Formal Verification of AHB-to-APB Bridge

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

Fall 2024

Contents

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 pivotal 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 rigorously 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. We will discuss formal verification briefly in the **Short Introduction of Formal Verification** section.

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. Its technology-independent nature allows for the reuse of intellectual property (IP) cores across diverse integrated circuit (IC) processes, encouraging modular system design and the development of reusable peripheral and system IP libraries.

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.
- Advanced eXtensible Interface (AXI): Part of AMBA 3 (2003), AXI is targeted at high-performance, high-frequency system designs and includes features that make it suitable for high-speed sub-micrometer interconnects.
- Advanced Trace Bus (ATB): Also introduced in AMBA 3, ATB is part of the CoreSight on-chip debug and trace solution.
- AXI Coherency Extensions (ACE): Introduced in AMBA 4 (2010), ACE extends AXI with additional signaling to introduce system-wide coherency, allowing multiple processors to share memory efficiently.
- Coherent Hub Interface (CHI): Part of AMBA 5 (2013), CHI features a redesigned high-speed transport layer and is designed to reduce congestion in complex SoC designs.

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.

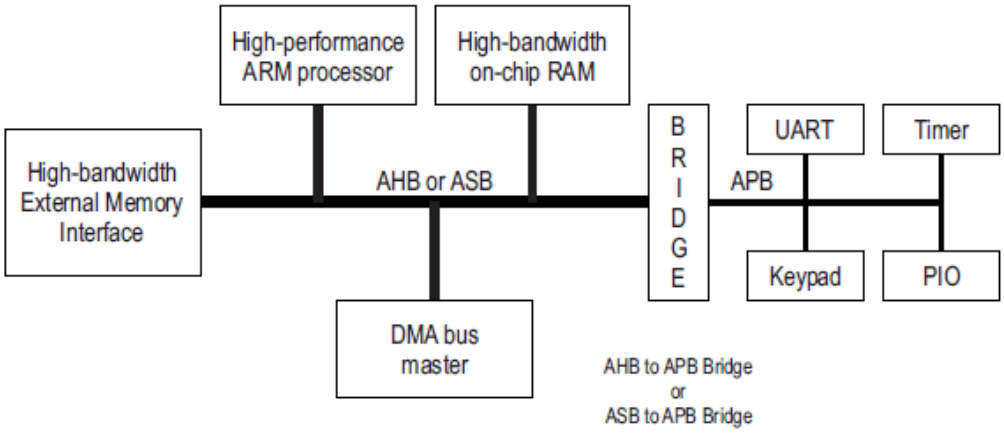


Figure 1: AHB and APB with Communication Bridge

3 Short Introduction of Formal Verification

Formal verification is a mathematical approach used to prove or disprove the correctness of hardware and software systems relative to a formal specification or set of properties. Unlike traditional simulation-based methods that test a system's behavior under specific scenarios, formal verification uses model checking to exhaustively examine all possible states and inputs to ensure compliance with the desired specifications. This exhaustive analysis makes formal verification particularly effective in identifying corner-case bugs that might be missed during conventional testing.

In formal verification, especially within hardware design, several key constructs are utilized to define and check system behaviors [3]:

- Assertions: Statements that specify expected behavior or properties of the system. They are used to ensure that certain conditions hold `TRUE` during system operation. If an assertion fails, it indicates a violation of the specified property, signaling a potential design flaw.
- Assumptions: Conditions presumed to be `TRUE` within the verification environment. Assumptions constrain the input space and operating conditions, guiding the formal tools to focus on relevant scenarios. They help simplify the verification process by limiting the analysis to feasible and meaningful real-world cases.
- Covers: Statements used to confirm that certain scenarios or behaviors can occur in the system. They are essential for functional coverage analysis, ensuring that all intended behaviors have been exercised and verified during the verification process.

Modern formal verification tools offer specialized applications to address various verification challenges [4]:

- Formal Property Verification (FPV): This application focuses on verifying that the design adheres to specified properties or assertions. FPV exhaustively checks all possible states and transitions to ensure that the design behaves as intended under all conditions.
- Automatic Extracted Properties (AEP): AEP tools automatically derive properties from the design without requiring manual assertion writing. These extracted properties can then be analyzed to identify potential issues or to confirm expected behaviors, streamlining the verification process.

- Formal X-Propagation Verification (FXP): FXP targets the detection and analysis of unknown signal values (denoted as 'X') within the design. It checks for unintended propagation of these unknowns, which can lead to unpredictable behavior, ensuring that the design can handle 'X' states robustly.
- Other related applications: Formal Coverage Analyzer (FCA), Connectivity Checking (CC), and so on.

Our project primarily relies on assertions and assumptions, as covers can be effectively represented through assertions. Additionally, we focus on using FPV, as it allows us to verify that the design complies with the specification document under all conditions.

4 Source Code of AHB-to-APB Bridge

4.1 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.

4.2 Source RTL Code Analysis

The RTL code we utilized is available here [5]. 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.

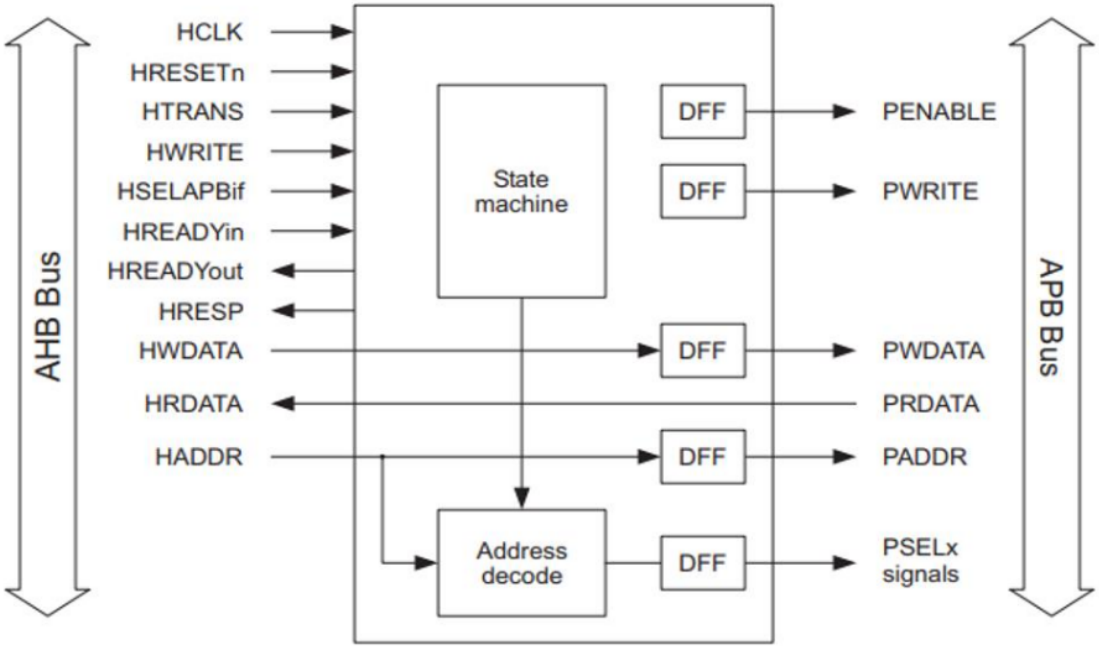


Figure 2: Mechanism of the AHB-to-APB Bridge

4.2.1 AHB_Master.v

The code in `AHB_Master.v` is shown below.

```
module AHB_Master(Hclk, Hresetn, Hresp, Hrdata, Hwrite,
                  Hreadyin, Hreadyout, Htrans, Hwdata, Haddr);

input Hclk, Hresetn, Hreadyout;
input [1:0] Hresp;
input [31:0] Hrdata;
output reg Hwrite, Hreadyin;
output reg [1:0] Htrans;
output reg [31:0] Hwdata, Haddr;

reg [2:0] Hburst;
reg [2:0] Hsize;

task single_write();
 begin
  @(posedge Hclk)
  #2;
   begin
    Hwrite=1;
    Htrans=2'b10;
    Hsize=3'b000;
    Hburst=3'b000;
    Hreadyin=1;
    Haddr=32'h8000_0001;
   end

  @(posedge Hclk)
  #2;
   begin
    Htrans=2'b00;
    Hwdata=8'hA3;
   end
 end
endtask
```

```
task single_read();
 begin
  @(posedge Hclk)
  #2;
   begin
    Hwrite=0;
    Htrans=2'b10;
    Hsize=3'b000;
    Hburst=3'b000;
    Hreadyin=1;
    Haddr=32'h8000_00A2;
   end

  @(posedge Hclk)
  #2;
   begin
    Htrans=2'b00;
   end
 end
endtask

endmodule
```

Port and Register Descriptions:

- **Input Ports:**
  - **Hclk**: Clock signal.
  - **Hresetn**: Active-low reset signal.
  - **Hreadyout**: Indicates whether the AHB slave is ready for a transaction.
  - **Hresp [1:0]**: Response from the AHB slave, typically indicating if there was an error or if the transaction was successful.
  - **Hrdata [31:0]**: Data read from the AHB slave.
- **Output Ports:**
  - **Hwrite**: Indicates whether the transaction is a write (1) or a read (0).
  - **Hreadyin**: Indicates whether the AHB master is ready for a transaction.

  - **Htrans [1:0]**: Transfer type (Idle 00, Busy 01, Non-sequential 10, Sequential 11).
  - **Hwdata [31:0]**: Data to be written to the AHB slave.
  - **Haddr [31:0]**: Address for the transaction.

- **Register Declarations:**
  - **Hburst [2:0]**: Defines the burst type, indicating the number of transfers to be performed.
  - **Hsize [2:0]**: Specifies the size of the transfer (byte 000, half-word 001, word 010, and so on).

The source code of AHB master is a straightforward component with minimal complexity, designed to perform single read or write operations in a fixed mode (`Htrans`, `Hburst`, and `Hsize` are all fixed). It provides predetermined outputs to the AHB-to-APB Bridge (at the AHB interface end) to facilitate testing and simulation. For comprehensive verification, however, this code will be ignored. The input to the bridge will be regulated using formal verification assumptions, as described in the **FPV for top.v** section.

4.2.2 AHB_Interface.v

The code in `AHB_Interface.v` is shown below.

```
module AHB_slave_interface(clk, rst, Hwrite, Hreadyin, Htrans,
                           Haddr, Hwdata, Prdata, valid, Haddr1,
                           Haddr2, Hwdata1, Hwdata2, Hrdata,
                           Hwritereg, tempselx, Hresp);
input clk, rst;
input Hwrite, Hreadyin;
input [1:0] Htrans;
input [31:0] Haddr, Hwdata, Prdata;
output reg valid;
output reg [31:0] Haddr1, Haddr2, Hwdata1, Hwdata2;
output [31:0] Hrdata;
output reg Hwritereg;
output reg [2:0] tempselx;
output  [1:0] Hresp;

/// Implementing Pipeline Logic for Address,Data and Control
↪  Signal
        always @(posedge clk)
                begin
                        if (~rst)
                                begin
                                        Haddr1<=0;
                                        Haddr2<=0;
                                end
                        else
                                begin
                                        Haddr1<=Haddr;
                                        Haddr2<=Haddr1;
                                end
                end

        always @(posedge clk)
                begin
                        if (~rst)
                                begin
```

```
                                        Hwdata1<=0;
                                        Hwdata2<=0;
                                end
                        else
                                begin
                                        Hwdata1<=Hwdata;
                                        Hwdata2<=Hwdata1;
                                end
                end

        always @(posedge clk)
                begin
                        if (~rst)
                                Hwritereg<=0;
                        else
                                Hwritereg<=Hwrite;
                end

/// Implementing Valid Logic Generation
        always @(Hreadyin, Haddr, Htrans, rst)
                begin
                        valid=0;
                        if (rst && Hreadyin &&
                        ↪  (Haddr>=32'h8000_0000 &&
                        ↪  Haddr<32'h8C00_0000) &&
                        ↪  (Htrans==2'b10 || Htrans==2'b11) )
                                valid=1;
                end

/// Implementing Tempselx Logic
        always @(Haddr, rst)
                begin
                        tempselx=3'b000;
                        if (rst && Haddr>=32'h8000_0000 &&
                        ↪  Haddr<32'h8400_0000)
                                tempselx=3'b001;
                        else if (rst && Haddr>=32'h8400_0000 &&
                        ↪  Haddr<32'h8800_0000)
```

```
                                tempselx=3'b010;
                            else if (rst && Haddr>=32'h8800_0000 &&
                            ↪  Haddr<32'h8C00_0000)
                                tempselx=3'b100;
                    end

assign Hrdata = Prdata;
assign Hresp=2'b00;

endmodule
```

Port Descriptions:

- **Input Ports:**
  - **clk**: Clock signal.
  - **rst**: Active-low reset signal.
  - **Hwrite**: Indicates whether the transaction is a write (1) or a read (0).
  - **Hreadyin**: Indicates whether the AHB master is ready to accept the transaction.
  - **Htrans [1:0]**: Transfer type (Idle 00, Busy 01, Non-sequential 10, Sequential 11).
  - **Haddr [31:0]**: Address for the transaction.
  - **Hwdata [31:0]**: Data from the AHB master that needs to be written to the APB.
  - **Prdata [31:0]**: Data from the APB that needs to be read to the AHB master.
- **Output Ports:**
  - **valid**: Indicates if the address and transaction are valid.
  - **Haddr1, Haddr2 [31:0]**: Pipeline registers for the address.
  - **Hwdata1, Hwdata2 [31:0]**: Pipeline registers for the data to be written to the APB.
  - **Hrdata [31:0]**: Data read from the APB.
  - **Hwritereg**: Pipeline register for the write control signal.

- **tempselx [2:0]**: Used to select different address ranges.
- **Hresp [1:0]**: Response to the AHB Master (OKAY 00, ERROR 01, RETRY 10, SPLIT 11).

The AHB slave interface employs pipelining mechanisms to manage addresses, write data, and control signals, ensuring synchronization and accurate processing of transactions. These pipelines are implemented through sequential logic blocks, as illustrated in the following:

- **Address Pipeline:**
  - The address signal (`Haddr`) is buffered in a two-stage pipeline (`Hadd1` and `Haddr2`).
  - This introduces a delay, allowing the AHB slave sufficient time to process transactions while ensuring alignment with the data and control signals.
- **Write Data Pipeline:**
  - The write data signal (`Hwdata`) is buffered in a two-stage pipeline (`Hwdata1` and `Hwdata2`).
  - This ensures proper synchronization with the address pipeline for write operations.
- **Write Control Signal Pipeline:**
  - The write control signal (`Hwrite`) is stored in a register (`Hwritereg`).
  - This aligns the control signal with the pipelined address and data signals.
- **Valid Logic Generation:**
  - The valid signal determines whether a transaction is ready to be processed.
  - It is asserted when:
    - The reset signal is inactive.
    - The AHB slave is ready to accept transactions.
    - The address falls within a specified range.
    - The transaction type is valid (Non-sequential 10 or Sequential 11).
  - This ensures only valid transactions are processed by the system.
- **Address Range Selection Logic:**

- The logic for `tempselx` assigns values based on the address range:
  - `32'h8000_0000` to `32'h8400_0000`: `tempselx = 3'b001`.
  - `32'h8400_0000` to `32'h8800_0000`: `tempselx = 3'b010`.
  - `32'h8800_0000` to `32'h8C00_0000`: `tempselx = 3'b100`.
- This division enables differentiated handling of address ranges, potentially designating specific peripherals or memory blocks.

- **Read Data and Response Signals:**
  - **Read Data (`Hrdata`):** Directly passes the data from the peripheral (`Prdata`) to the master, ensuring the master can access the requested data.
  - **Response Signal (`Hresp`):** Indicates the transaction status, with a value of `2'b00` typically representing a "OKAY" operation.

As discussed in the **Source RTL Code Analysis** section, this code functions as a core component of the AHB-to-APB bridge. Since the AHB slave interface primarily serves as an I/O module, simulation testing will be used instead of formal property verification (FPV) for simplicity. Further details are provided in the **Testbench for AHB_Interface.v** and **Testbench Results of AHB_Interface.v** sections below.

4.2.3 APB_FSM.v

The code in AHB_FSM.v is shown below.

```
module APB_FSM(clk, rst, valid, Haddr1,
               Haddr2, Hwdata1, Hwdata2, Prdata,
               Hwrite, Haddr, Hwdata, Hwritereg,
               tempselx, Pwrite, Penable, Pselx,
               Paddr, Pwdata, Hreadyout);

input clk,rst,valid,Hwrite,Hwritereg;
input [31:0] Hwdata,Haddr,Haddr1,Haddr2,Hwdata1,Hwdata2,Prdata;
input [2:0] tempselx;
output reg Pwrite,Penable;
output reg Hreadyout;
output reg [2:0] Pselx;
output reg [31:0] Paddr,Pwdata;

//////////////////////////////////////////////////////states
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;

////////////////////////////// PRESENT STATE LOGIC
reg [2:0] CS,NS;

always @(posedge clk or negedge rst)
 begin:CS_LOGIC
  if (~rst)
    CS<=IDLE;
  else
    CS<=NS;
 end
```

```
//////////////////////////////////// NEXT STATE LOGIC
always @(CS,valid,Hwrite,Hwritereg)
 begin:NS_LOGIC
  case (CS)
         IDLE:begin
                 if (~valid)
                  NS=IDLE;
                 else if (valid && Hwrite)
                  NS=WAIT;
                 else
                  NS=READ;
                end

        WAIT:begin
                  if (~valid)
                   NS=WRITE;
                  else
                   NS=WRITEP;
                 end

        READ: begin
                    NS=RENABLE;
                  end

        WRITE:begin
                   if (~valid)
                    NS=WENABLE;
                   else
                    NS=WENABLEP;
                  end

        WRITEP:begin
                    NS=WENABLEP;
                   end

        RENABLE:begin
                      if (~valid)
                       NS=IDLE;
```

```
                    else if (valid && Hwrite)
                     NS=WAIT;
                    else
                     NS=READ;
                  end

        WENABLE:begin
                    if (~valid)
                     NS=IDLE;
                    else if (valid && Hwrite)
                     NS=WAIT;
                    else
                     NS=READ;
                  end

        WENABLEP:begin
                     if (~valid && Hwritereg)
                      NS=WRITE;
                     else if (valid && Hwritereg)
                      NS=WRITEP;
                     else
                      NS=READ;
                   end

        default: begin
                    NS=IDLE;
                  end
  endcase
 end

//////////////////////////OUTPUT LOGIC:COMBINATIONAL
reg Penable_temp,Hreadyout_temp,Pwrite_temp;
reg [2:0] Pselx_temp;
reg [31:0] Paddr_temp, Pwdata_temp;

always @(*)
 begin:OUTPUT_COMBINATIONAL_LOGIC
   case(CS)
```

```
IDLE: begin
                  if (valid && ~Hwrite)
                   begin:IDLE_TO_READ
                        Paddr_temp=Haddr;
                        Pwrite_temp=Hwrite;
                        Pselx_temp=tempselx;
                        Penable_temp=0;
                        Hreadyout_temp=0;
                   end

                  else if (valid && Hwrite)
                   begin:IDLE_TO_WWAIT
                        Pselx_temp=0;
                  //Pselx_temp=tempselx;
                  Penable_temp=0;
                  Hreadyout_temp=1;
                   end

                  else
                    begin:IDLE_TO_IDLE
                        Pselx_temp=0;
                        Penable_temp=0;
                        Hreadyout_temp=1;
                        Paddr_temp='0;
                        Pwrite_temp='0;
                   end
              end

WAIT:begin
          if (~valid)
                   begin:WAIT_TO_WRITE
                    Paddr_temp=Haddr1;
                        Pwrite_temp=1;
                        Pselx_temp=tempselx;
                        Penable_temp=0;
                        Pwdata_temp=Hwdata;
                        Hreadyout_temp=0;
```

```
                    end

                  else
                   begin:WAIT_TO_WRITEP
                    Paddr_temp=Haddr1;
                        Pwrite_temp=1;
                        Pselx_temp=tempselx;
                        Pwdata_temp=Hwdata;
                        Penable_temp=0;
                        Hreadyout_temp=0;
                   end

              end

READ: begin:READ_TO_RENABLE
                  Penable_temp=1;
                  Hreadyout_temp=1;
              end

WRITE:begin
      if (~valid)
                   begin:WRITE_TO_WENABLE
                        Penable_temp=1;
                        Hreadyout_temp=1;
                   end

                  else
                   begin:WRITE_TO_WENABLEP ///DOUBT
                        Penable_temp=1;
                        Hreadyout_temp=1;
                   end
              end

WRITEP:begin:WRITEP_TO_WENABLEP
       Penable_temp=1;
                    Hreadyout_temp=1;
             end
```

```
RENABLE:begin
            if (valid && ~Hwrite)
                        begin:RENABLE_TO_READ
                               Paddr_temp=Haddr;
                               Pwrite_temp=Hwrite;
                               Pselx_temp=tempselx;
                               Penable_temp=0;
                               Hreadyout_temp=0;
                        end

                  else if (valid && Hwrite)
                    begin:RENABLE_TO_WWAIT
                     Pselx_temp=0;
                         Penable_temp=0;
          Hreadyout_temp=1;
                    end

                  else
        begin:RENABLE_TO_IDLE
                     Pselx_temp=0;
                         Penable_temp=0;
                         Hreadyout_temp=1;
                    end

               end

WENABLEP:begin
         if (~valid && Hwritereg)
                      begin:WENABLEP_TO_WRITEP
                       Paddr_temp=Haddr2;
                           Pwrite_temp=Hwrite;
                           Pselx_temp=tempselx;
                           Penable_temp=0;
                           Pwdata_temp=Hwdata;
                           Hreadyout_temp=0;
                          end
```

```
                            else
                             begin:WENABLEP_TO_WRITE_OR_READ
                             ↪  /////DOUBT
                              Paddr_temp=Haddr2;
                                  Pwrite_temp=Hwrite;
                                  Pselx_temp=tempselx;
                                  Pwdata_temp=Hwdata;
                                  Penable_temp=0;
                        Hreadyout_temp=0;
                                 end
                            end

          WENABLE :begin
                        if (~valid && Hwritereg)
                                 begin:WENABLE_TO_IDLE
                                      Pselx_temp=0;
                                      Penable_temp=0;
                                      Hreadyout_temp=0;
                                     end


                            else
                             begin:WENABLE_TO_WAIT_OR_READ
                             ↪  /////DOUBT
                                  Pselx_temp=0;
                                  Penable_temp=0;
                        Hreadyout_temp=0;
                                 end

                        end
 endcase
end

////////////////////////////////////OUTPUT LOGIC:SEQUENTIAL
always @(posedge clk or negedge rst)
 begin

  if (~rst)
```

```
    begin
     Paddr<=0;
          Pwrite<=0;
          Pselx<=0;
          Pwdata<=0;
          Penable<=0;
          Hreadyout<=0;
    end

   else
    begin
          Paddr<=Paddr_temp;
          Pwrite<=Pwrite_temp;
          Pselx<=Pselx_temp;
          Pwdata<=Pwdata_temp;
          Penable<=Penable_temp;
          Hreadyout<=Hreadyout_temp;
    end
  end

 endmodule
```

Port Descriptions:

- **Input Ports:**
  - **clk:** Clock signal.
  - **rst** Active-low reset signal.
  - **valid:** Indicates if the address and transaction are valid.
  - **Hwrite:** Indicates whether the transaction is a write (1) or a read (0).
  - **Hwritereg:** Pipeline register for the write control signal.
  - **Hwdata, Haddr, Haddr1, Haddr2, Hwdata1, Hwdata2:** Address and data signals coming from the AHB side.
  - **Prdata:** Data read from the APB peripheral.
  - **tempselx:** Used for address decoding to select specific peripherals on the APB.
- **Output Ports:**

- **Pwrite:** Indicates if the current APB operation is a write (1) or a read (0).
- **Penable:** Controls when the APB transaction is enabled.
- **Pselx:** APB peripheral selection signal.
- **Paddr:** The address for the APB transaction.
- **Pwdata:** Data for APB write transactions.
- **Hreadyout:** Signal indicating to the AHB side that the APB bridge is ready.

The FSM controlling the AHB-to-APB bridge consists of several states, logic to determine the current and next states, and output signal generation, as demonstrated in the following:

- **State Definitions:**
  - The FSM includes predefined states such as `IDLE`, `WAIT`, `READ`, `WRITE`, `WRITEP`, and various enable states.
  - These states represent different stages of APB transactions, including idle periods, write and read operations, and enabling control signals.
- **Present State Logic:**
  - The current state of the FSM is updated on each clock cycle.
  - When reset is active, the FSM transitions to the `IDLE` state.
  - Otherwise, the state transitions are driven by the next state logic.
- **Next State Logic:**
  - The FSM transitions between states based on the current state and input signals such as `valid`, `Hwrite`, and `Hwritereg`.
- **Output Logic:**
  - The output logic generates control signals required for APB transactions based on the FSM state.
  - Each state has specific logic for configuring output signals such as `Paddr`, `Pwrite`, `Pselx`, and `Penable`.
- **Sequential Output Logic:**

- The final output signals are updated on each clock cycle based on the temporary values computed in the combinational logic.
- This ensures synchronization of outputs such as `Paddr`, `Pwrite`, `Pselx`, `Pwdata`, `Penable`, and `Hreadyout` with the clock signal.

As outlined in the **Source RTL Code Analysis** section, this code serves as the heart of the AHB-to-APB bridge. FPV will be employed to ensure comprehensive verification. Additional details can be found in the **FPV for APB_FSM.v** and **FPV Results of APB_FSM.v** sections below.

4.2.4 AHB_Interface.v

The code in `APB_Interface.v` is shown below.

```
module APB_Interface(Pwrite, Pselx, Penable, Paddr,
                     Pwdata, Pwriteout, Pselxout, Penableout,
                     Paddrout, Pwdataout, Prdata);

input Pwrite, Penable;
input [2:0] Pselx;
input [31:0] Pwdata, Paddr;

output Pwriteout, Penableout;
output [2:0] Pselxout;
output [31:0] Pwdataout, Paddrout;
output reg [31:0] Prdata;

assign Penableout=Penable;
assign Pselxout=Pselx;
assign Pwriteout=Pwrite;
assign Paddrout=Paddr;
assign Pwdataout=Pwdata;

always @(*)
 begin
  if (~Pwrite && Penable)
   Prdata=($random)%256;
  else
   Prdata=0;
 end

endmodule
```

Port Descriptions:

- **Input Ports:**
  - **Pwrite:** Indicates if the current APB operation is a write (1) or a read (0).
  - **Penable:** Controls when the APB transaction is enabled.

  - **Pselx [2:0]:** APB peripheral selection signal.
  - **Paddr [31:0]:** The address for the APB transaction.
  - **Pwdata [31:0]:** Data for APB write transactions.

- **Output Ports:**
  - **Pwriteout:** Forwarded value of the Pwrite signal.
  - **Penableout:** Forwarded value of the Penable signal.
  - **Pselxout [2:0]:** Forwarded value of the Pselx signal.
  - **Paddrout [31:0]:** Forwarded value of the Paddr signal.
  - **Pwdataout [31:0]:** Forwarded value of the Pwdata signal.
  - **Prdata [31:0]:** Data read from the APB peripheral.

The APB interface source code is a simple component with minimal complexity, created to simulate basic responses from a peripheral. This code is not relevant to our verification process and will be ignored.

4.2.5 top.v

The code in `top.v` is shown below.

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

input clk,rst,Hwrite,Hreadyin;
input [31:0] Hwdata,Haddr,Prdata;
input[1:0] Htrans;
output Penable,Pwrite,Hreadyout;
output [1:0] Hresp;
output [2:0] Pselx;
output [31:0] Paddr,Pwdata;
output [31:0] Hrdata;

//////////////////////////////////////INTERMEDIATE SIGNALS
wire valid;
wire [31:0] Haddr1,Haddr2,Hwdata1,Hwdata2;
wire Hwritereg;
wire [2:0] tempselx;

//////////////////////////////////////MODULE INSTANTIATIONS
AHB_slave_interface AHBSlave (clk,rst,Hwrite,Hreadyin,
                              Htrans,Haddr,Hwdata,Prdata,
                              valid,Haddr1,Haddr2,Hwdata1,
                              Hwdata2,Hrdata,Hwritereg,
                              tempselx,Hresp);

APB_FSM APBFSM (clk,rst,valid,Haddr1,
                Haddr2,Hwdata1,Hwdata2,Prdata,
                Hwrite,Haddr,Hwdata,Hwritereg,
                tempselx,Pwrite,Penable,Pselx,
                Paddr,Pwdata,Hreadyout);

endmodule
```

The port descriptions can be found in the **AHB_Interface.v** and **APB_FSM.v**

sections above.

As described in the **Source RTL Code Analysis** section, this core top module serves as a bridge, integrating the AHB slave interface with the APB FSM and instantiating them. Hence, the majority of our formal verification efforts are concentrated on this top module. To achieve comprehensive verification, both simulation testbenches and FPV will be employed. Detailed explanations can be found in the **FPV for top.v**, **Testbench for top.v**, **FPV Results of top.v**, and **Testbench Results of top.v** sections below.

5 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.

6 Formal Verification Processes

This project employs a modular 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.

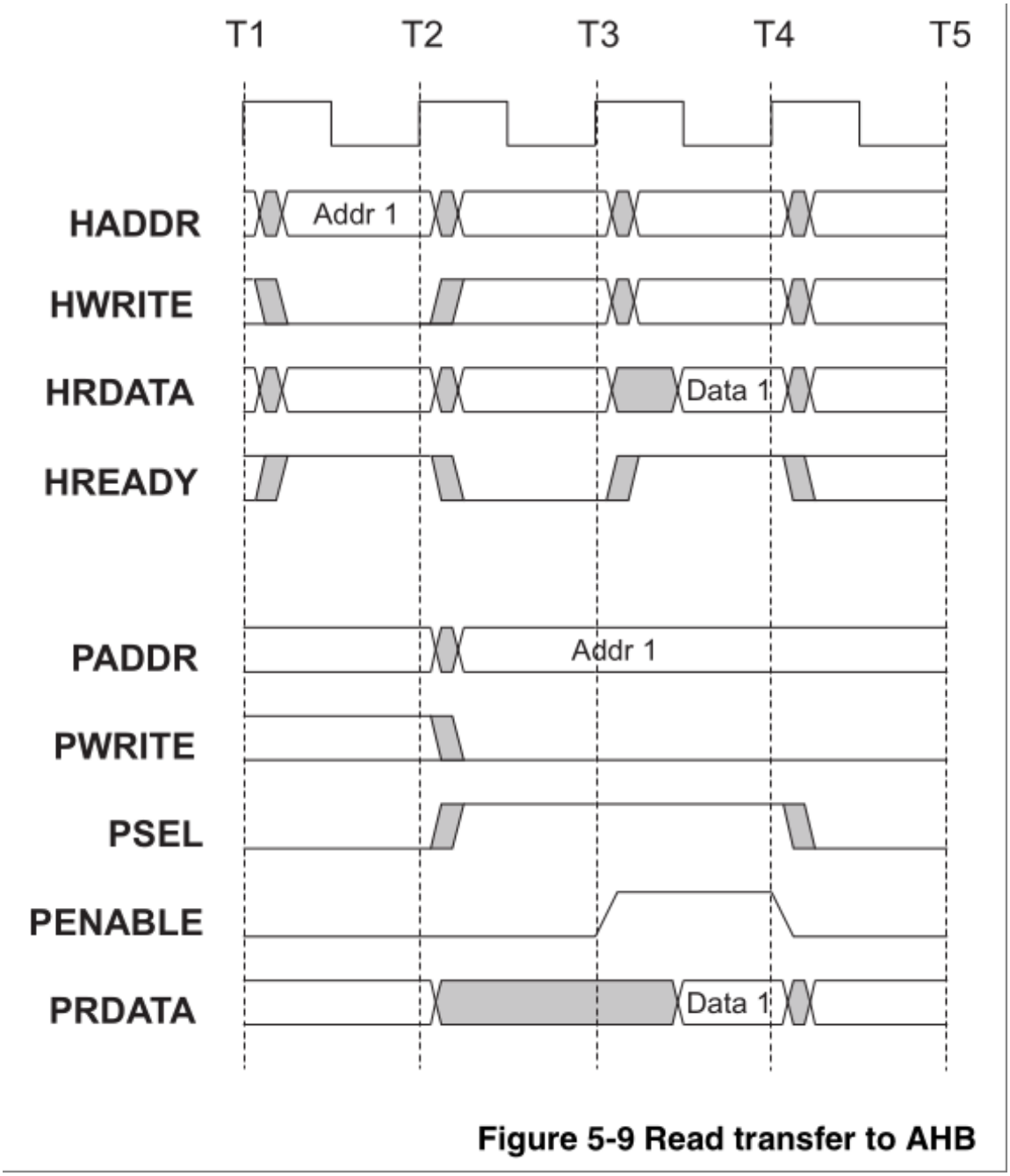


Figure 3: Single Read of the AHB-to-APB Bridge

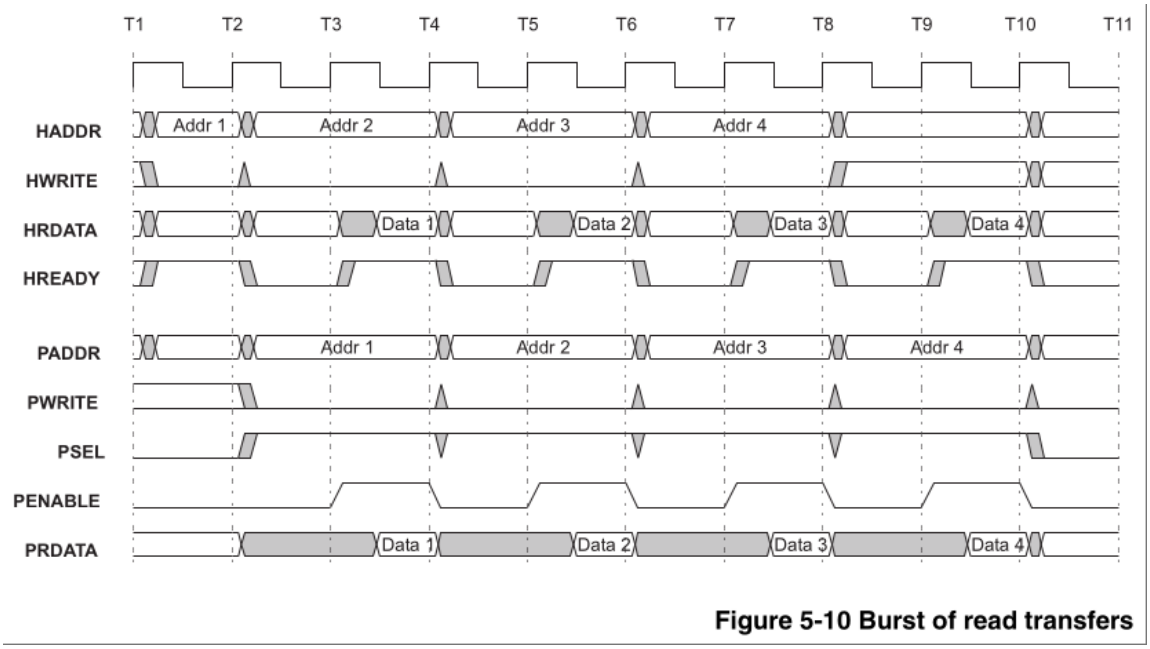


Figure 4: Burst Read of the AHB-to-APB Bridge

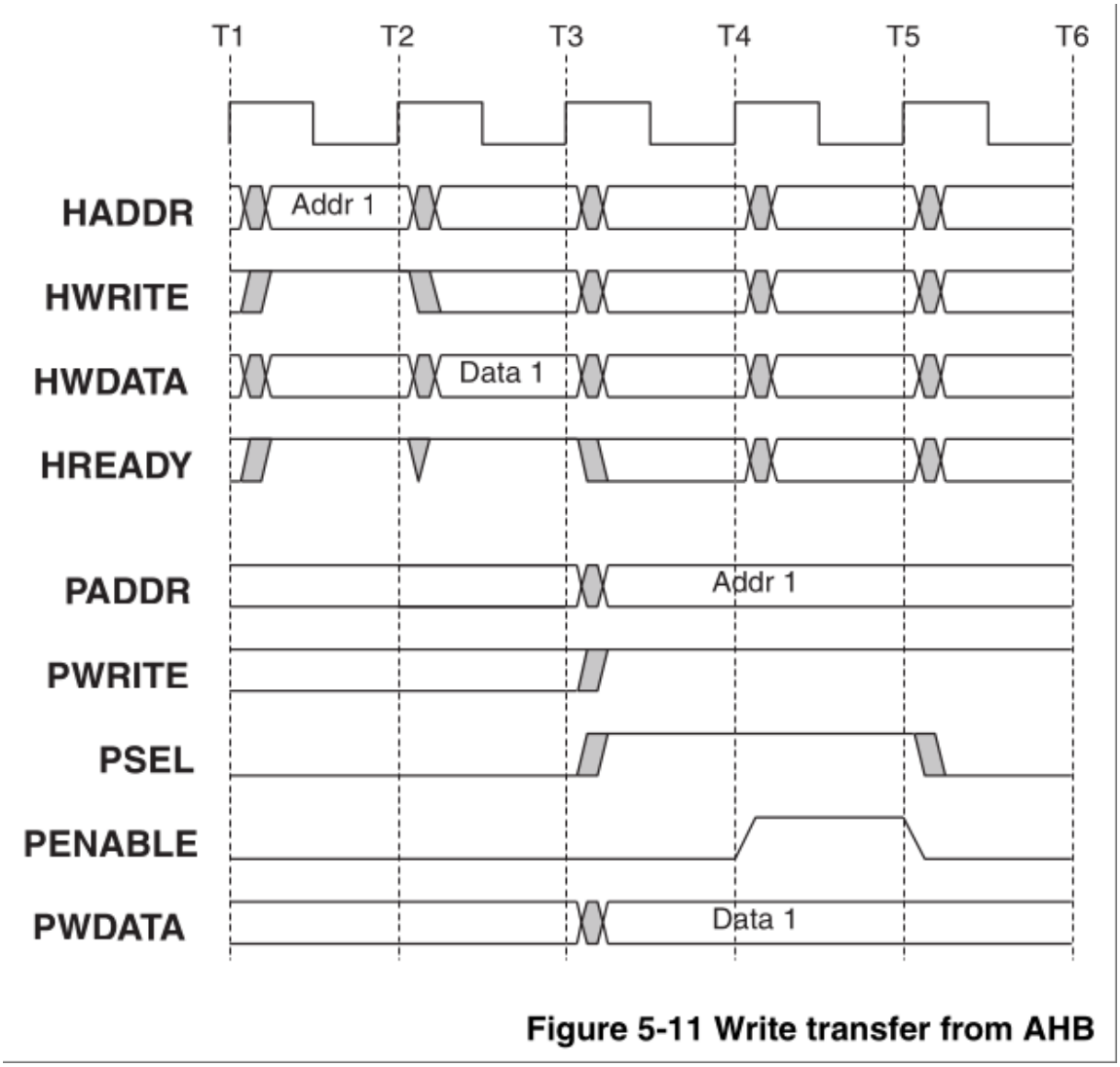


Figure 5: Single Write of the AHB-to-APB Bridge

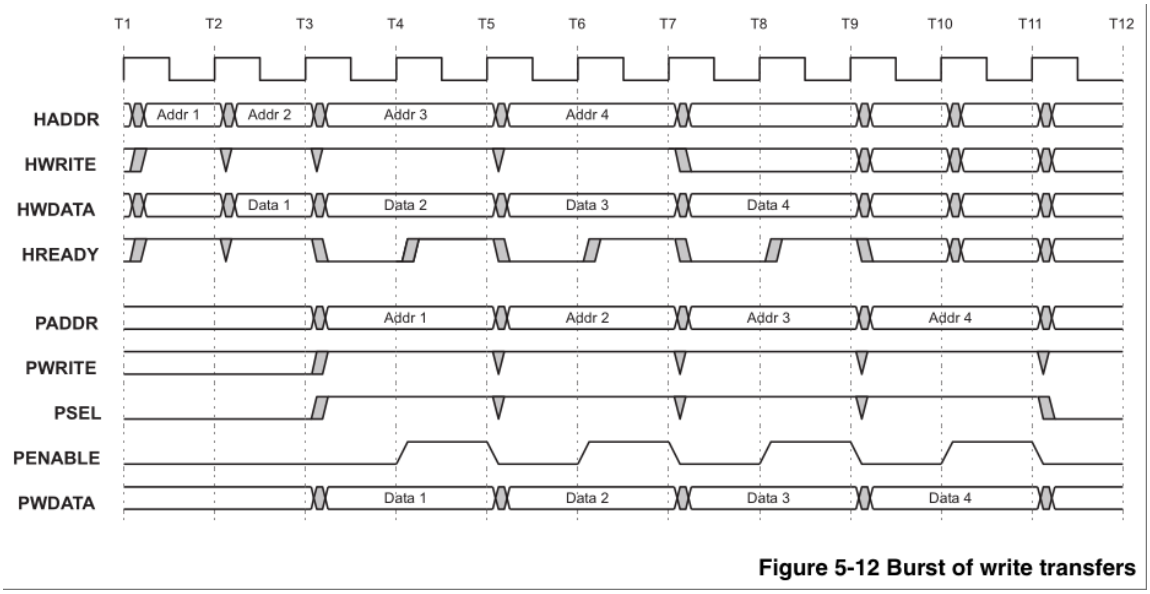


Figure 6: Burst Write of the AHB-to-APB Bridge

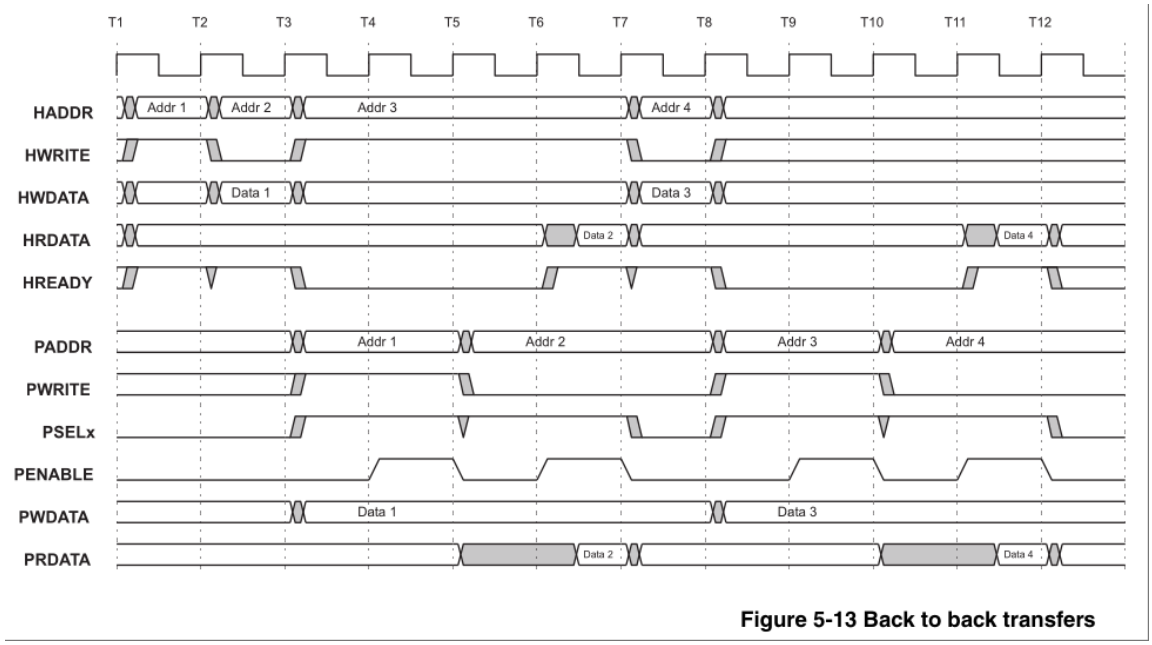


Figure 7: Back-to-Back Transfers of the AHB-to-APB Bridge

6.1 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
```

As outlined in the **AHB_Interface.v** section above, the AHB slave interface operates as an I/O module, making simulation-based testbenches sufficient for verifying its input and output behavior. The testbench is designed to assess 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. Additionally, the testbench ensures the proper functionality of the active-low reset mechanism. The primary verification objectives include validating data transfers, ensuring correct address handling, and detecting invalid inputs. The results are explained in the **Testbench Results of AHB_Interface.v** section below.

6.2 FPV for APB_FSM.v

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

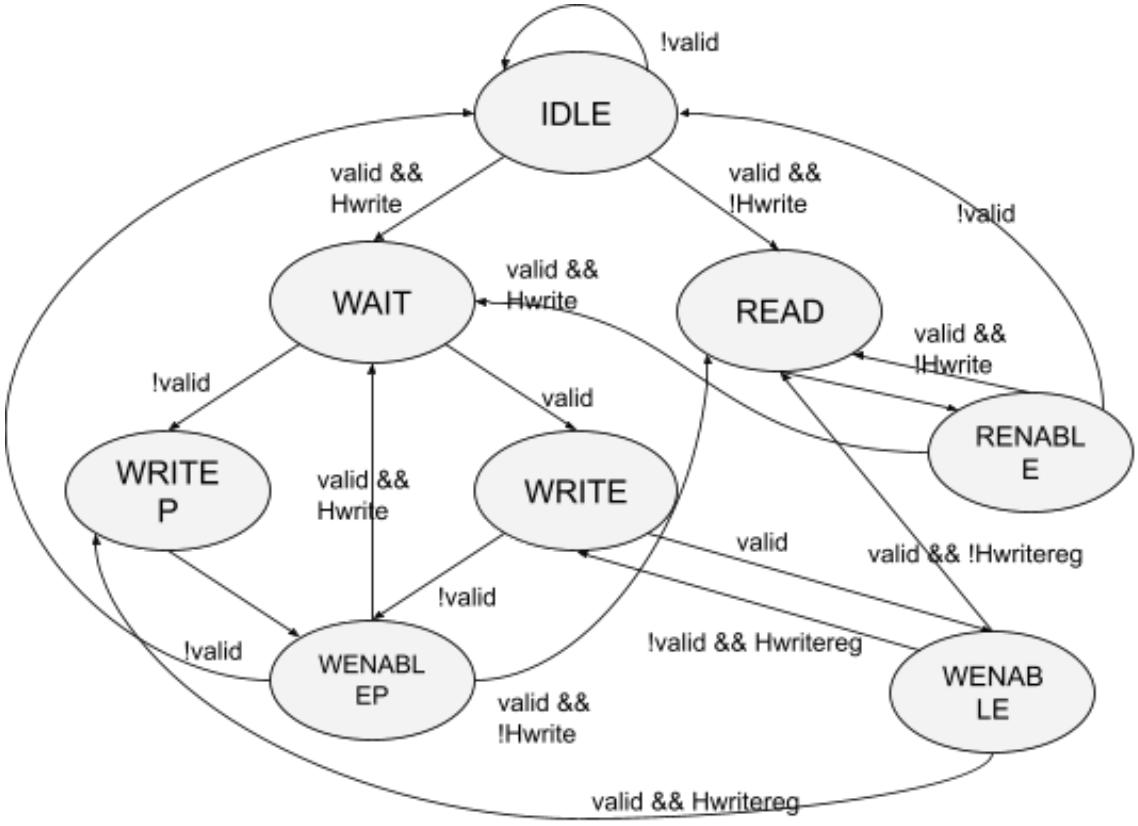


Figure 8: FSM State Transitions of the APB Controller

The FPV code `APB_FSM_sva.sv` using SystemVerilog Assertions (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(.*);
```

Based on the source code in the **APB_FSM.v** section above, SVAs are employed to verify the correctness of state transitions. By using each `if` and `else` condition provided in the source code 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.

6.3 FPV for top.v

The FPV code `top_sva.sv` using SystemVerilog Assertions and Assumptions 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, as mentioned in **AHB_Master.v** section above.

The top-level design (can refer to the source code in the **top.v** section above) 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 either Non-sequential `2'b10` or Sequential `2'b11`. The design does not support `IDLE` or `BUSY` 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.

6.4 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.

7 Results

7.1 Testbench Results of AHB_Interface.v

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

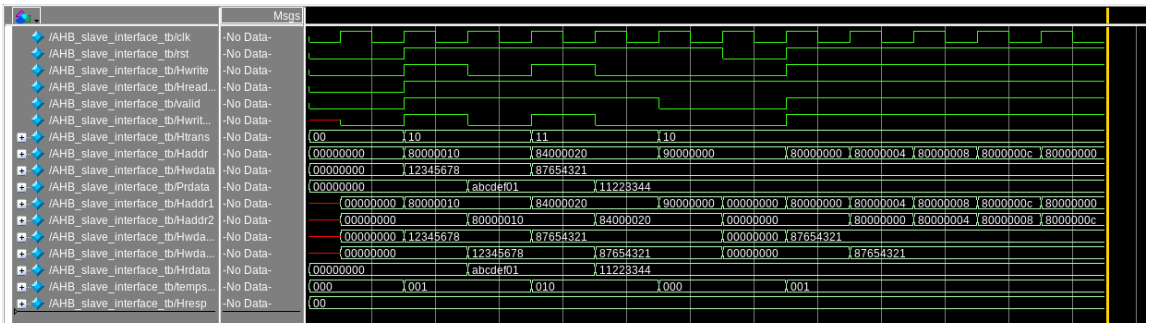


Figure 9: ModelSim Testbench Wave Results of `AHB_Slave_Interface_tb.v`

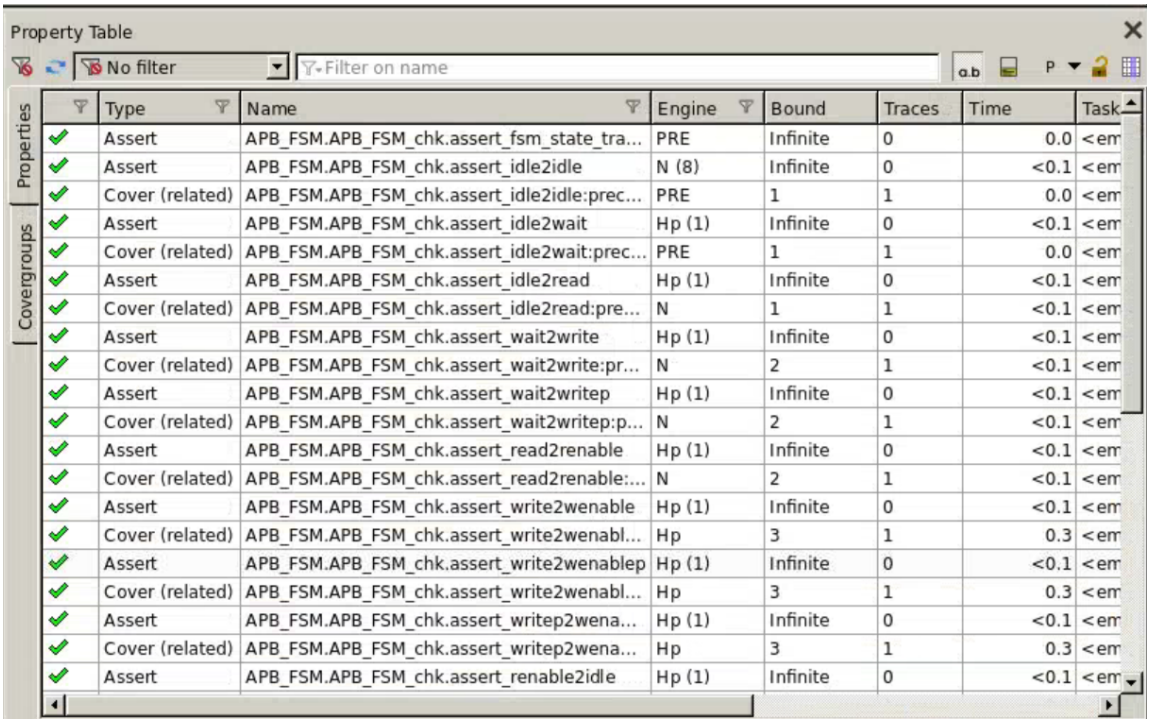
```
# Starting Read-Write-Read-Write Test
# Read-Write-Read-Write Test Completed
# Starting Invalid Address Test
# Invalid Address Test Passed
# Starting Reset Test
# Reset Test Passed
# Starting Burst Test
# Burst Test Completed
# ** Note: $stop    : AHB_Slave_Interface_tb.v(128)
#    Time: 125 ns  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 conditions when `rst` is toggled to low.

7.2 FPV Results of APB_FSM.v

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



Property Table

No filter | Filter on name

Properties | Covergroups

	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.

7.3 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`

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

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

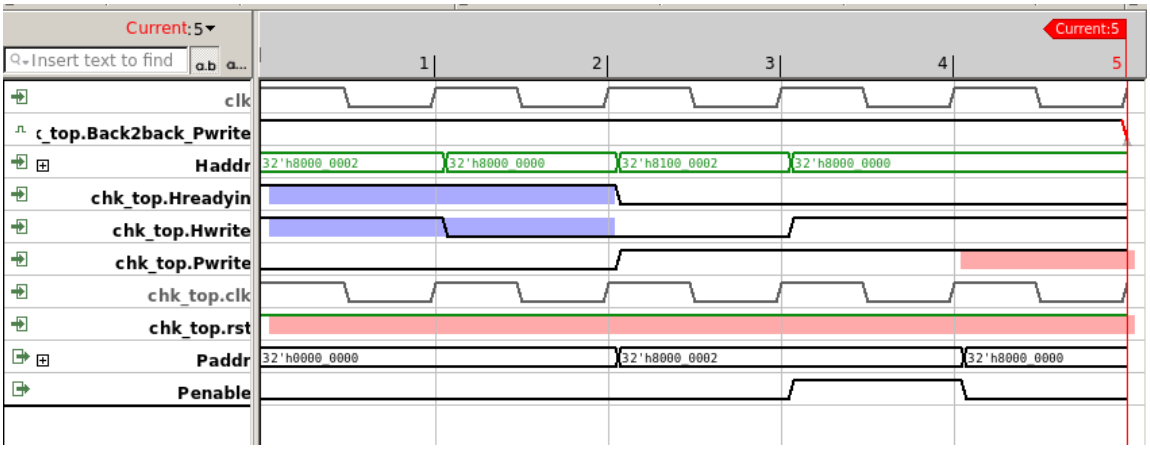


Figure 15: Jasper Wave Results of `Back2back_Pwrite` Assertion

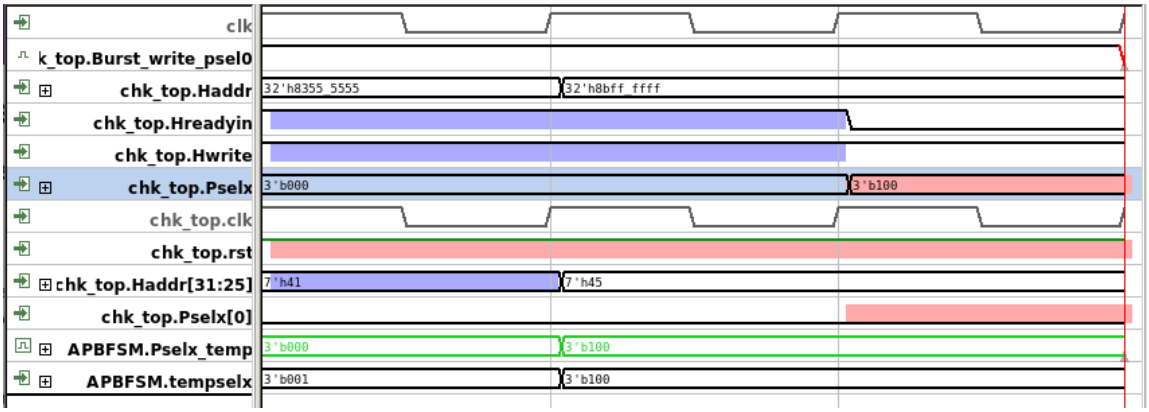


Figure 16: Jasper Wave Results of `Burst_write_psel0` Assertion

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 **APB_FSM.v** code, `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

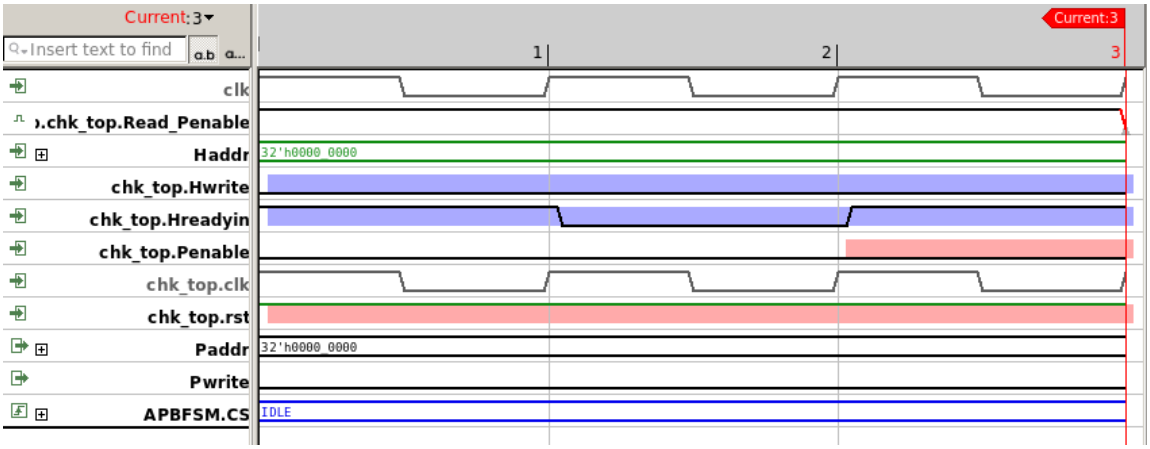


Figure 18: Jasper Wave Results of `Read_Penable` Assertion

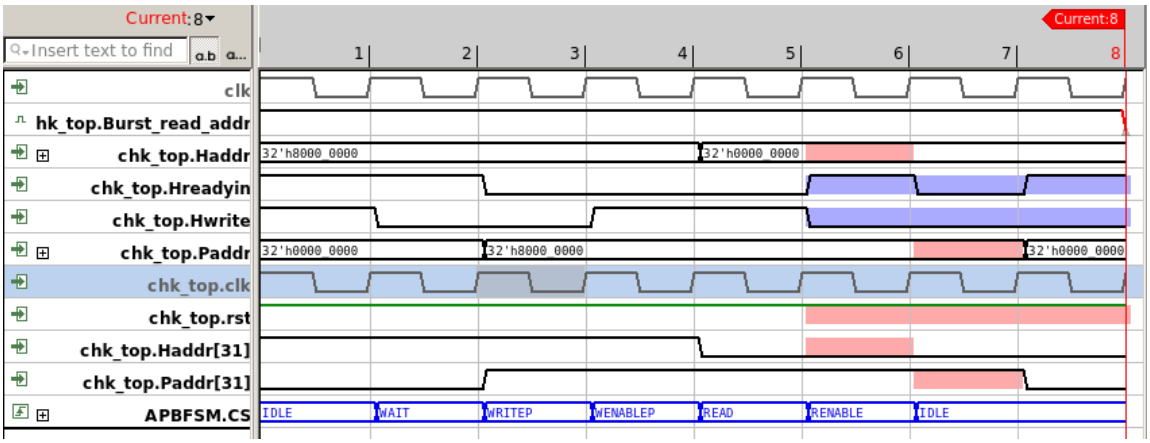


Figure 19: Jasper Wave Results of `Burst_read_addr` Assertion

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

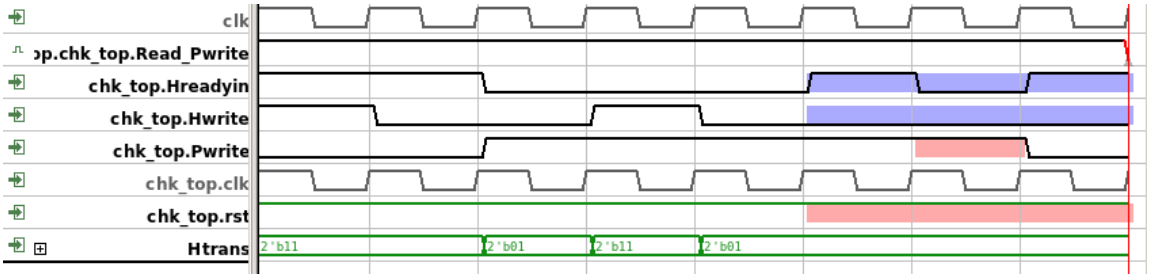


Figure 21: Jasper Wave Results of `Read_Pwrite` Assertion

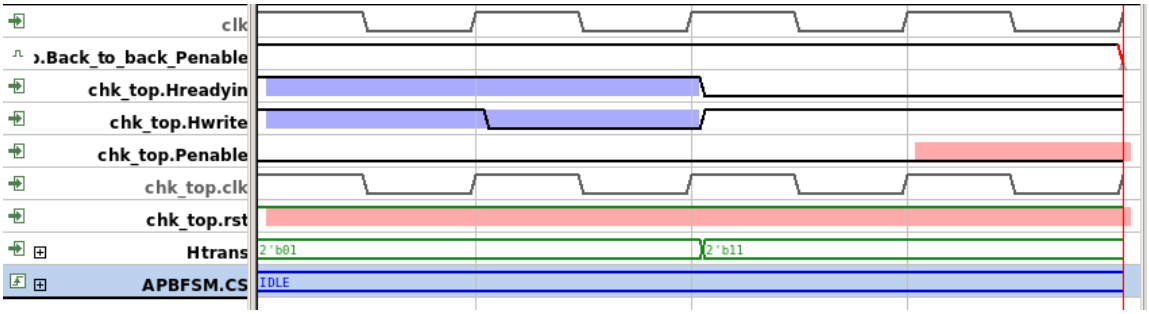


Figure 22: Jasper Wave Results of `Back_to_back_Penable` Assertion

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].

7.4 Testbench Results of top.v

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

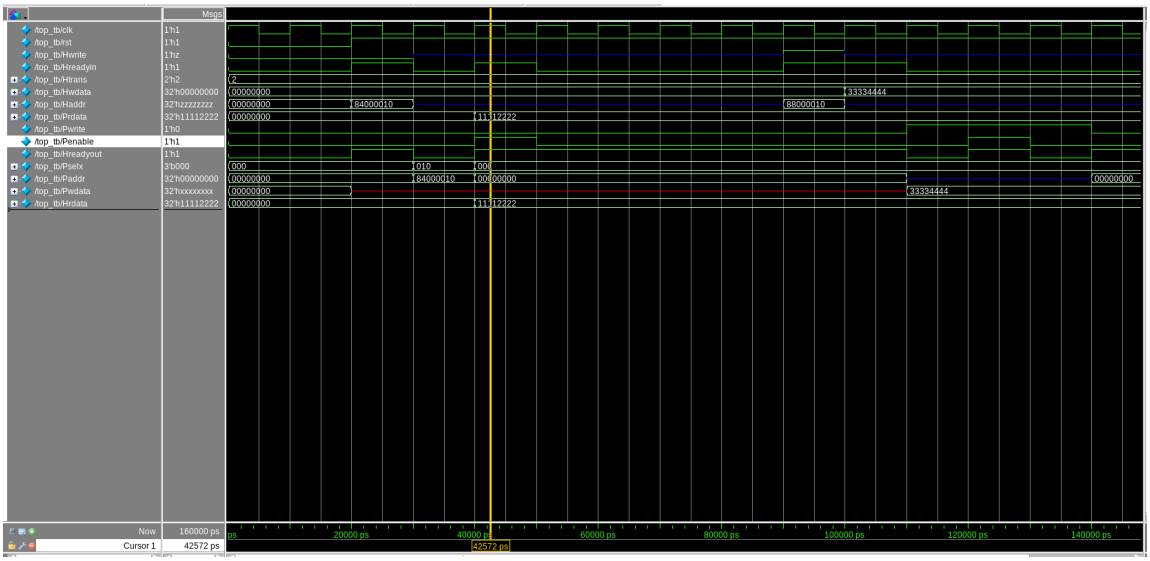


Figure 23: ModelSim Testbench Wave Results of `top_tb.v`

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.

7.5 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.

8 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] ——, "Formal verification — Wikipedia, the free encyclopedia," 2024, [Online; accessed 22-December-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Formal_verification&oldid=1258345258

[4] I. Synopsys, "Vc formal: Static and formal verification," accessed: 2024-12-21. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html#fpv

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