Formal Verification of AHB-to-APB Bridge

Yelin Mao, Yuxiang Xia, Hongkuan Yu
UNI: ym3000, yx2821, hy2819

Fall 2024

Contents

1 Overview **2**

2 Short Introduction of the Advanced Microcontroller Bus Architecture **3**

3 Source Code of AHB-to-APB Bridge **5**
3.1 AHB-to-APB Bridge Overview 5
3.2 Source RTL Code Analysis 5

4 File Hierarchy **7**

5 Formal Verification Processes **8**
5.1 Testbench for AHB_Interface.v 14
5.2 FPV for APB_FSM.v 18
5.3 FPV for top.v 24
5.4 Testbench for top.v 35

6 Results **38**
6.1 Testbench Results of AHB_Interface.v 38
6.2 FPV Results of APB_FSM.v 39
6.3 FPV Results of top.v 41
6.4 Testbench Results of top.v 48
6.5 FPV Results v.s. Testbench Results of top.v 49

7 Conclusions and Future Improvements **50**

1 Overview

In our final project, we focus on the verification of a critical component within the AMBA protocol family: the **AHB-to-APB bridge**. The Advanced Microcontroller Bus Architecture (AMBA) is a widely adopted standard for on-chip communication in System-on-Chip (SoC) designs. Among its key components, the AHB-to-APB bridge plays a vital role in interfacing the high-speed Advanced High-performance Bus (AHB) with the low-power Advanced Peripheral Bus (APB). We will introduce AMBA in the **Short Introduction of the Advanced Microcontroller Bus Architecture** section.

To ensure the correct functionalities of this bridge, we employ hardware formal verification using SystemVerilog to specify the desired properties and behaviors of the design. The Cadence Jasper Gold formal verification tool will be the primary platform for exhaustively analyzing and proving these properties. Formal Property Verification (FPV) allows us to detect potential issues at an early stage, ensuring a robust and functionally correct AHB-to-APB bridge that adheres to the AMBA specification document [1] at the RTL level.

2 Short Introduction of the Advanced Microcontroller Bus Architecture

The **Advanced Microcontroller Bus Architecture**, abbreviated as AMBA, is an open-standard, on-chip interconnect specification developed by Arm Ltd. Introduced in 1996, AMBA facilitates the connection and management of functional blocks within system-on-a-chip (SoC) designs, promoting efficient communication between components such as CPUs, GPUs, and signal processors.

Over the years, AMBA has evolved to include several bus protocols, each tailored to specific performance and power requirements [2]:

- Advanced System Bus (ASB): One of the initial buses introduced with AMBA, ASB is designed for high-performance modules.
- Advanced Peripheral Bus (APB): Also part of the initial AMBA release, APB is optimized for low-power peripherals and is used for connecting simple, low-bandwidth devices.
- Advanced High-performance Bus (AHB): Introduced in AMBA 2 (1999), AHB is a single clock-edge protocol designed for high-performance and high clock frequency system modules.

In our project, we primarily work with the AHB interface and the APB interface, as well as the bridge facilitating communication between them. Figure 1 below visually represents the structure of the design.

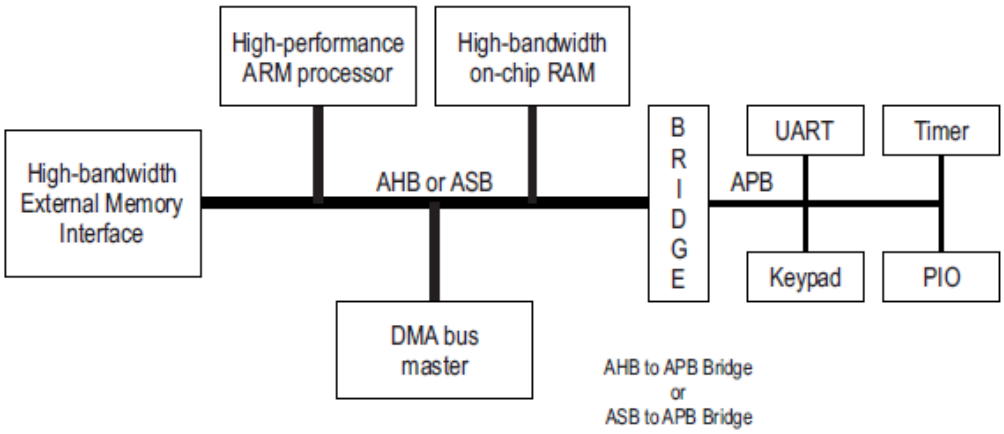


Figure 1: AHB and APB with Communication Bridge

3 Source Code of AHB-to-APB Bridge

3.1 AHB-to-APB Bridge Overview

The AHB-to-APB bridge serves as an essential interface connecting the AHB with APB. It is designed to facilitate various read-and-write operations between these two buses. Key features of the bridge include an AHB slave bus interface, an APB transfer state machine that functions independently of the device's memory map, and mechanisms for generating APB output signals.

The primary role of the AHB-to-APB bridge is to store addresses, control signals, and data from the AHB and then transmit them to the APB peripherals while also relaying data and response signals back to the AHB. It manages the APB data bus via two distinct channels: the read data path (Prdata) and the write data path (Pwdata). Furthermore, the bridge supports both sequential and non-sequential data transfers of varying data sizes (Hsize), providing a flexible and efficient communication interface between high-speed and low-speed buses.

3.2 Source RTL Code Analysis

The RTL code we utilized is available here [3]. Its structure is outlined as follows:

```
AHB_Master.v
AHB_Slave_Interface.v
APB_Controller.v
APB_Interface.v
bridge_top.v
```

The core components of the AHB-to-APB Bridge, which we focus on for formal verification, are:

```
AHB_Slave_Interface.v
APB_Controller.v
bridge_top.v
```

We renamed them according to our preferences as follows:

```
AHB_Interface.v
APB_FSM.v
top.v
```

The core code of the AHB-to-APB Bridge does not independently manage the HSIZE and HBURST functions. Instead, while the bridge can handle data of varying sizes and supports burst read and write operations, these functionalities are controlled by the AHB master, with the bridge simply receiving the inputs from the AHB master. Additionally, the core code only supports a maximum of 32-bit data (defined as a word in the AMBA specification document [1]), so for simplicity, we feed the bridge's data input with the size of 32 bits. The mechanism of the bridge is depicted in Figure 2 below.

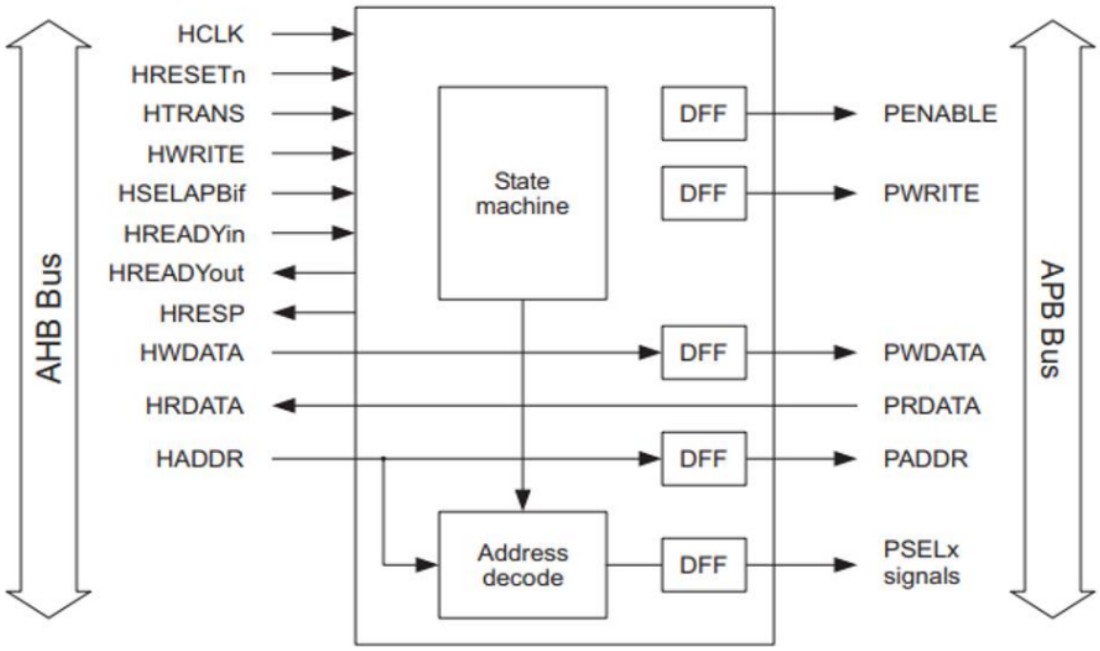


Figure 2: Mechanism of the AHB-to-APB Bridge

4 File Hierarchy

In our file hierarchy, the files are organized into distinct folders: **RTL**, **SVA**, **TB**, and **Specification_Sheets**. The **SVA** folder contains several `.tcl` files, along with a `run.sh` script, to execute Cadence Jasper Gold. For testbench simulation, the **TB** folder includes the testbench code, along with `waveform.do`, `runsim.do`, and `run.sh` files, to execute the simulation in Siemens ModelSim.

5 Formal Verification Processes

This project employs a model-checking approach to verify that the AHB-to-APB bridge adheres to the AMBA specification document [1]. The verification strategy contains simulation-based test benches for the `AHB_Interface.v` and the `top.v` bridge, along with FPV for both the `AHB_FSM.v` and the `top.v` bridge.

We verify five functions of the bridge: Single Read (in Figure 3), Burst Read (in Figure 4), Single Write (in Figure 5), Burst Write (in Figure 6), and Back-to-Back Transfers (in Figure 7). The expected waveforms for each function are presented in the following figures.

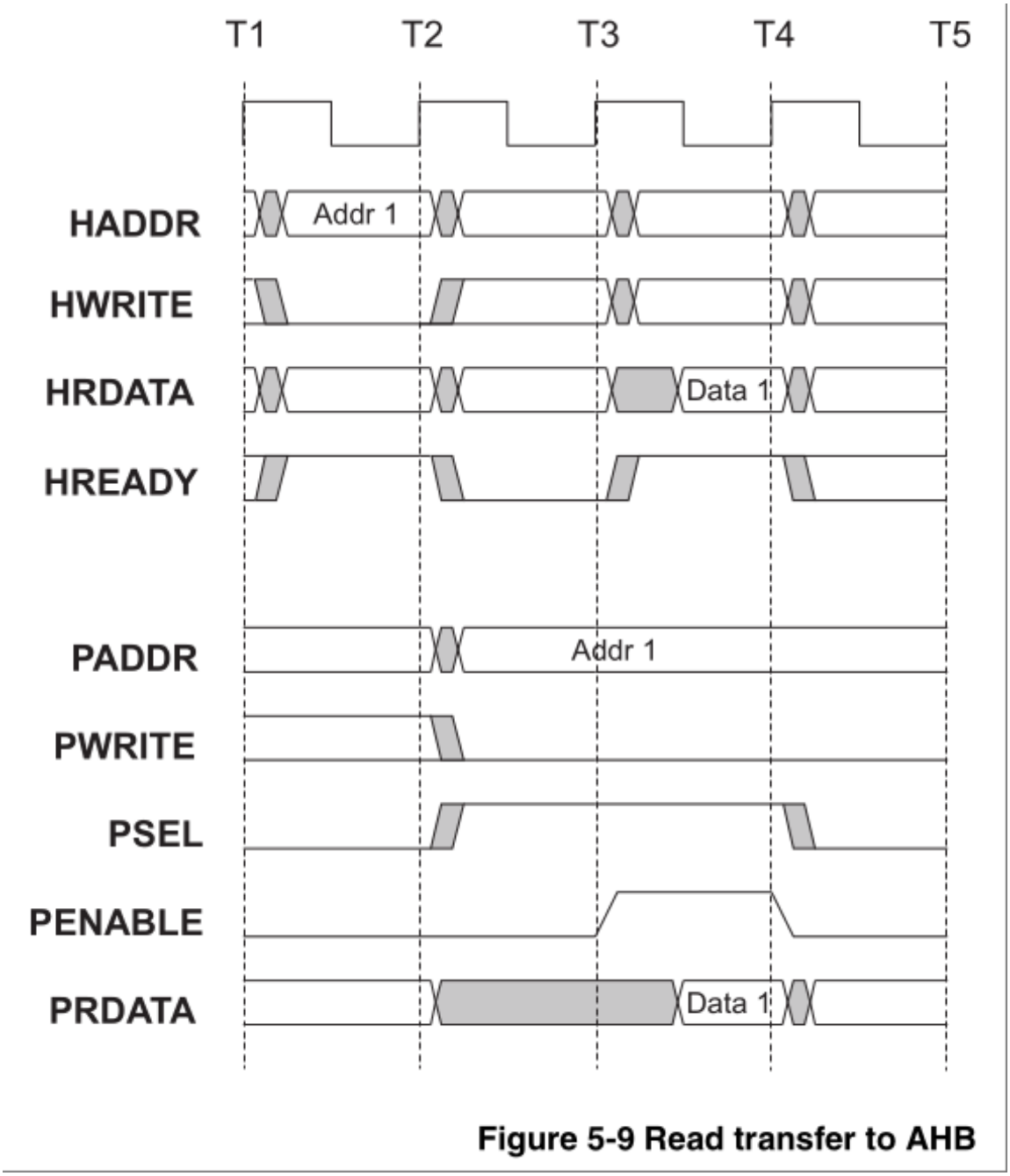


Figure 3: Single Read of the AHB-to-APB Bridge

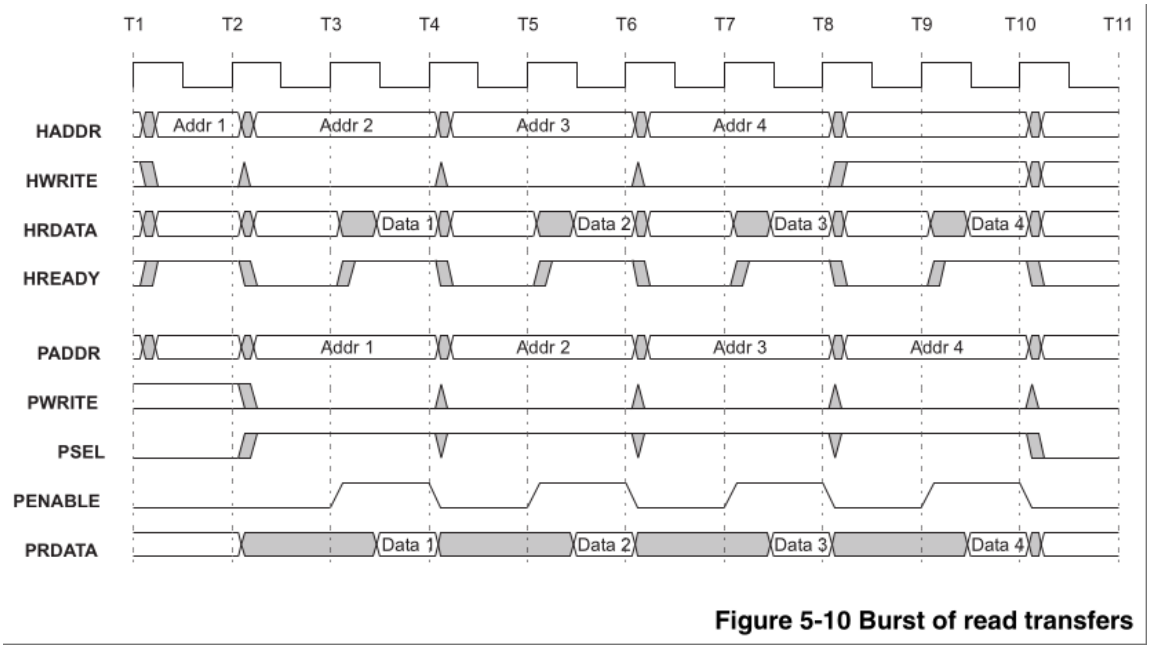


Figure 4: Burst Read of the AHB-to-APB Bridge

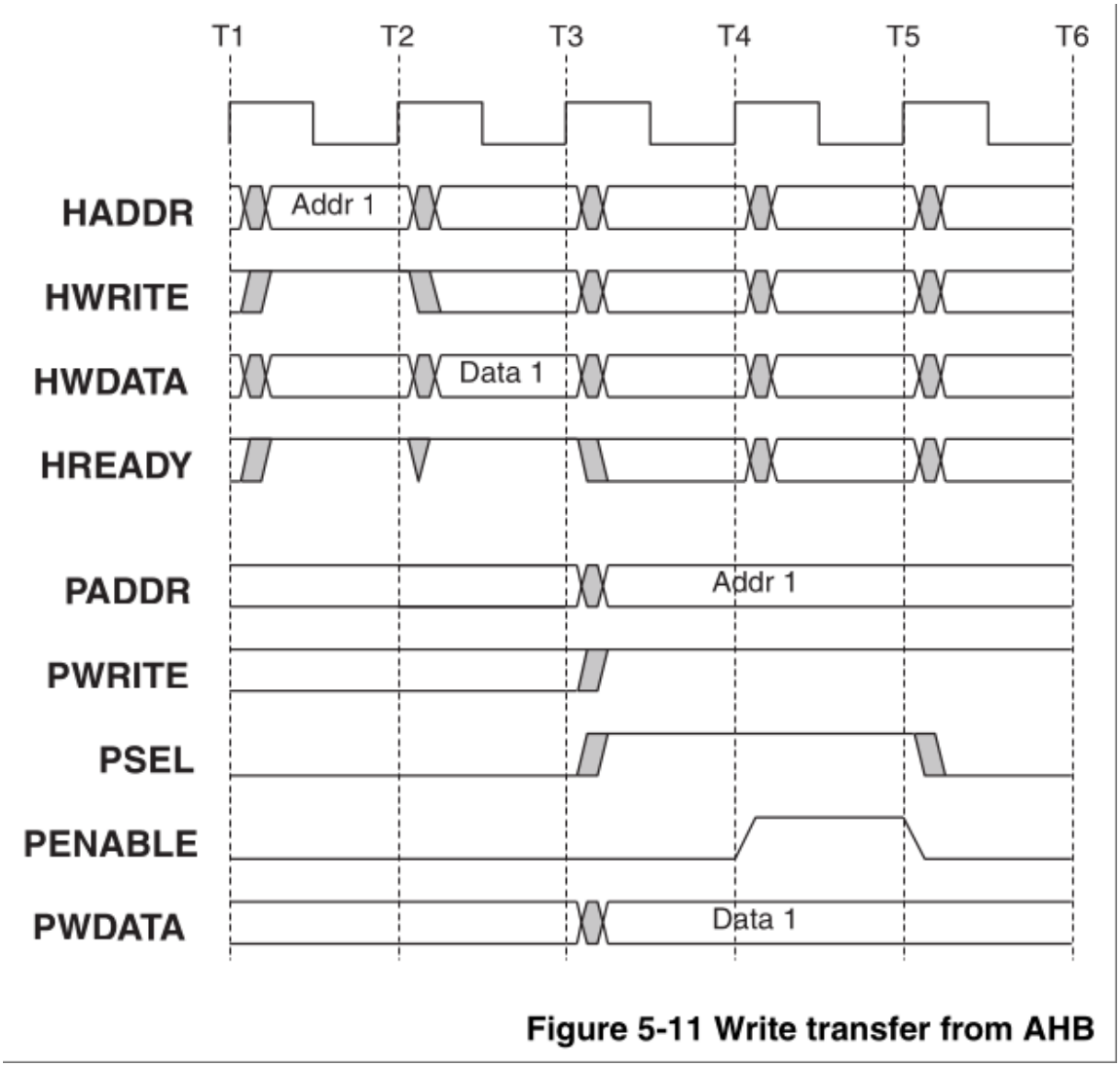


Figure 5: Single Write of the AHB-to-APB Bridge

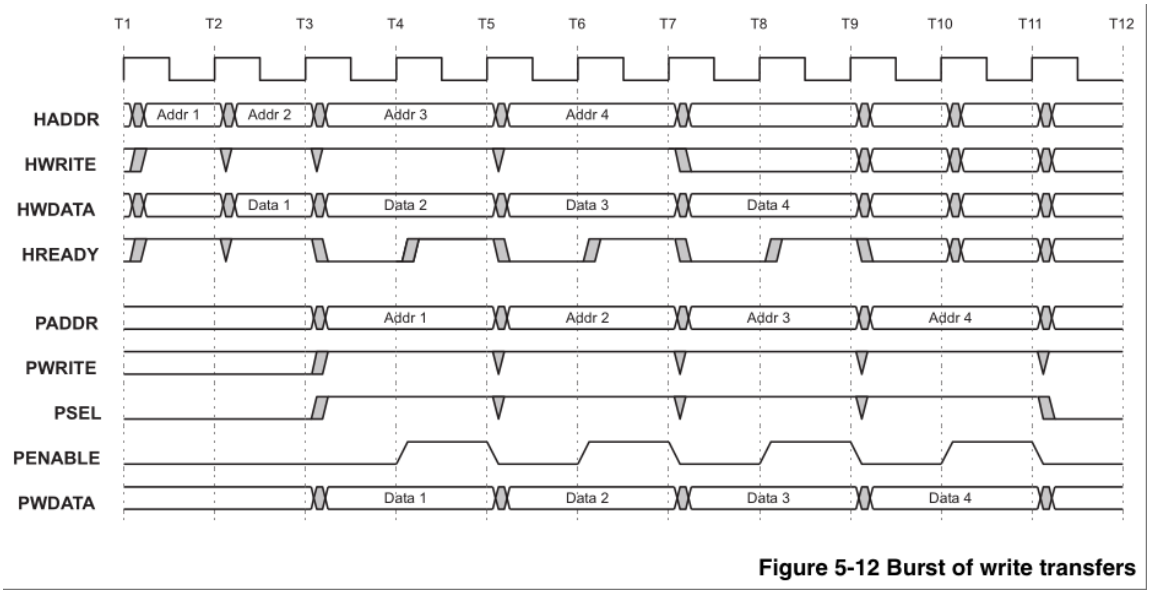


Figure 6: Burst Write of the AHB-to-APB Bridge

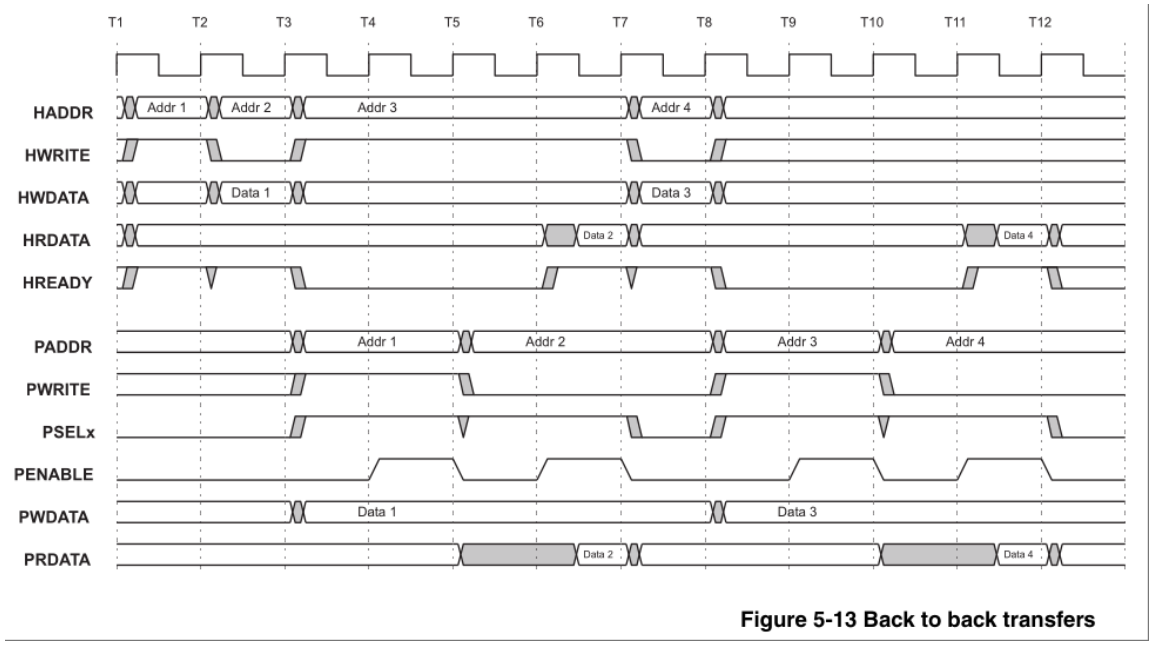


Figure 7: Back-to-Back Transfers of the AHB-to-APB Bridge

5.1 Testbench for AHB_Interface.v

The simulation testbench `AHB_Slave_Interface_tb.v` for `AHB_Interface.v` is shown below.

```
`timescale 1ns / 1ps

module AHB_slave_interface_tb;

// DUT signals
reg clk, rst;
reg Hwrite, Hreadyin;
reg [1:0] Htrans;
reg [31:0] Haddr, Hwdata, Prdata;
wire valid;
wire [31:0] Haddr1, Haddr2, Hwdata1, Hwdata2;
wire [31:0] Hrdata;
wire Hwritereg;
wire [2:0] tempselx;
wire [1:0] Hresp;

// Instantiate the DUT
AHB_slave_interface DUT (
    .clk(clk),
    .rst(rst),
    .Hwrite(Hwrite),
    .Hreadyin(Hreadyin),
    .Htrans(Htrans),
    .Haddr(Haddr),
    .Hwdata(Hwdata),
    .Prdata(Prdata),
    .valid(valid),
    .Haddr1(Haddr1),
    .Haddr2(Haddr2),
    .Hwdata1(Hwdata1),
    .Hwdata2(Hwdata2),
    .Hrdata(Hrdata),
    .Hwritereg(Hwritereg),
    .tempselx(tempselx),
```

```
    .Hresp(Hresp)
);

// Clock Generation
always #5 clk = ~clk; // 10ns clock period

// Testbench Procedure
initial begin
    // Step 1: Initialize
    clk = 0;
        rst = 0;
        Hwrite = 0;
        Hreadyin = 0;
        Htrans = 2'b00;
        Haddr = 32'b0;
        Hwdata = 32'b0;
        Prdata = 32'b0;
    #15 rst = 1; // Release reset after 15ns

    // Step 2: Read-Write-Read-Write Test
    $display("Starting Read-Write-Read-Write Test");

    // Write Transaction 1
    Haddr = 32'h8000_0010;
    Hwdata = 32'h1234_5678;
    Htrans = 2'b10; // Non-sequential
    Hwrite = 1;
    Hreadyin = 1;
    #10;

    // Read Transaction 1
    Hwrite = 0;
    Prdata = 32'hABCD_EF01; // Simulate data from APB
    #10;

    // Write Transaction 2
    Haddr = 32'h8400_0020;
    Hwdata = 32'h8765_4321;
```

```
Hwrite = 1;
Htrans = 2'b11; // Sequential
#10;

// Read Transaction 2
Hwrite = 0;
Prdata = 32'h1122_3344; // Simulate data from APB
#10;

$display("Read-Write-Read-Write Test Completed");

// Step 3: Invalid Address Test
$display("Starting Invalid Address Test");
Haddr = 32'h9000_0000; // Out of range
Htrans = 2'b10; // Non-sequential
Hreadyin = 1;
#10;
if (!valid)
    $display("Invalid Address Test Passed");
else
    $display("Invalid Address Test Failed");

// Step 4: Reset Test
$display("Starting Reset Test");
rst = 0;
#10;
if (Haddr1 == 0 && Haddr2 == 0 && Hwdata1 == 0 && Hwdata2 ==
↪  0)
    $display("Reset Test Passed");
else
    $display("Reset Test Failed");
rst = 1;

// Step 5: Burst Test (Incremental and Wrapping)
$display("Starting Burst Test");
Haddr = 32'h8000_0000;
Hwrite = 1;
Htrans = 2'b10; // Non-sequential
```

```
        Hreadyin = 1;
        #10;

        // Incremental burst
        Haddr = 32'h8000_0004;
        #10;
        Haddr = 32'h8000_0008;
        #10;

        // Wrapping burst
        Haddr = 32'h8000_000C; // Wrap to lower boundary
        #10;
        Haddr = 32'h8000_0000;
        #10;

        $display("Burst Test Completed");

        // Finish simulation
        $stop;
    end

    endmodule
```

The AHB slave interface simply operates as an I/O module, making the simulation testbench sufficient for verifying its input and output behavior. The primary verification goals include validating data transfers, ensuring correct address handling, and detecting invalid inputs.

The testbench is designed to test the functionality of the AHB slave interface under a variety of scenarios, including normal operations, edge cases, and invalid conditions. Specific test cases focus on evaluating the correct toggling of signals during read-write sequences and verifying the interface's ability to handle invalid addresses. Also, the testbench ensures the proper functionality of the active-low reset mechanism. The results are explained in the **Testbench Results of AHB_Interface.v** section below.

5.2 FPV for APB_FSM.v

Figure 8 below shows the FSM state transitions of the APB controller within the AHB-to-APB bridge.

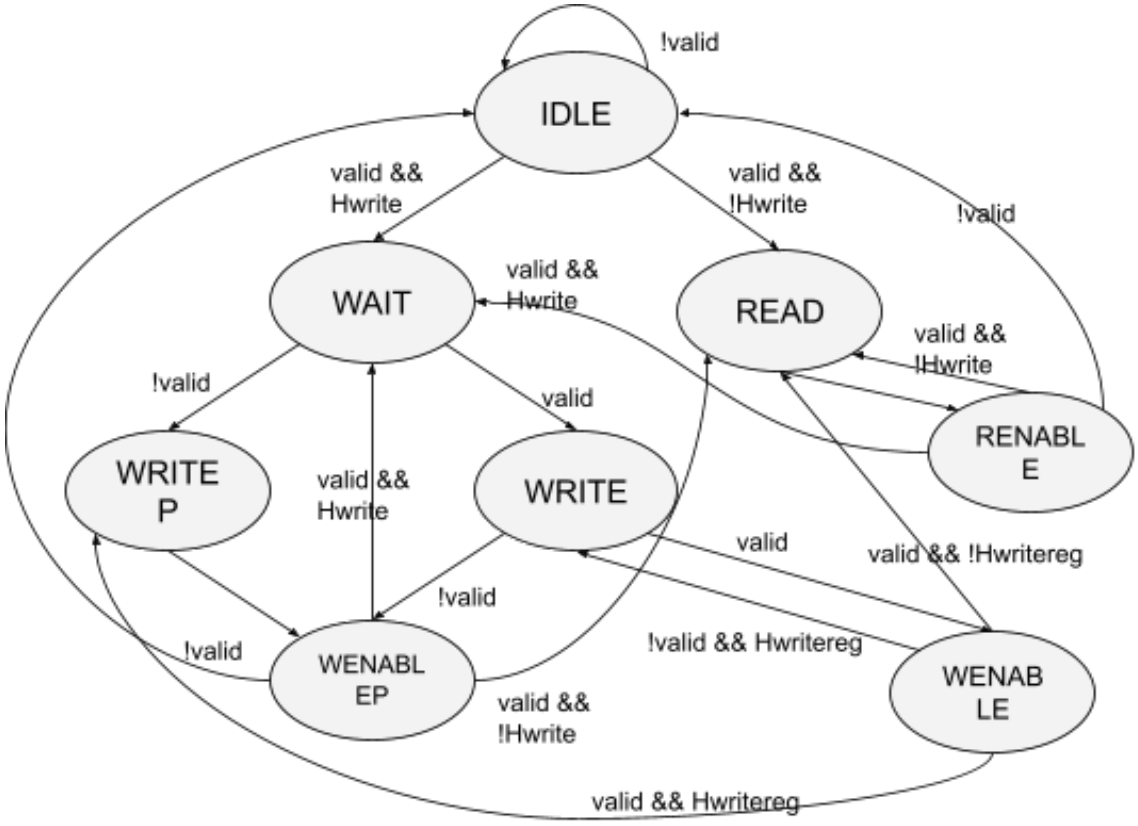


Figure 8: FSM State Transitions of the APB Controller

The FPV code `APB_FSM_sva.sv` using SystemVerilog Assertion (SVA) for `APB_FSM.v` is shown below.

```
module APB_FSM_sva(
    input wire clk,
    input wire rst,
    input wire valid,
    input wire Hwrite,
    input wire [31:0] Hwdata,
```

```
    input wire [31:0] Haddr,
    input wire [31:0] Haddr1,
    input wire [31:0] Haddr2,
    input wire [31:0] Hwdata1,
    input wire [31:0] Hwdata2,
    input wire [31:0] Prdata,
    input wire [2:0] tempselx,
    input wire Hwritereg,
    input wire Pwrite,
    input wire Penable,
    input wire [2:0] Pselx,
    input wire [31:0] Paddr,
    input wire [31:0] Pwdata,
    input wire Hreadyout,
    input wire [2:0] CS,
    input wire [2:0] NS

);
//PARAMETERS
        parameter IDLE=3'b000;
        parameter WAIT=3'b001;
        parameter READ= 3'b010;
        parameter WRITE=3'b011;
        parameter WRITEP=3'b100;
        parameter RENABLE=3'b101;
        parameter WENABLE=3'b110;
        parameter WENABLEP=3'b111;

//assertions
//verify basic state transfer
property FSM_state_transfer;
        @(posedge clk) disable iff(!rst) 1 |-> ##2 CS ==
        ↪  $past(NS);
endproperty
assert_fsm_state_transfer: assert property (FSM_state_transfer);

//check each state transfer
//IDLE
```

```
property IDLE_TO_IDLE;
        @(posedge clk) disable iff(!rst)
        CS == IDLE && !valid |-> NS == IDLE;
endproperty
assert_idle2idle: assert property (IDLE_TO_IDLE);

property IDLE_TO_WAIT;
        @(posedge clk) disable iff(!rst)
        CS == IDLE && valid && Hwrite |-> NS == WAIT;
endproperty
assert_idle2wait: assert property (IDLE_TO_WAIT);

property IDLE_TO_READ;
        @(posedge clk) disable iff(!rst)
        CS == IDLE && valid && !Hwrite |-> NS == READ;
endproperty
assert_idle2read: assert property (IDLE_TO_READ);

//WAIT
property WAIT_TO_WRITE;
        @(posedge clk) disable iff(!rst)
        CS == WAIT && !valid |-> NS == WRITE;
endproperty
assert_wait2write: assert property (WAIT_TO_WRITE);

property WAIT_TO_WRITEP;
        @(posedge clk) disable iff(!rst)
        CS == WAIT && valid |-> NS == WRITEP;
endproperty
assert_wait2writep: assert property (WAIT_TO_WRITEP);

//READ
property READ_TO_RENABLE;
        @(posedge clk) disable iff(!rst)
        CS == READ |-> NS == RENABLE;
endproperty
assert_read2renable: assert property (READ_TO_RENABLE);
```

```
//WRITE
property WRITE_TO_WENABLE;
        @(posedge clk) disable iff(!rst)
        CS == WRITE && !valid |-> NS == WENABLE;
endproperty
assert_write2wenable: assert property (WRITE_TO_WENABLE);

property WRITE_TO_WENABLEP;
        @(posedge clk) disable iff(!rst)
        CS == WRITE && valid |-> NS == WENABLEP;
endproperty
assert_write2wenablep: assert property (WRITE_TO_WENABLEP);

//WRITEP
property WRITEP_TO_WENABLEP;
        @(posedge clk) disable iff(!rst)
        CS == WRITEP |-> NS == WENABLEP;
endproperty
assert_writep2wenablep: assert property (WRITEP_TO_WENABLEP);

//RENABLE
property RENABLE_TO_IDLE;
        @(posedge clk) disable iff(!rst)
        CS == RENABLE && !valid |-> NS == IDLE;
endproperty
assert_renable2idle: assert property (RENABLE_TO_IDLE);

property RENABLE_TO_WAIT;
        @(posedge clk) disable iff(!rst)
        CS == RENABLE && valid && Hwrite |-> NS == WAIT;
endproperty
assert_renable2wait: assert property (RENABLE_TO_WAIT);

property RENABLE_TO_READ;
        @(posedge clk) disable iff(!rst)
        CS == RENABLE && valid && !Hwrite |-> NS == READ;
endproperty
assert_renable2read: assert property (RENABLE_TO_READ);
```

```
//WENABLE
property WENABLE_TO_IDLE;
        @(posedge clk) disable iff(!rst)
        CS == WENABLE && !valid |-> NS == IDLE;
endproperty
assert_wenable2idle: assert property (WENABLE_TO_IDLE);

property WENABLE_TO_WAIT;
        @(posedge clk) disable iff(!rst)
        CS == WENABLE && valid && Hwrite |-> NS == WAIT;
endproperty
assert_wenable2wait: assert property (WENABLE_TO_WAIT);

property WENABLE_TO_READ;
        @(posedge clk) disable iff(!rst)
        CS == WENABLE && valid && !Hwrite |-> NS == READ;
endproperty
assert_wenable2read: assert property (WENABLE_TO_READ);

//WENABLEP
property WENABLEP_TO_WRITE;
        @(posedge clk) disable iff(!rst)
        CS == WENABLEP && !valid && Hwritereg |-> NS == WRITE;
endproperty
assert_wenablep2write: assert property (WENABLEP_TO_WRITE);

property WENABLEP_TO_WRITEP;
        @(posedge clk) disable iff(!rst)
        CS == WENABLEP && valid && Hwritereg |-> NS == WRITEP;
endproperty
assert_wenablep2writep: assert property (WENABLEP_TO_WRITEP);

property WENABLEP_TO_READ;
        @(posedge clk) disable iff(!rst)
        CS == WENABLEP && valid && !Hwritereg |-> NS == READ;
endproperty
assert_wenablep2read: assert property (WENABLEP_TO_READ);
```

```
endmodule

bind APB_FSM APB_FSM_sva APB_FSM_chk(.*);
```

SVAs are employed to verify the correctness of state transitions. By using each `if` and `else` condition provided in the source code [3] as assertion criteria, we ensure that the current state `CS` and next state `NS` transition as intended according to the logic defined in the source code. It also ensures that every valid state transition, such as from `IDLE` to `READ` or from `WAIT` to `WRITE`, occurs only under appropriate conditions, preventing any illegal transitions or undefined behaviors. The result can be found in the **FPV Results of APB_FSM.v** section below.

5.3 FPV for top.v

The FPV code `top_sva.sv` using SystemVerilog Assertion and Assumption for `top.v` is shown below.

```
module top_sva(input logic clk,
                  input logic rst,
                  input logic Hwrite,
                  input logic Hreadyin,
                  input logic [31:0] Hwdata,
                  input logic [31:0] Haddr,
                  input logic [1:0] Htrans,
                  input logic [31:0] Prdata,
                  input logic [2:0] Pselx,
                  input logic [31:0] Paddr,
                  input logic [31:0] Pwdata,
                  input logic Penable,
                  input logic Pwrite,
                  input logic Hreadyout,
                  input logic [1:0] Hresp,
                  input logic [31:0] Hrdata);

let addr0 = Haddr>=32'h8000_0000 && Haddr<32'h8400_0000;
let addr1 = Haddr>=32'h8400_0000 && Haddr<32'h8800_0000;
let addr2 = Haddr>=32'h8800_0000 && Haddr<32'h8C00_0000;

let possible_Pselx = ((Pselx == 0) || (Pselx == 1) || (Pselx ==
↪  2) || (Pselx == 4));

/**************input addr and mode constraint*************/
assume property (@(posedge clk) Haddr>=32'h8000_0000 &&
↪  Haddr<32'h8C00_0000);
assume property (@(posedge clk) Htrans == 2'b1x);

/*************back-to-back constraint*************/
property read_follows_write1;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> ##1 !Hreadyin ##1 !Hreadyin ##1
        ↪  !Hreadyin;
```

```
endproperty
assume property (read_follows_write1);

property read_follows_write2;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> $past(!Hreadyin,2) &&
         ↪  $past(!Hreadyin,3) && $past(!Hreadyin,4);
endproperty
assume property (read_follows_write2);
//three invalid cycles before and after write-read pattern

property write_follows_read1;
        @(posedge clk) disable iff(!rst)
        read_h ##1 Hwrite |-> ($past(Hwrite && Hreadyin,2));
endproperty
assume property (write_follows_read1);
//must be write-read-write

property write_follows_read2;
        @(posedge clk) disable iff(!rst)
        read_h ##1 Hwrite |-> (!Hreadyin ##1 Hwrite && !Hreadyin
         ↪  ##1 Hwrite && !Hreadyin ##1 write_h);
endproperty
assume property (write_follows_read2);
//three invalid cycles before and after write-read pattern

/***************burst constraint***************/
//burst read
property burst_read_pre;
        @(posedge clk) disable iff(!rst)
        read_h ##1 !Hwrite ##1 read_h|-> (!$past(Hwrite,3) &&
         ↪  !$past(Hwrite,4) && !$past(Hwrite,5)) ||
         ↪  ($past(!Hreadyin,3) && $past(!Hreadyin,4) &&
         ↪  $past(!Hreadyin,5));
endproperty
assume property (burst_read_pre);

property burst_read_post;
```

```
        @(posedge clk) disable iff(!rst)
        (read_h ##1 !Hwrite ##1 read_h) |-> $past(!Hreadyin,3) &&
         ↪  $past(!Hreadyin,1) ##1 !Hreadyin ##1 Hreadyin;
endproperty
assume property (burst_read_post);
//Hreadyin pattern for burst read
//make sure that there are no valid write in 3 cycles before
 ↪  burst read starts

//burst write
property burst_write_pre;
        @(posedge clk) disable iff(!rst)
        Hwrite ##1 Hwrite ##1 Hwrite|-> ($past(Hwrite,3) &&
         ↪  $past(Hwrite,4)) || ($past(!Hreadyin,3) &&
         ↪  $past(!Hreadyin,4));
endproperty
assume property (burst_write_pre);
//make sure that there are no valid read in 2 cycles before
 ↪  burst write starts

property burst_write_start;
        @(posedge clk) disable iff(!rst)
         write_h ##1 write_h |-> ##1 Hwrite && !Hreadyin ##1
          ↪  Hreadyin;
endproperty
assume property (burst_write_start);
//start pattern of burst write

property burst_write_body;
        @(posedge clk) disable iff(!rst)
        write_h ##1 Hwrite && !Hreadyin ##1 write_h |->
         ↪  ($past(Hwrite && Hreadyin,3)) || ($past(Hwrite,3) &&
         ↪  $past(Hwrite,4));
endproperty
assume property (burst_write_body);
//when burst write body pattern occurs, it follows either start
 ↪  pattern or body pattern
```

```
property burst_write_post;
        @(posedge clk) disable iff(!rst)
        //write_h ##1 Hwrite && !Hreadyin ##1 write_h |-> ##1
         ↪  !Hreadyin ##1 Hreadyin;
        write_h ##1 Hwrite ##1 write_h |-> $past(!Hreadyin,1) ##1
         ↪  !Hreadyin ##1 Hreadyin;
endproperty
assume property (burst_write_post);

property burst_write_Hwrite;
        @(posedge clk) disable iff(!rst)
         write_h ##1 Hwrite && !Hreadyin |-> ##1 Hwrite ;
endproperty
assume property (burst_write_Hwrite);

property burst_Hwdata1;
        @(posedge clk) disable iff(!rst)
        burst_write1 |-> ##2 (Hwdata == $past(Hwdata,1));
endproperty
assume property (burst_Hwdata1);

property burst_Hwdata2;
        @(posedge clk) disable iff(!rst)
        burst_write1 ##1 (burst_write2[*]) |-> ##2 (Hwdata ==
         ↪  $past(Hwdata,1));
endproperty
assume property (burst_Hwdata2);

//reset check
property reset_check_Paddr;
    @(posedge clk)
        $rose(rst) |-> Paddr == 0;
endproperty

property reset_check_Penable;
    @(posedge clk)
        $rose(rst) |-> Penable == 0;
endproperty
```

```
//APB output waveform discription
sequence write_p;
        Pwrite &&  possible_Pselx  && !Penable ##1 Pwrite &&
        ↪  possible_Pselx && Penable;
endsequence

sequence read_p;
        !Pwrite && possible_Pselx && !Penable ##1 !Pwrite &&
        ↪  possible_Pselx && Penable;
endsequence

//AHB valid input waveform discription
sequence write_h;
        Hwrite && Hreadyin;
endsequence

sequence read_h;
        !Hwrite && Hreadyin;
endsequence

//single read/write check---------//use testbench to test single
↪  read and write case

/*************burst write check*************/

property read_data_transfer;
        @(posedge clk) disable iff(!rst)
        !Pwrite && Penable |-> Hrdata == Prdata;
endproperty

sequence burst_read;
        read_h ##1 !Hwrite && !Hreadyin ##1 read_h;
endsequence

property read_Pwrite;
        @(posedge clk) disable iff(!rst)
        burst_read |-> $past(!Pwrite);
```

```
endproperty

property read_Penable;
        @(posedge clk) disable iff(!rst)
        burst_read |-> Penable ##1 !Penable ##1 Penable ##1
         ↪  !Penable;
endproperty

property burst_read_addr;
        @(posedge clk) disable iff(!rst)
        //addr0 ##0 burst_read |-> $past(Pselx[0],1) && Pselx[0]
         ↪  && $onehot(Pselx);
        burst_read |-> Paddr == $past(Haddr,2) && $past(Paddr,1)
         ↪  == $past(Haddr,2);
endproperty

/*************burst write check*************/

property write_Pwrite;
        @(posedge clk) disable iff(!rst)
        write_h ##1 Hwrite |-> ##1 Pwrite;
endproperty

sequence burst_write1;
        write_h ##1 write_h ;
endsequence

sequence burst_write2;
        Hwrite && !Hreadyin ##1 write_h ;
endsequence

property burst_write_data1;
        @(posedge clk) disable iff(!rst)
        burst_write1 |-> ##1 (Pwdata == $past(Hwdata,1)) ##1
         ↪  (Pwdata == $past(Hwdata,2))  ##1 (Pwdata ==
         ↪  $past(Hwdata,2)) ##1 (Pwdata == $past(Hwdata,2));
endproperty
```

```
property burst_write_data2;
        @(posedge clk) disable iff(!rst)
        burst_write1 ##1 (burst_write2[*]) |-> ##3 (Pwdata ==
        ↪  $past(Hwdata,2)) ##1 (Pwdata == $past(Hwdata,2)) &&
        ↪  Pwdata == $past(Pwdata,1);
endproperty

property write_Penable1;
        @(posedge clk) disable iff(!rst)
        burst_write1 |-> ##2 Penable ##1 !Penable ##1 Penable;
endproperty

property write_Penable2;
        @(posedge clk) disable iff(!rst)
        burst_write1 ##1 (burst_write2[*]) |-> ##1 !Penable ##1
        ↪  Penable ##1 !Penable;
        //write_h ##1 Hwrite && !Hreadyin ##1 write_h |-> ##4
        ↪  Penable ##1 !Penable ##1 Penable;
endproperty

property burst_write_addr1;
        @(posedge clk) disable iff(!rst)
        burst_write1 |-> ##1 Paddr == $past(Haddr,2) ##1 Paddr ==
        ↪  $past(Haddr,3);
endproperty

property burst_write_addr2;
        @(posedge clk) disable iff(!rst)
        burst_write1 ##1 burst_write2 |-> ##1 Paddr ==
        ↪  $past(Haddr,3) ##1 Paddr == $past(Paddr,1);
endproperty


/*************back-to-back check*************/

property back_to_back_Paddr;
        @(posedge clk) disable iff(!rst)
```

```
        write_p ##1 read_p |-> ##1  $past(Paddr,2) ==
        ↪  $past(Paddr);
endproperty

//Penable check
property back_to_back_Penable;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> ##2 Penable ##1 !Penable ##1
        ↪  Penable;
endproperty

//pselx check: pselx should hold for 2 cycles
//For Address range 8000_0000 to 8400_0000
sequence psel_s0;
        $onehot(Pselx) ##0 (Pselx[0] ##1 Pselx[0]);
endsequence

//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

//check read addr
property read_after_write_addr;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> ##3 (Paddr == $past(Haddr,3)) ##1
        ↪  (Paddr == $past(Paddr,1));
endproperty

//check write addr
property write_after_read_addr;
        @(posedge clk) disable iff(!rst)
```

```
        read_h ##1 !Hreadyin ##1 !Hreadyin ##1 !Hreadyin ##1
        ↪  write_h |-> ##2 (Paddr == $past(Haddr,2)) ##1 (Paddr
        ↪  == $past(Paddr,1));
endproperty

//check data
property back2back_data;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> ##1 (Pwdata == $past(Hwdata));
endproperty

property back2back_Pwrite;
        @(posedge clk) disable iff(!rst)
        write_h ##1 read_h |-> ##1 Pwrite ##1 Pwrite ##1 !Pwrite
        ↪  ##1 !Pwrite;
endproperty

//failed check -- psel
//burst write
property burst_write_psel0;
        @(posedge clk) disable iff(!rst)
        addr0 ##0 burst_write1 |-> ##1 psel_s0;
endproperty


//assertion check
Reset_check_Paddr: assert property (reset_check_Paddr);
Reset_check_Penable: assert property (reset_check_Penable);

Read_data_transfer: assert property (read_data_transfer);
Read_Pwrite: assert property (read_Pwrite);
Read_Penable: assert property (read_Penable);
Burst_read_addr: assert property (burst_read_addr);

Burst_write_data1: assert property (burst_write_data1);
Burst_write_data2: assert property (burst_write_data2);
Write_Pwrite: assert property (write_Pwrite);
Write_Penable1: assert property (write_Penable1);
```

```
Write_Penable2: assert property (write_Penable2);
Burst_write_addr1: assert property (burst_write_addr1);
Burst_write_addr2: assert property (burst_write_addr2);

Back_to_back_Penable: assert property (back_to_back_Penable);
Back_to_back_Paddr: assert property (back_to_back_Paddr);
Read_after_write_addr: assert property (read_after_write_addr);
Write_after_read_addr: assert property (write_after_read_addr);
Back2back_data: assert property (back2back_data);

Back2back_Pwrite: assert property (back2back_Pwrite);
Burst_write_psel0: assert property (burst_write_psel0);

endmodule


bind Bridge_Top top_sva chk_top (.clk(clk), .rst(rst),
 ↪  .Hwrite(Hwrite), .Hreadyin(Hreadyin), .Hwdata(Hwdata),
 ↪  .Haddr(Haddr), .Htrans(Htrans), .Prdata(Prdata),
 ↪  .Pselx(Pselx), .Paddr(Paddr), .Pwdata(Pwdata),
 ↪  .Penable(Penable), .Pwrite(Pwrite), .Hreadyout(Hreadyout),
 ↪  .Hresp(Hresp), .Hrdata(Hrdata));
```

We use assumptions to constrain inputs to valid conditions and assertions to verify the outputs and internal states of the AHB-to-APB bridge.

The top-level design (can refer to the source code [3]) integrates the AHB slave interface logic with the APB FSM. The primary goal is to formally verify end-to-end functionality under valid conditions, enforced by assumptions on address ranges and transfer types, while using assertions to confirm correct behavior. Key assumptions include ensuring that `Haddr` always falls within the range of `32'h8000_0000` to `32'h8C00_0000`; `Htrans` is restricted to `Non-sequential 2'b10` or `Sequential 2'b11`. The design does not support `IDLE 2'b00` or `BUSY 2'b01` transaction types in this environment, and invalid address ranges are considered out of scope for the bridge's operational region. When these assumptions are satisfied, most properties are expected to hold `TRUE`.

The remaining assumptions are applied to validate burst read, burst write, and back-to-back transfer operations. These assumptions ensure that the timing relationships between input signals are strictly constrained as the AMBA specification document [1]. The properties are defined to capture the expected output behavior for these three transfer scenarios, assuming correct input assumptions. Removing certain assumptions highlights how the design behaves under invalid or out-of-range inputs. All assumptions play a crucial role in maintaining the validity of the results, and any removal of these assumptions leads to assertion failures. The counterexample resulting from the removal of `Haddr` and `Htrans` assumptions, along with an analysis of their impact, are elaborated in the **FPV Results of top.v** section below.

5.4 Testbench for top.v

The simulation testbench `top_tb.v` for `top.v` is shown below.

```
`timescale 1ns / 1ps
module top_tb;

reg clk;
reg rst;
reg Hwrite;
reg Hreadyin;
reg [31:0] Hwdata,Haddr,Prdata;
reg [1:0] Htrans;

wire Penable,Pwrite,Hreadyout;
wire [1:0] Hresp;
wire [2:0] Pselx;
wire [31:0] Paddr,Pwdata;
wire [31:0] Hrdata;

Bridge_Top dut(
        .clk(clk),
    .rst(rst),
    .Hwrite(Hwrite),
    .Hreadyin(Hreadyin),
    .Htrans(Htrans),
    .Haddr(Haddr),
    .Hwdata(Hwdata),
    .Prdata(Prdata),
    .Hrdata(Hrdata),
        .Penable(Penable),
        .Pwrite(Pwrite),
        .Hreadyout(Hreadyout),
        .Hresp(Hresp),
        .Pselx(Pselx),
        .Paddr(Paddr),
        .Pwdata(Pwdata)
);
```

```
// Clock Generation
always #5 clk = ~clk; // 10ns clock period

initial begin
        clk = 1;
        rst = 0;
        Hwrite = 0;
        Hreadyin = 0;
        Haddr = 32'b0;
        Hwdata = 32'b0;
        Prdata = 32'b0;
        Htrans = 2'b10;

        #20 rst = 1;

        //single read
        Haddr = 32'h8400_0010;
        Hwrite = 0;
        Hreadyin = 1;
        #10
        Haddr = 32'bz;
        Hwrite = 1'bz;
        Hreadyin = 0;
        #10
        Hreadyin = 1;
        Prdata = 32'h1111_2222;
        #10
        Hreadyin = 0;
        #40

        //single write
        Haddr = 32'h8800_0010;
        Hwrite = 1;
        Hreadyin = 1;
        #10
        Haddr = 32'bz;
        Hwrite = 1'bz;
        Hreadyin = 1;
```

```
        Hwdata = 32'h3333_4444;
        #10
        Hreadyin = 0;
        #10
        Hreadyin = 0;
        #40
        $stop;
    end
    endmodule
```

A standalone simulation testbench is used to verify the entire AHB-to-APB bridge for single read and single write functions. These operations are straightforward and do not require FPV. The primary purpose of this testbench is to serve as a complement case for FPV by simulating simple functional tests. The result can be found in the **Testbench Results of top.v** section below.

6 Results

6.1 Testbench Results of AHB_Interface.v

Figure 9 and 10 below show the testbench results for `AHB_Interface.v`.

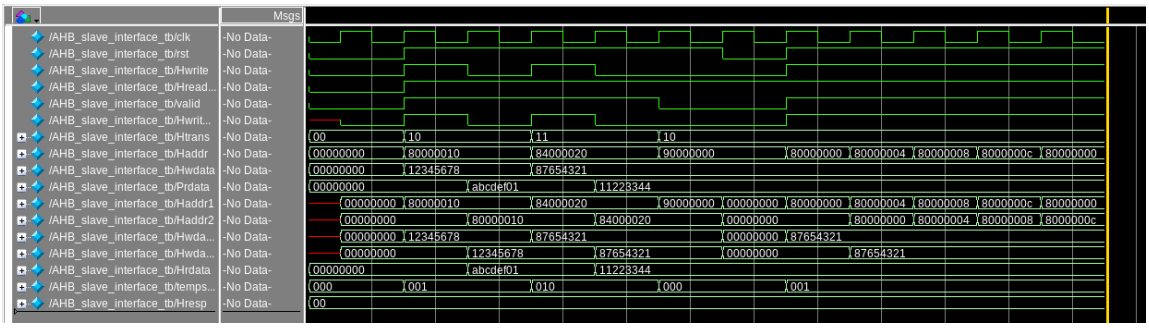


Figure 9: ModelSim Testbench Wave Results of `AHB_Slave_Interface_tb.v`

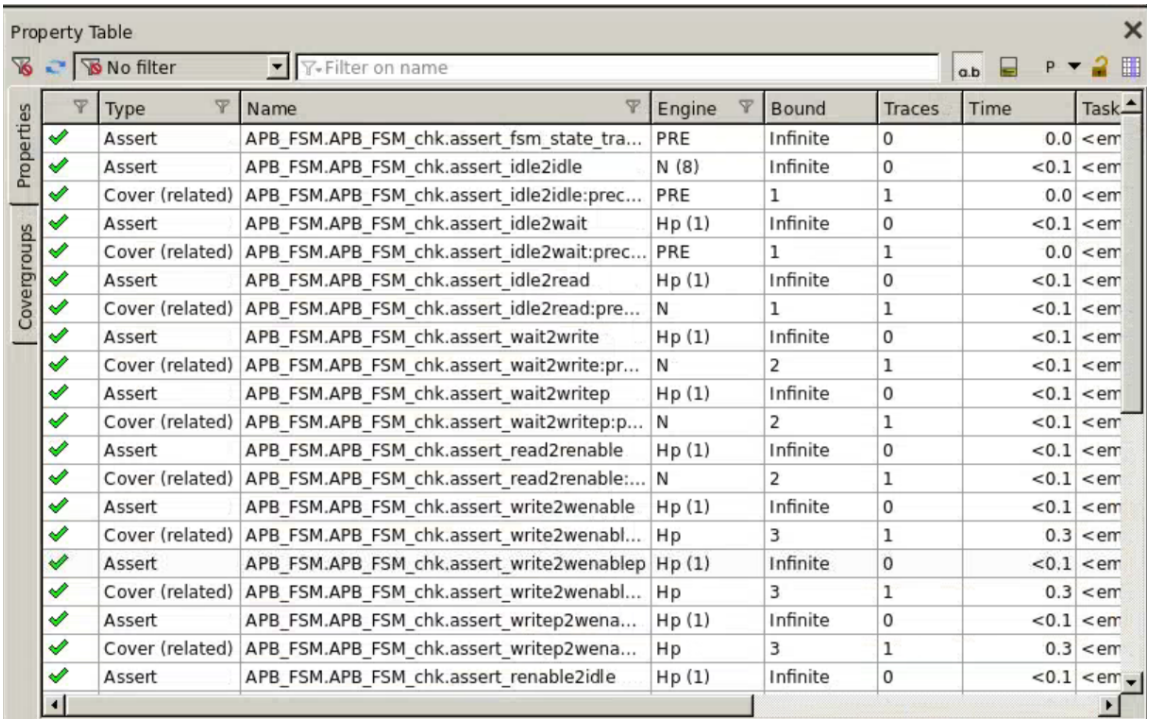
```
# Starting Read-Write-Read-Write Test
# Read-Write-Read-Write Test Completed
# Starting Invalid Address Test
# Invalid Address Test Passed
# Starting Reset Test
# Reset Test Passed
# Starting Burst Test
# Burst Test Completed
# ** Note: $stop    : AHB_Slave_Interface_tb.v(128)
#    Time: 125 ns  Iteration: 0  Instance: /AHB_slave_interface_tb
```

Figure 10: ModelSim Terminal Outputs of `AHB_Slave_Interface_tb.v`

The AHB slave interface testbench simulates single read and write operations, burst modes, invalid addresses, and reset functionality. In all scenarios, the interface performs as expected: it reads data from `Prdata` when `Hwrite` is low, writes data to `Hwdata` when `Hwrite` is high, and sets `valid` to low for out-of-range addresses. Incremental and wrapping burst tests confirm no protocol stalls or data misalignments. Also, the reset check verifies that internal registers are properly cleared and remain consistent with `IDLE` state when `rst` is toggled to low.

6.2 FPV Results of APB_FSM.v

Figure 11 and 12 below show the FPV results for `APB_FSM.v`.



	Type	Name	Engine	Bound	Traces	Time	Task
✔	Assert	APB_FSM.APB_FSM_chk.assert_fsm_state_tra...	PRE	Infinite	0	0.0	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_idle2idle	N (8)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_idle2idle:prec...	PRE	1	1	0.0	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_idle2wait	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_idle2wait:prec...	PRE	1	1	0.0	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_idle2read	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_idle2read:pre...	N	1	1	<0.1	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_wait2write	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_wait2write:pr...	N	2	1	<0.1	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_wait2writep	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_wait2writep:p...	N	2	1	<0.1	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_read2renable	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_read2renable:...	N	2	1	<0.1	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_write2wenable	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_write2wenabl...	Hp	3	1	0.3	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_write2wenablep	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_write2wenabl...	Hp	3	1	0.3	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_writep2wena...	Hp (1)	Infinite	0	<0.1	<em
✔	Cover (related)	APB_FSM.APB_FSM_chk.assert_writep2wena...	Hp	3	1	0.3	<em
✔	Assert	APB_FSM.APB_FSM_chk.assert_renable2idle	Hp (1)	Infinite	0	<0.1	<em

Figure 11: Jasper Results of `APB_FSM_sva.sv`

```
================================================================
SUMMARY
================================================================
            Properties Considered                : 37
                  assertions                     : 19
                   - proven                      : 19 (100%)
                   - bounded_proven (user)       : 0 (0%)
                   - bounded_proven (auto)       : 0 (0%)
                   - marked_proven               : 0 (0%)
                   - cex                         : 0 (0%)
                   - ar_cex                      : 0 (0%)
                   - undetermined                : 0 (0%)
                   - unknown                     : 0 (0%)
                   - error                       : 0 (0%)
                  covers                         : 18
                   - unreachable                 : 0 (0%)
                   - bounded_unreachable (user): 0 (0%)
                   - covered                     : 18 (100%)
                   - ar_covered                  : 0 (0%)
                   - undetermined                : 0 (0%)
                   - unknown                     : 0 (0%)
                   - error                       : 0 (0%)
```

Figure 12: Jasper Terminal Outputs of `APB_FSM_sva.sv`

Formal verification exhaustively proved that the FSM transitions only according to the design's next-state logic. No counterexamples were generated for the main properties, which indicates that the FSM correctly progresses from `IDLE` to `WAIT`, `READ`, `WRITE`, and so on, depending on signals such as `valid` and `Hwrite`. Coverage of these transitions suggests that all states were reachable and no illegal transitions occurred. The properties ensuring that `Pwrite` is asserted only in valid write states and that `Penable` is toggled correctly also passed.

6.3 FPV Results of top.v

Figure 13 and 14 below show the FPV results for `top.v`.

✔	Cover (related)	Bridge_Top.chk_top.Back2back_data:precond...	N	2
✘	Assert	Bridge_Top.chk_top.Back2back_Pwrite	Ht	5
✔	Cover (related)	Bridge_Top.chk_top.Back2back_Pwrite:preco...	N	2
✘	Assert	Bridge_Top.chk_top.Burst_write_psel0	Ht	3
✔	Cover (related)	Bridge_Top.chk_top.Burst_write_psel0:precon...	Hp	2

Figure 13: Jasper Results of `top_sva.sv`

```
================================================================
SUMMARY
================================================================
           Properties Considered              : 53
                 assertions                   : 20
                  - proven                    : 18 (90%)
                  - bounded_proven (user)     : 0 (0%)
                  - bounded_proven (auto)     : 0 (0%)
                  - marked_proven             : 0 (0%)
                  - cex                       : 2 (10%)
                  - ar_cex                    : 0 (0%)
                  - undetermined              : 0 (0%)
                  - unknown                   : 0 (0%)
                  - error                     : 0 (0%)
                 covers                       : 33
                  - unreachable               : 0 (0%)
                  - bounded_unreachable (user): 0 (0%)
                  - covered                   : 33 (100%)
                  - ar_covered                : 0 (0%)
                  - undetermined              : 0 (0%)
                  - unknown                   : 0 (0%)
                  - error                     : 0 (0%)
```

Figure 14: Jasper Terminal Outputs of `top_sva.sv`

Almost all assertions pass with the full set of assumptions, confirming that the integration of the AHB slave interface and APB FSM is almost correct. How-

ever, two assertions fail under fully valid input assumptions: `Back2back_Pwrite` and `Burst_write_psel0`, as shown in Figure 15 and 16 below.

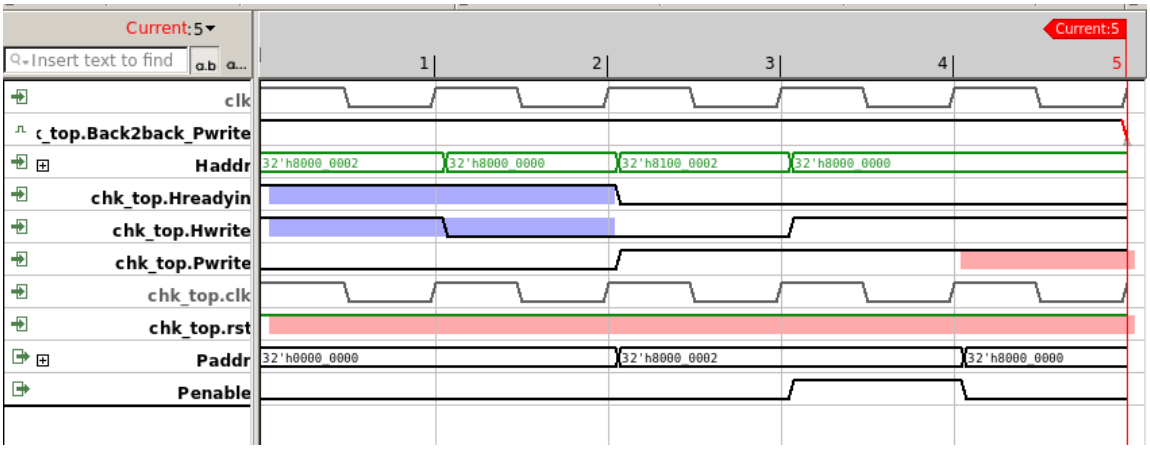


Figure 15: Jasper Wave Results of `Back2back_Pwrite` Assertion

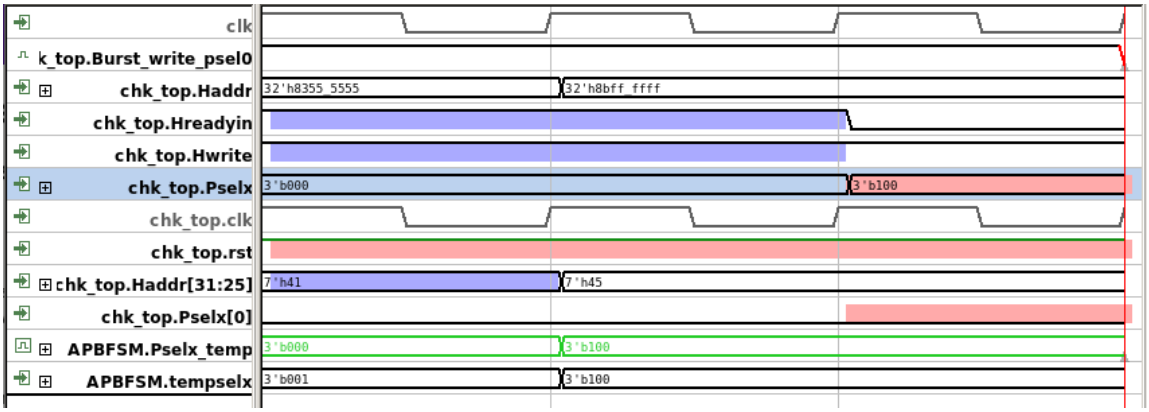


Figure 16: Jasper Wave Results of `Burst_write_psel0` Assertion

The scenario of assertion `Back2back_Pwrite` depicted in Figure 15 is as follows: start with a write operation, then a read operation, then an invalid, then an invalid, then an invalid. This is a back-to-back transfer, which must begin with a write operation; the expected waveform is illustrated in Figure 7 above. The counterexample in Figure 15 demonstrates that `Pwrite` is incorrectly high during the fifth cycle. In

the correct behavior, `Pwrite` in the fifth cycle should retain the value of `Hwrite` from the second cycle, which is low. The primary issue lies in the source RTL code, where `Pwrite` incorrectly takes the value of `Hwrite` from the fifth cycle.

The scenario of assertion `Burst_write_psel0` depicted in Figure 16 is as follows: The `Pselx` signal should be `3'b001` rather than `3'b100`. `3'b100` should occur in the fifth cycle instead of the third. In the source RTL code [3], `Pselx` does not transit correctly, leading to this issue.

We then remove the two fundamental assumptions: the valid address range for `Haddr` and the valid transfer type for `Htrans`, as previously discussed in the **FPV for top.v** section. This allows us to analyze the resulting counterexamples and demonstrate that these assumptions are critical for ensuring the validity of the FPV results.

First, we eliminate the assumption regarding the valid address range for `Haddr`. The results are presented in Figure 17, 18, and 19 below.

	Type	Name	Engine	Bour
✓	Cover (related)	Bridge_Top.chk_top.Read_data_transfer:prec...	N	3
✗	Assert	Bridge_Top.chk_top.Read_Pwrite	N	8
✓	Cover (related)	Bridge_Top.chk_top.Read_Pwrite:precondition1	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Read_Penable	N	2 - 3
✓	Cover (related)	Bridge_Top.chk_top.Read_Penable:preconditi...	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Burst_read_addr	N	3 - 8
✓	Cover (related)	Bridge_Top.chk_top.Burst_read_addr:precond...	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Burst_write_data1	N	3
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_data1:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_data2	N	5
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_data2:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Write_Pwrite	N	2 - 3
✓	Cover (related)	Bridge_Top.chk_top.Write_Pwrite:precondition1	N	2
✗	Assert	Bridge_Top.chk_top.Write_Penable1	N	4
✓	Cover (related)	Bridge_Top.chk_top.Write_Penable1:precondi...	N	2
✗	Assert	Bridge_Top.chk_top.Write_Penable2	N	4
✓	Cover (related)	Bridge_Top.chk_top.Write_Penable2:precondi...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_addr1	Hp	3
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_addr1:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_addr2	B	5
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_addr2:preco...	N	4
✗	Assert	Bridge_Top.chk_top.Back_to_back_Penable	Bm	4
✓	Cover (related)	Bridge_Top.chk_top.Back_to_back_Penable:p...	N	2
✓	Assert	Bridge_Top.chk_top.Back_to_back_Paddr	Hp (3)	Infini
✓	Cover (related)	Bridge_Top.chk_top.Back_to_back_Paddr:pre...	L	6
✗	Assert	Bridge_Top.chk_top.Read_after_write_addr	Hp	5
✓	Cover (related)	Bridge_Top.chk_top.Read_after_write_addr:pr...	N	2
✗	Assert	Bridge_Top.chk_top.Write_after_read_addr	Ht	7
✓	Cover (related)	Bridge_Top.chk_top.Write_after_read_addr:pr...	L	5
✗	Assert	Bridge_Top.chk_top.Back2back_data	Ht	3

Figure 17: Jasper Results of Remove `Haddr` Range Assumption

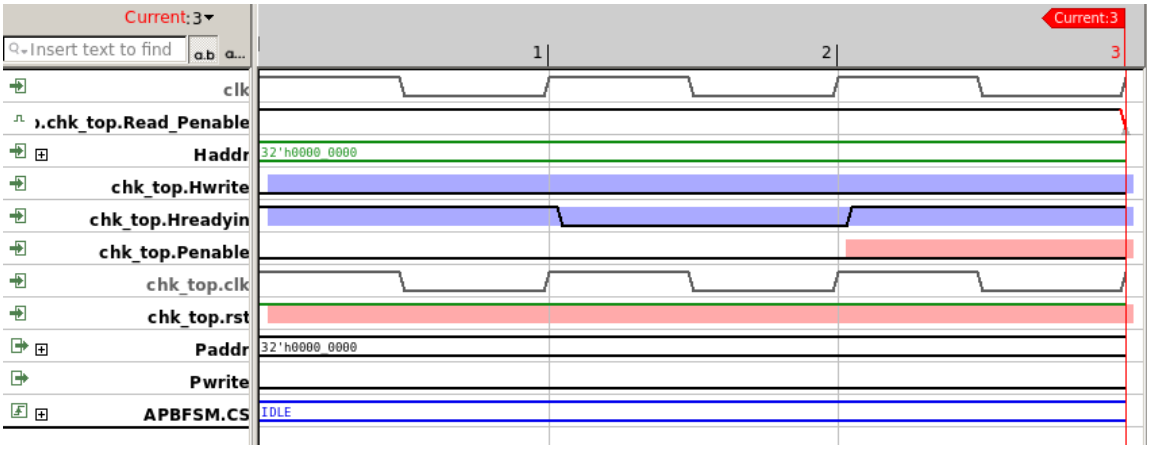


Figure 18: Jasper Wave Results of `Read_Penable` Assertion

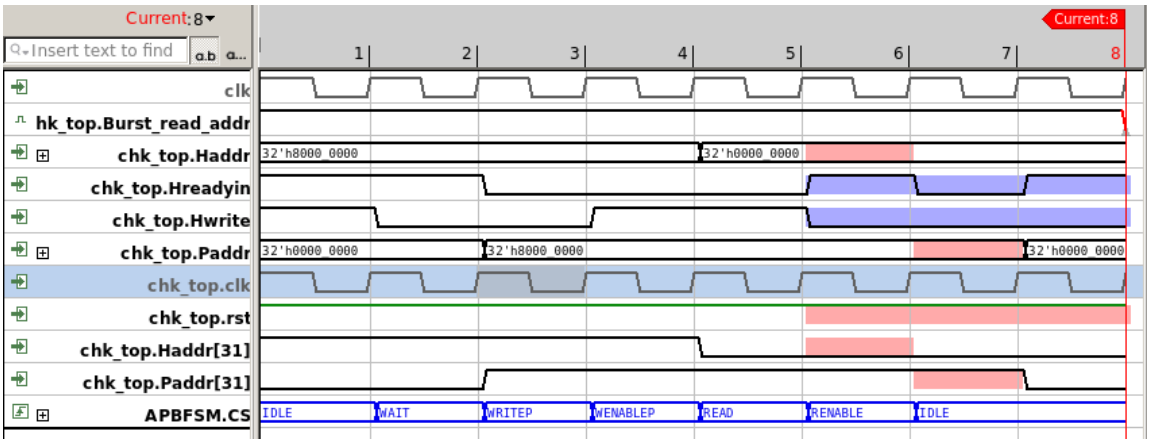


Figure 19: Jasper Wave Results of `Burst_read_addr` Assertion

Removing the assumption on valid address range causes the state machine to remain in `IDLE` for out-of-range addresses, which leads to failures in properties like `Read_Penable` and `Burst_read_addr` because a read sequence never begin, as shown in Figure 18 and 19 above.

Next, we eliminate the assumption regarding the valid transfer type for `Htrans`. The results are presented in Figure 20, 21, and 22 below.

	Type	Name	Engine	Bour
✓	Cover (related)	Bridge_Top.chk_top.Read_data_transfer:prec...	Hp	3
✗	Assert	Bridge_Top.chk_top.Read_Pwrite	Ht	8
✓	Cover (related)	Bridge_Top.chk_top.Read_Pwrite:precondition1	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Read_Penable	N	2 - 3
✓	Cover (related)	Bridge_Top.chk_top.Read_Penable:preconditi...	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Burst_read_addr	N	2 - 3
✓	Cover (related)	Bridge_Top.chk_top.Burst_read_addr:precond...	N	2 - 3
✗	Assert	Bridge_Top.chk_top.Burst_write_data1	Hp	3
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_data1:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_data2	Ht	5
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_data2:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Write_Pwrite	N	2 - 3
✓	Cover (related)	Bridge_Top.chk_top.Write_Pwrite:precondition1	N	2
✗	Assert	Bridge_Top.chk_top.Write_Penable1	Hp	4
✓	Cover (related)	Bridge_Top.chk_top.Write_Penable1:precondi...	N	2
✗	Assert	Bridge_Top.chk_top.Write_Penable2	Hp	4
✓	Cover (related)	Bridge_Top.chk_top.Write_Penable2:precondi...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_addr1	Hp	3
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_addr1:preco...	N	2
✗	Assert	Bridge_Top.chk_top.Burst_write_addr2	Hp	5
✓	Cover (related)	Bridge_Top.chk_top.Burst_write_addr2:preco...	Hp	4
✗	Assert	Bridge_Top.chk_top.Back_to_back_Penable	Hp	4
✓	Cover (related)	Bridge_Top.chk_top.Back_to_back_Penable:p...	N	2
✓	Assert	Bridge_Top.chk_top.Back_to_back_Paddr	Hp (3)	Infini
✓	Cover (related)	Bridge_Top.chk_top.Back_to_back_Paddr:pre...	Hp	6
✗	Assert	Bridge_Top.chk_top.Read_after_write_addr	Hp	5
✓	Cover (related)	Bridge_Top.chk_top.Read_after_write_addr:pr...	N	2
✗	Assert	Bridge_Top.chk_top.Write_after_read_addr	Hp	7
✓	Cover (related)	Bridge_Top.chk_top.Write_after_read_addr:pr...	Hp	5
✗	Assert	Bridge_Top.chk_top.Back2back_data	Hp	3

Figure 20: Jasper Results of Remove `Htrans` Type Assumption

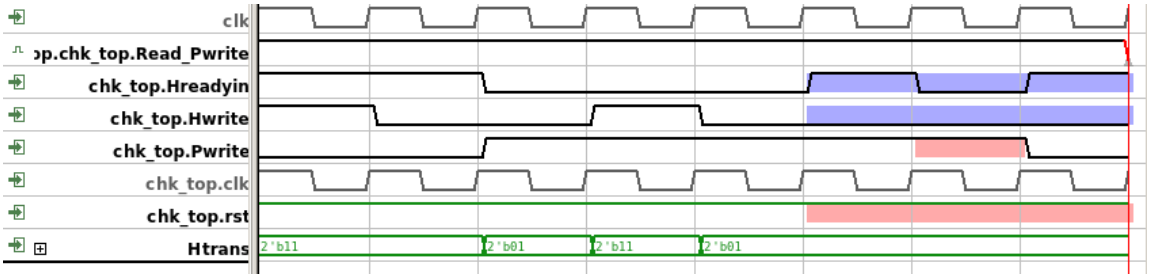


Figure 21: Jasper Wave Results of `Read_Pwrite` Assertion

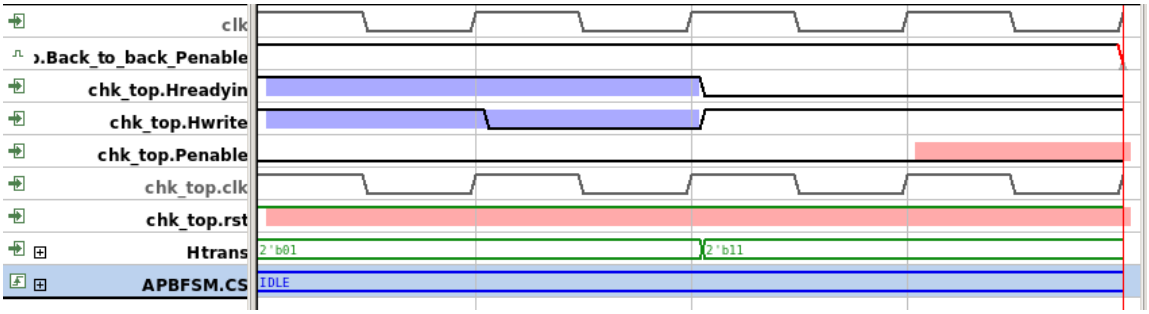


Figure 22: Jasper Wave Results of `Back_to_back_Penable` Assertion

Eliminating the assumption on valid `Htrans` type results in assertion failures such as `Read_Pwrite` (in Figure 21) and `Back_to_back_Penable` (in Figure 22). These issues arise because the APB FSM encounters `Htrans` in states such as `BUSY 2'b01` or `IDLE 2'b00`, leading to either a stalled state or incorrect transitions.

Hence, the results of these tests without the assumptions confirm that the bridge was not designed to handle transfer types or addresses beyond the constraints outlined in the AMBA specification document [1].

6.4 Testbench Results of top.v

Figure 23 below shows the testbench results for `top.v`.

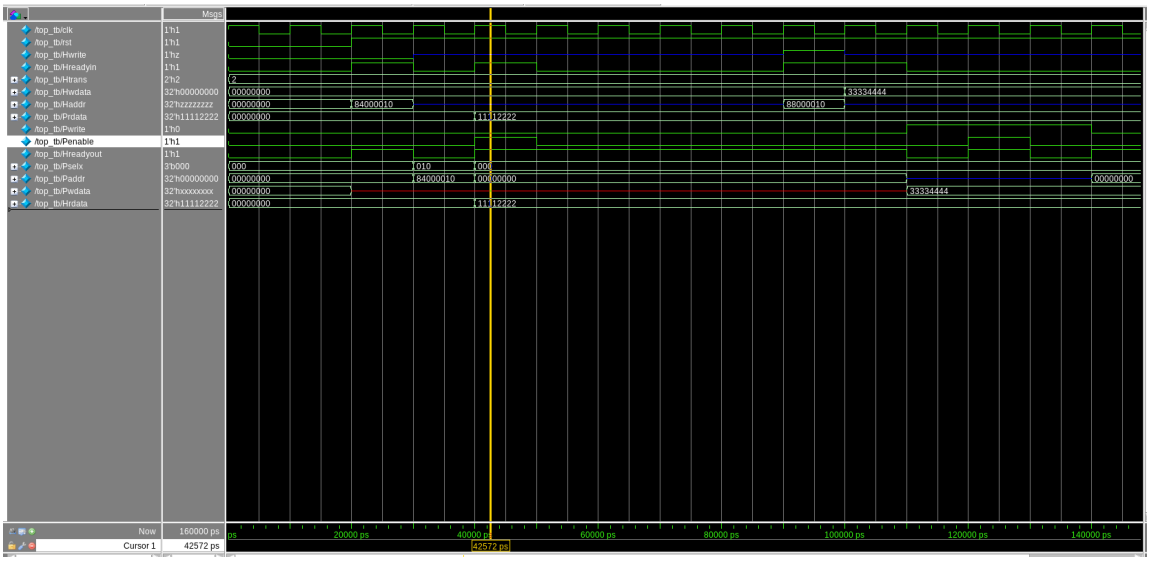


Figure 23: ModelSim Testbench Wave Results of `top_tb.v`

When a valid address, `Hwrite`, and `Htrans` are applied for a read operation, the testbench confirms that the design correctly returns data via `Hrdata` when `Prdata` is driven by the simulated APB side. In a subsequent single-write scenario, the testbench verifies that both the data and address are properly latched on the APB side. However, the `Pselx` signal is still incorrect.

6.5 FPV Results v.s. Testbench Results of top.v

As described in the **Testbench Results of top.v** section above, the simulation testbench `top_tb.v` effectively handles single-read and single-write operations, identifying errors such as `Pselx` misbehavior in specific scenarios. It validates nominal transactions and detects straightforward signal routing errors.

In contrast, the FPV explores a broader range of scenarios by exhaustively verifying all states while enforcing constraints on valid address ranges and transaction types. This ensures that any invalid conditions immediately trigger property failures. For example, `top_tb.v` lacks explicit logic to prevent random or incorrect addresses or transaction types, allowing such issues to pass unnoticed in simulation. However, the same design that performs well during single-read and single-write sequences in simulation testbench may fail in formal verification if assumptions on valid addresses or valid transaction types are removed.

The FPV environment is also good for verifying complex state transitions in the APB FSM, particularly under burst and back-to-back operations. It identifies timing violations and deeper pipeline issues that may not manifest in the testbench. Unlike simulation, which primarily focuses on directed or random nominal checks, the FPV environment exhaustively explores all possible sequential and non-sequential transaction permutations, identifying errors that might otherwise remain undetected.

7 Conclusions and Future Improvements

The combined use of simulation test benches and FPV has provided a comprehensive evaluation of the AHB-to-APB bridge. Simulation of the AHB interface demonstrated correct behavior for single and burst transactions, detecting invalid addresses, and functionality of the active low reset. FPV confirmed the correctness of the APB FSM across all states, with no illegal transitions detected. At the top bridge level, FPV identified two specific issues, `Back2back_Pwrite` and `Burst_write_psel0`, which highlight timing or control logic flaws in the RTL. The removal of assumptions about valid address ranges or transfer types led to predictable property failures, emphasizing that the design is not intended to handle out-of-specification scenarios. Simulation at the top bridge level validates functionality for single transactions, while formal verification ensures exhaustive coverage of corner cases, ensuring strong robustness of the AHB-to-APB bridge design. Overall, all results suggest that, with appropriate fixes to the identified bugs, the source RTL code will obey the AMBA specification document [1] under normal operating conditions.

Several enhancements can be made to improve the design and verification processes. First, the errors identified in the source RTL code should be fixed to ensure the AHB-to-APB bridge operates as the AMBA specification document [1]. Moreover, if time permits, the verification scope can be expanded to include a broader range of scenarios and edge cases, providing more thorough coverage of potential issues. Last but not least, the integration of verification tools can be optimized, and the FPV methodology can be extended to other AMBA components, enabling a more comprehensive and robust AMBA system validation.

References

[1] A. Ltd., *AMBA Specification (Rev 2.0)*, 1999, accessed: 2024-12-21. [Online]. Available: https://documentation-service.arm.com/static/642569a2314e245d086bc87e?token=

[2] Wikipedia contributors, "Advanced microcontroller bus architecture — Wikipedia, the free encyclopedia," 2024, [Online; accessed 22-December-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Advanced_Microcontroller_Bus_Architecture&oldid=1250943266

[3] P. Gekkouga, "Ahb-to-apb bridge implementation," https://github.com/prajwalgekkouga/AHB-to-APB-Bridge/tree/main, 2022, accessed: 2024-12-21.