE	'h6	I5	00:00:50
✓		...TATE_ST_WRITEP_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WRITEP	'h4	I5	00:00:50
✓		...STATE_ST_WRITE_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WRITE	'h3	I5	00:00:51
✓		...STATE_ST_WWAIT_deadlock_safe	fsm_deadlock	...ontroller.sv:40		PRESENT_STATE	ST_WWAIT	'h1	I5	00:00:51



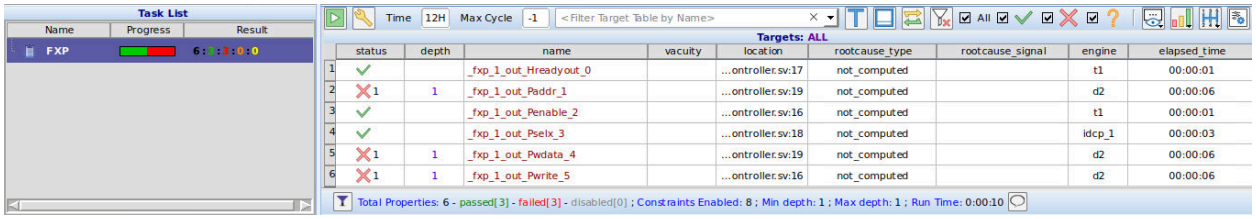
Formal X-Propagation (FXP) in formal verification is a methodology used to identify and analyze the propagation of unknown values ('X') in digital designs. Unknown states can arise from uninitialized registers, clock gating, low-power modes, or

non-deterministic behaviors. FXP ensures that these 'X' values do not propagate in ways that could lead to functional issues, such as masking design errors or causing metastability. By leveraging formal tools, FXP exhaustively checks all potential scenarios where 'X' values may propagate, helping to detect hidden bugs and improve design robustness, especially in safety-critical applications. This approach enhances confidence in the design's reliability across all possible operating conditions.

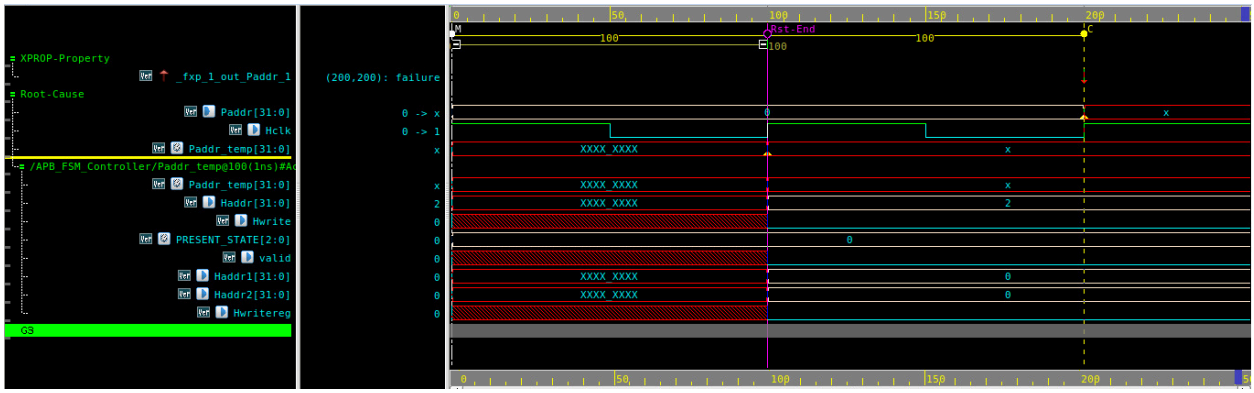
FXP for AHB_slave_interface module:
It generates multiple properties and all of them are passed. This implies there is no X-propagation in the AHB module.



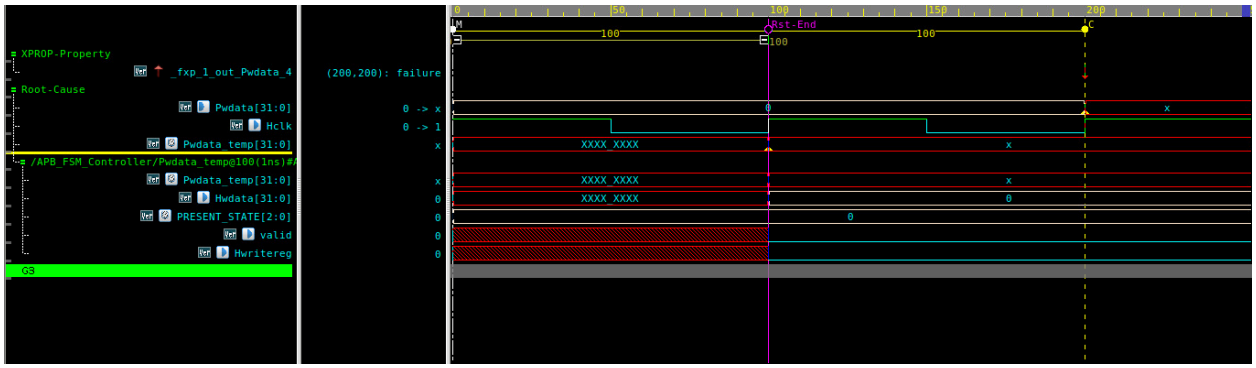
FXP for APB_FSM_Controller module
It generates 6 properties out of which 3 failed due to X-propagation in the design. We are going to fix by finding the root cause and CEX.



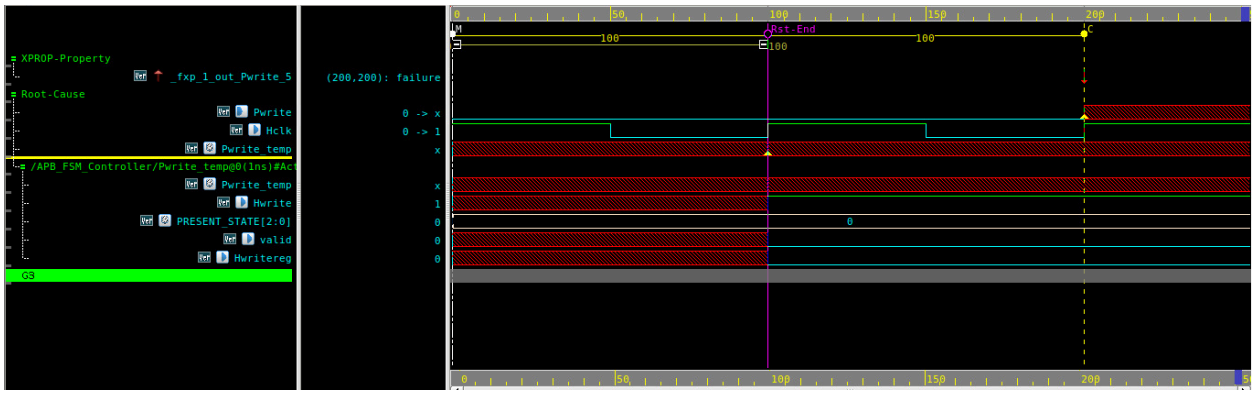
Paddr signal is getting X-propagation



PWdata signal is getting X-propagation



Pwrite signal is getting X-propagation

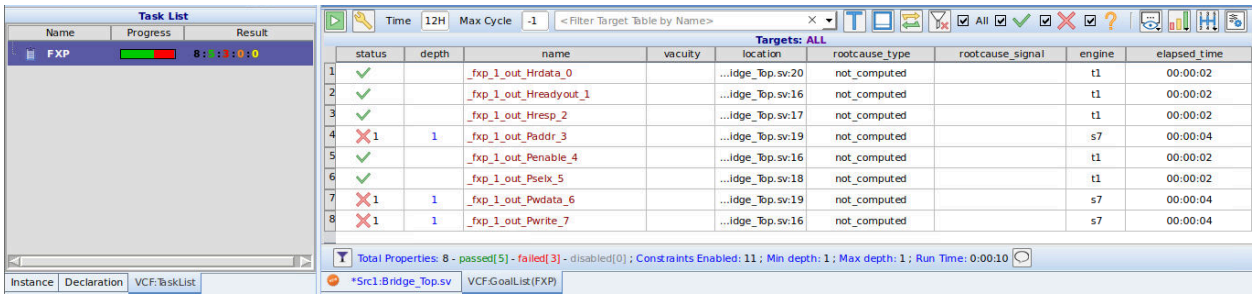


After initializing all three signals to zero in the design we can see all the properties are getting passed.

```
always @(*)
 begin:OUTPUT_COMBINATIONAL_LOGIC
        //Paddr='0;
        //Pwdata='0;
        //Pwrite='0;
   case(PRESENT_STATE)
```

All the passing properties

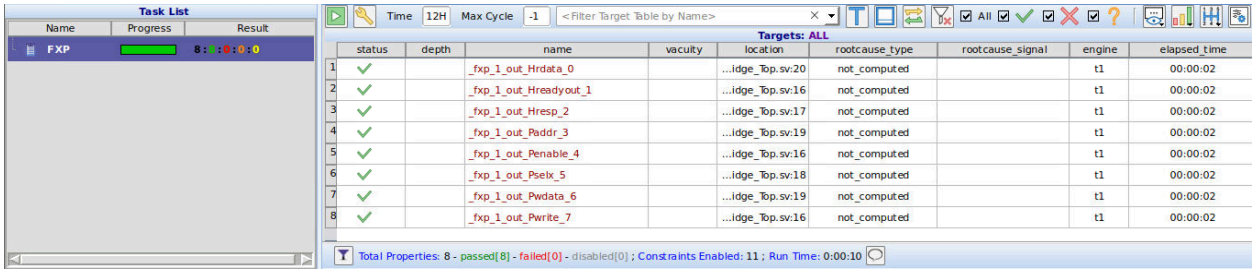
FXP for Bridge_Top module:
The same 3 signal properties are failing in the top module as well. We are going to initialize them to zero and remove the X-propagation.



Initializing,

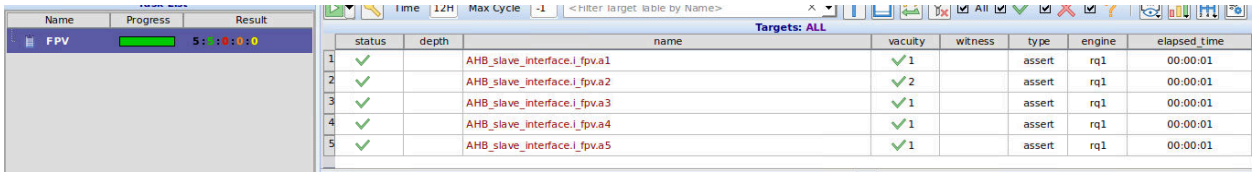
```
always @(*)
 begin:OUTPUT_COMBINATIONAL_LOGIC
        //Paddr='0;
        //Pwdata='0;
        //Pwrite='0;
   case(PRESENT_STATE)
```

All the properties are not getting passed and we can conclude that there in no X-propagation in the design

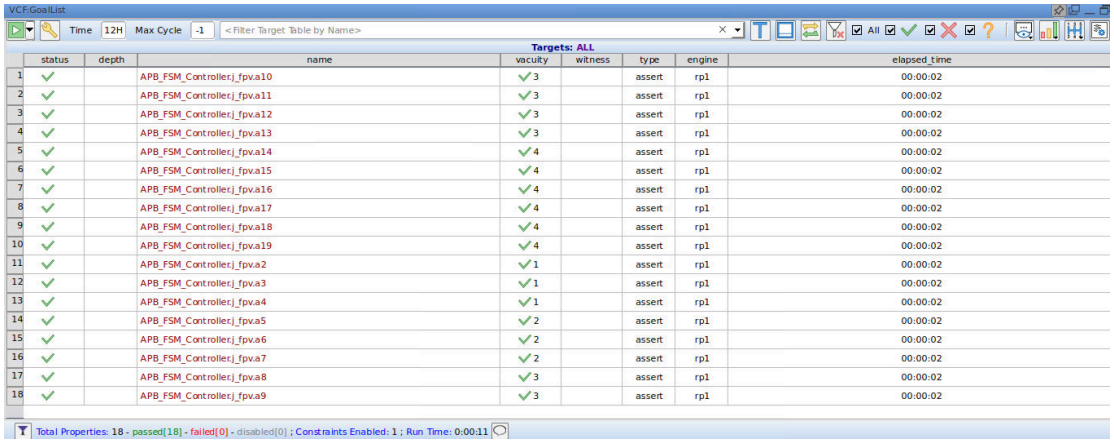


Formal Property Verification (FPV) is a method in formal verification where specified properties of a digital design are mathematically proven to hold across all possible input scenarios and states. These properties, written in languages like SystemVerilog Assertions (SVA), define the expected behavior of the design, such as protocol compliance, data integrity, or safety requirements. FPV uses exhaustive state-space exploration without requiring test benches, enabling it to catch corner-case bugs that traditional simulation might miss. It is particularly useful for validating critical functionalities and ensuring that the design meets its specifications under all conditions.

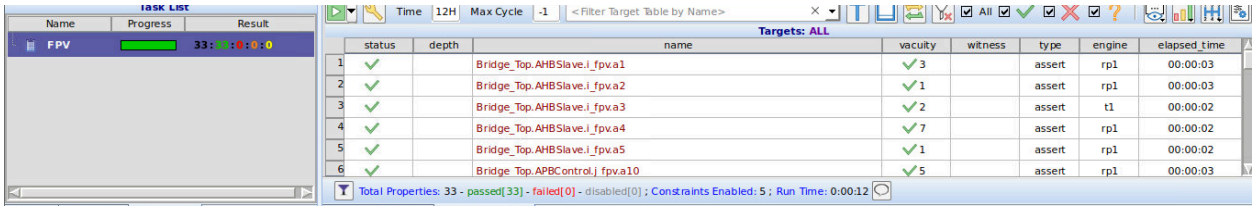
FPV for AHB slave module:
We wrote five assertions for the AHB module and all of the properties are passing in FPV mode.



FPV for APB FSM module:
We wrote eighteen assertions that cover all the state transitions and all the properties are covered.



FPV for Bridge_Top module:
We wrote ten assertions and this will take all the design assertions and all the properties are getting passed



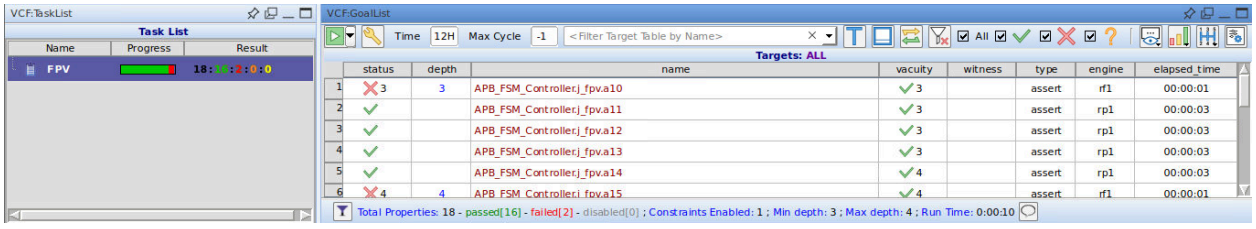
BUG INJECTION

We intentionally added two bugs to the FSM controller and two of the assertions found that bug and we can easily fix the bugs.

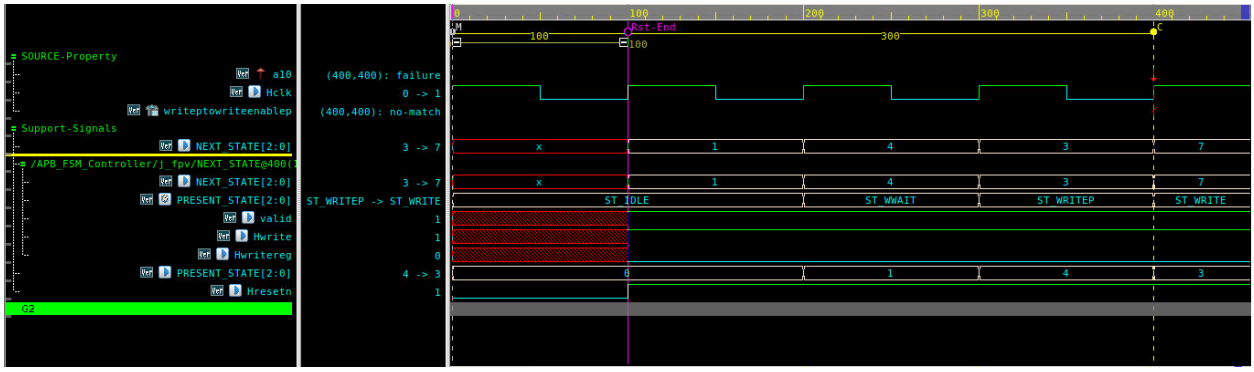
```
ST_WRITEP:begin
      //NEXT_STATE=ST_WENABLEP;
      NEXT_STATE=ST_WRITE;//bug injected
     end
```

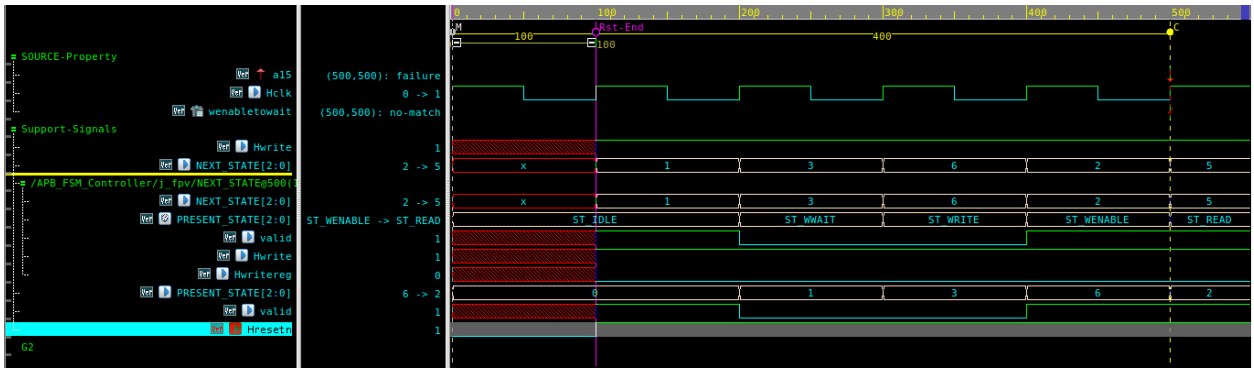
```
ST_WENABLE:begin
        if (~valid)
         NEXT_STATE=ST_IDLE;
        //else if (valid && Hwrite)
        else if (valid && Hwrite && Hwritereg) //bug injected
         NEXT_STATE=ST_WWAIT;
        else
         NEXT_STATE=ST_READ;
       end
```

Out of 18 state transitions 2 assertions are getting failed, by fixing the bug in the design We can make the properties passed.



CEX for failing properties





ASSUMPTIONS:

```
//address range
property addressrange;
  @(posedge Hclk) disable iff(!Hresetn) (Haddr >= 32'h8000_0000 && Haddr < 32'h8C00_0000);
endproperty

//valid Hwdata
property validhwdata;
  @(posedge Hclk) disable iff(!Hresetn) (Hwrite |=> !$isunknown(Hwdata));
endproperty

//valid Htrans
property validhtrans;
  @(posedge Hclk) disable iff(!Hresetn) (Htrans == 2'b10 || 2'b11);
endproperty

//hwrite and hreadyin
property validhreadyin;
  @(posedge Hclk) disable iff(!Hresetn) (Hwrite |-> Hreadyin ##1 Hreadyin);
endproperty
```

FUTURE WORK

Our analysis of the AHB-to-APB bridge design revealed that the implementation was largely bug-free, demonstrating reliable functionality in most aspects. However, we identified a critical issue related to the PSELX signal, which had the potential to cause significant problems during operation. Fortunately, our assertions proved effective in identifying this bug, underscoring the importance of assertion-based verification in catching such critical errors early in the development process.

One of the primary challenges we encountered was sourcing a comprehensive and robust RTL design for the AHB2APB bridge. While the design we selected adhered to many of the AHB-to-APB bridge specifications and implemented key functionalities, it lacked support for burst transactions due to the absence of a burst signal. This limitation restricted our ability to thoroughly test advanced features such as burst and wrap transactions, which are integral to certain use cases.

Looking forward, our future work will focus on addressing these limitations. This includes exploring more advanced RTL designs that include burst signal support to enable comprehensive testing of burst and wrap transaction scenarios. If suitable RTL designs are unavailable, we plan to develop and implement an enhanced version of the bridge design ourselves, ensuring it adheres to all required specifications, including those related to burst and wrap transfers.

Additionally, expanding the scope of our verification efforts to cover more complex scenarios and stress-testing the bridge under varying workloads will further solidify its robustness. This iterative process will not only enhance the reliability of the current design but also contribute to the development of a more flexible and efficient verification framework for future projects.

CONCLUSION

The verification of the AHB-APB bridge design proved to be an enlightening and rewarding endeavor, offering profound insights into the practical application of Formal Verification techniques and system verilog Assertions (SVA) for formal property verification. This project emphasized the importance of systematic approaches in ensuring design integrity and compliance with stringent industry standards such as the ARM AMBA specifications.

Throughout the project, our team encountered and overcame several challenges. Chief among these was the task of identifying a suitable RTL code that fulfilled our verification requirements while adhering to the specifications of the AHB-APB bridge. The process of developing precise assertions, assumptions, and cover properties demanded a detailed understanding of the bridge's functionality, enabling us to rigorously evaluate its behavior across a variety of scenarios. Despite the complexity of this task, our well-structured approach and collaborative efforts allowed us to efficiently address potential issues and ensure comprehensive coverage.

This project also underscored the importance of teamwork and organization in achieving verification goals. By maintaining a clear and well-organized file structure and leveraging open communication, we streamlined the development and verification processes. These practices not only fostered a productive workflow but also ensured that team members could collaborate effectively and respond to challenges with agility.

The verification process has equipped us with a deeper understanding of the nuances and complexities associated with formal verification methodologies. It has honed our ability to craft and execute verification plans, identify design flaws, and validate compliance with industry standards. Furthermore, the experience reinforced the critical role of attention to detail in identifying subtle design flaws that can significantly impact functionality.

REFERENCES

RTL Source:https://github.com/prajwalgekkouga/AHB-to-APB-Bridge.
ChatGPT
Professor Slides