AMBA APB Protocol Verification Using System Verilog

By Anubhav Agarwal

<u>What is AMBA APB:-</u>

- AMBA = Advanced Microcontroller Bus Architecture
- The first version of AMBA APB(APB 1.0) was developed in 1996 by ARM Ltd for SoC designs., and the current version in use is Version 2.0, developed in 2010.
- Provides standard on-chip communication
- Bus families include:
  - AXI (high performance)
  - AHB (medium, pipelined)
  - APB (low power, simple, non-pipelined)
- APB is used mainly for peripheral register access

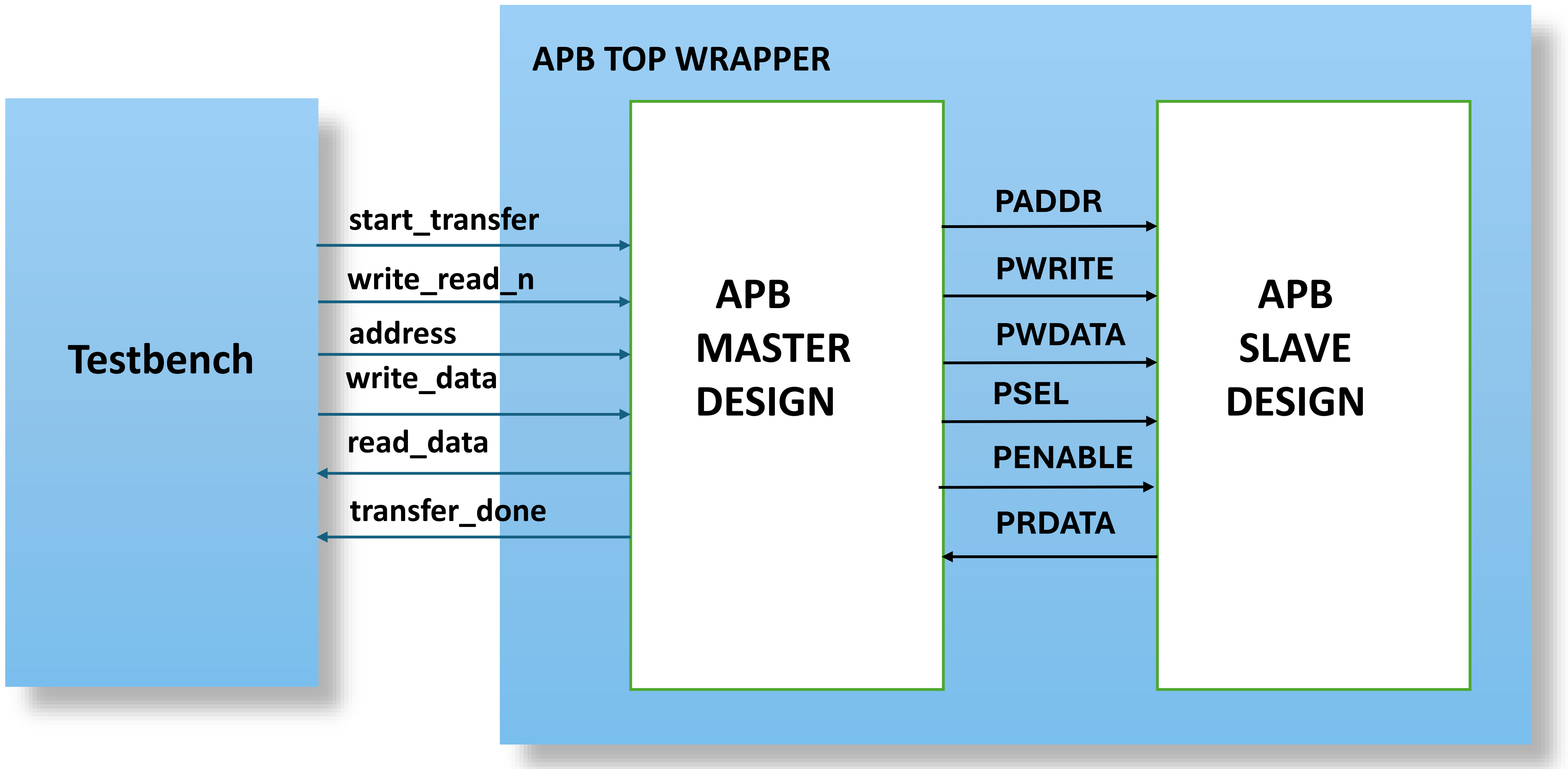
Signals Used in AMBA APB Design:

Signal Name	Width	Direction	Description
PCLK	1-bit	Input	APB Clock - Rising edge activate
PRESETn	1-bit	Input	APB Reset – Activate Low
PADDR	32-bit	(Master → Slave)	Address Bus
PSEL	1-bit	(Master → Slave)	Select signal for Slave
PENABLE	1-bit	(Master → Slave)	Enable signal to transfer
PWRITE	1-bit	(Master → Slave)	Write Enable(1 = Write, 0 = Read)
PWDATA	32-bit	(Master → Slave)	Write Data Bus
PRDATA	32-bit	(Slave → Master)	Read Data Bus
PREADY	1-bit	(Slave → Master)	Ready signal from Slave
PSLVERR	1-bit	(Slave → Master)	Slave Error signal

Additional Controller/Top-Level Signals:

Signal Name	Width	Direction	Description
start_transfer	1-bit	(TB → Master)	Triggers APB read/write operation.
write_read_n	1-bit	(TB → Master)	Read Write operation(1 = W, 0 = R)
address	32-bit	(TB → Master)	Target Address for APB Transfer
write_data	32-bit	(TB → Master)	Data to write during write transactions.
read_data	32-bit	(Master → TB)	Read data returned to testbench.
transfer_done	1-bit	(Master → TB)	Indicates completion of APB transaction.

How I Design AMBA APB:-



Breakdown of the AMBA APB Design:

I have divided the AMBA APB Design Into three parts.

1. APB TOP WRAPPER.
2. APB MASTER Design
3. APB SLAVE Design

Role of the APB Top Wrapper:

- In this code Testbench only Connect to APB TOP WRAPPER.
- APB TOP WRAPPER work as bridge between testbench and internal APB components.
- APB Master and Slave both take input and give output from APB TOP WRAPPER.

A. **APB MASTER DESIGN:** I break down design of APB Master in parts.

1. **Port List and Signal Declaration:** This part defines all ports used by the master.

```
module apb_master #(
  parameter ADDR_WIDTH = 32,
  parameter DATA_WIDTH = 32
) (
  input PCLK,
  input PRESETn,
  // Control Interface
  input start_transfer,
  input write_read_n,
  input [ADDR_WIDTH -1 :0] address,
  input [DATA_WIDTH -1 :0] write_data,
  output reg [DATA_WIDTH -1 :0] read_data,
  output reg transfer_done,
```

```
 //APB Interface
  output reg PSEL,
  output reg PENABLE,
  output reg [ADDR_WIDTH -1 :0]
PADDR,
  output reg [DATA_WIDTH -1 :0]
PWDATA,
  output reg PWRITE,
  input  PREADY,
  input  PSLVERR,
  input  [DATA_WIDTH -1 :0] PRDATA
);
```

Port List and Signal Declaration Explanation:-

- In this part, I am just declaring all the ports that my APB Master will use.
- I create parameters ADDR_WIDTH and DATA_WIDTH in module. So I can change the address or data width anytime.
- All the basic control signals and APB interface signals are listed here before I start writing the main logic.
- This section basically sets up the input/output structure for the whole design, so the next parts become easier to understand.

2. State Declaration and Current State Logic:

This part updates current state according to PCLK clock posedge.

```
// States defined parameters
 parameter IDLE = 2'b00, SETUP = 2'b01,
ACCESS = 2'b10;


 //State Variables
 reg [1:0] c_state;
 reg [1:0] n_state;
```

```
// Current State Logic - Sequential
Circuit
 always @(posedge PCLK or negedge
PRESETn) begin
   if (PRESETn == 0) begin// !reset_n
     c_state <= IDLE;
     PWDATA <= 0;
   end
   else
     c_state <= n_state;
 end
```

State Declaration and Current State Logic Explanation:

- In this part, I declared all the states of my APB Master (IDLE, SETUP, ACCESS).
- I also created two variables: one for the current state and one for the next state.
- The sequential always block updates the current state on every clock edge.
- Whenever reset is low, the master goes back to the IDLE state.
- Otherwise, c_state simply takes the value of n_state in every cycle.
- This block basically handles how the APB Master moves from one state to another.

3. **FSM Next State Logic(n_state):**

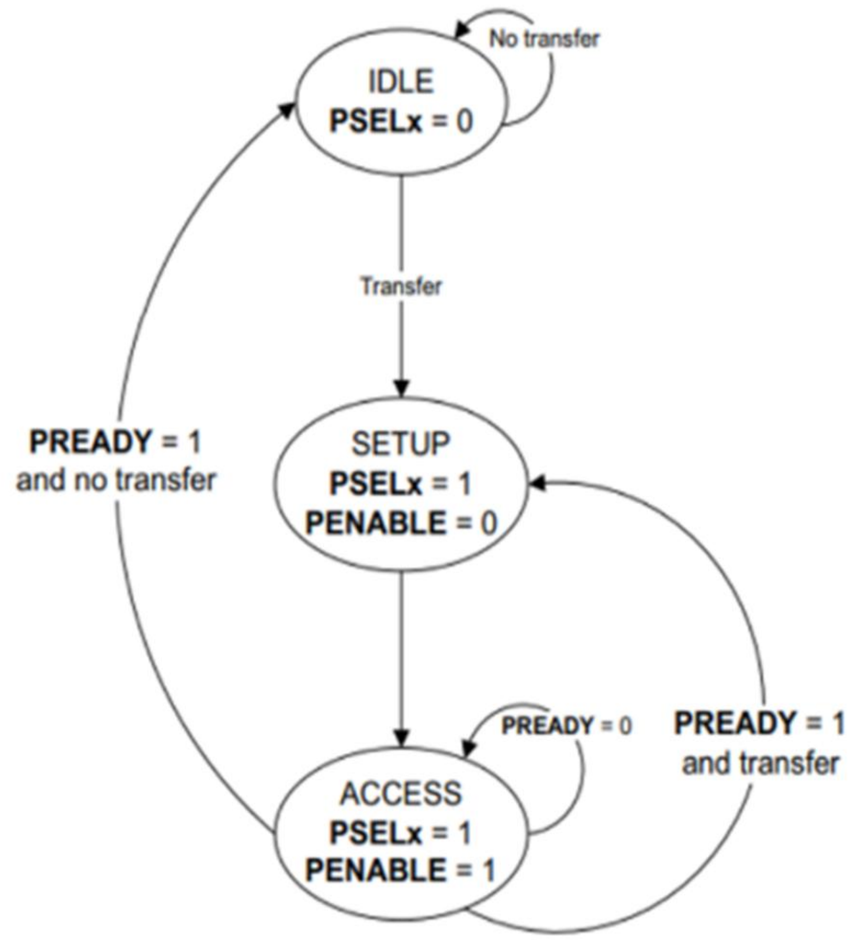
This part decides next state based on conditions.”

```
// Next State Logic - Combinational Circuit
 always @(*) begin
   n_state = c_state;
   case (c_state)
     IDLE: begin
       if (start_transfer)
         n_state = SETUP;
     end
     SETUP: begin
       n_state = ACCESS;
     end
```

```
ACCESS: begin
       if (PREADY)
         n_state = IDLE;
     end
     default: n_state = IDLE;
   endcase
 end
```

How FSM Of AMBA APB Look Like:

1. IDLE State. Default state when no transfer is required. PSEL = 0, PENABLE = 0. No slave is selected
2. SETUP State. Master drives address, write data, and controlsignals. PSEL = 1, PENABLE = 0. Slave is selected but transfer hasn't started. Lasts exactly one clock cycle
3. ACCESS State. Transfer is in progress. PSEL = 1, PENABLE = 1. Slave can extend transfer using PREADY. Transfer completes when PREADY = 1APB Finite State Machine



FSM Next State Logic(n_state) Explanation:

- In this part, I am deciding what the next state should be based on the current state and inputs.
- This is a combinational block, so it updates instantly whenever there is any change.
- From IDLE, the FSM moves to SETUP only when a transfer starts.
- From SETUP, it directly goes to ACCESS in the next cycle.
- In ACCESS, the FSM waits for the slave to assert PREADY, and then it returns back to IDLE.
- So basically, this logic controls the complete flow of the APB transfer step-by-step.

4. Output Logic and Read Write Logic:

This part drives all APB outputs and handles read-data.

```
 // Output Logic - Combinational +
Sequential Circuit
  always @(*) begin
    PSEL = (c_state != IDLE);
    PENABLE = (c_state == ACCESS);
    PWRITE = write_read_n;
    PADDR = address;
    PWDATA = write_data;
    transfer_done = (c_state == ACCESS) &&
PREADY;
  end
```

```
 // Read data capture
  always @(posedge PCLK) begin
    if (transfer_done && !PWRITE)
      read_data <= PRDATA;
  end
endmodule
```

Output Logic and Read Write Logic Explanation:

- In this part, I am generating all the APB output signals using the current state.
- PSEL becomes high whenever the master is not in the IDLE state.
- PENABLE is only high during the ACCESS state.
- The PWRITE signal decides whether the transfer is write or read(1 = W, 0 = R).
- During a write transfer, the master places the address on PADDR and write data on PWDATA.
- For a read transfer, the master still sends the address, but PWRITE is low. So The slave responds by placing the read value on PRDATA.
- transfer_done becomes high when the master is in ACCESS and the slave asserts PREADY.

B. <u>APB SLAVE DESIGN:</u> I break down design of APB Slave in parts.

1. <u>Port List and Signal Declaration</u>:

```
module apb_slave #(
  parameter ADDR_WIDTH = 32,
  parameter DATA_WIDTH = 32
) (
  input PCLK,
  input PRESETn,
  //APB Interface
  input PSEL,
  input PENABLE,
  input PWRITE,
  input [ADDR_WIDTH -1 :0] PADDR,
  input [DATA_WIDTH -1 :0] PWDATA,
```

```
output PSLVERR,
  output PREADY,
  output [DATA_WIDTH -1 :0]
PRDATA
);

  integer i;
  integer j = 0;
  // Internal registers
  reg [DATA_WIDTH -1 :0] memory
[0:15]; // 16 registers
```

Port List and Signal Declaration Explanation:

- In this part, I defined all the ports that my APB Slave will use.
- I create parameters ADDR_WIDTH and DATA_WIDTH in module. So I can change the address or data width anytime.
- The APB interface signals like PSEL, PENABLE, PWRITE, PADDR, and PWDATA come from the Master.
- The Slave sends responses back using PSLVERR, PREADY, and PRDATA.
- I also created internal registers (memory array) to store data inside the slave.
- These registers work like a small RAM, where each address stores one data value.
- Overall, this section sets up the complete interface and memory required for the slave before writing the main logic.

2. Address and Write operation Logic:

```
//Address decode
  wire [3:0] reg_addr;
  assign reg_addr = PADDR[5:2]; // Word
addressing
```

```
 // Write Operation
  always @(posedge PCLK or negedge
PRESETn) begin
    if (!PRESETn) begin
      for (i = 0; i < 16; i=i+1)
        memory[i] <= 32'h0;
    end
    else if(PSEL && PENABLE &&
PWRITE && PREADY) begin
      memory[reg_addr] <= PWDATA;
      j = j+1;
    end
  end
```

Address and Write operation Logic Explanation:

- In this part, I am doing write operation.
- I am using only bits [5:2] of PADDR for word addressing, which selects one of the 16 memory locations inside the slave.
- On reset, all 16 memory registers are cleared to zero using a simple loop when PRESETn = 0.
- During a valid write cycle (PSEL = 1, PENABLE = 1, PWRITE = 1, and PREADY = 1), the data from PWDATA is written into the selected memory address.
- I also increment a counter 'j' just for tracking how many writes have happened (optional).
- So overall, this block handles both the address decoding and the actual write operation into the slave memory.

3. Output and Read operation Logic:

```
 // Output Logic
  assign PRDATA = (PSEL && !PWRITE) ? memory[reg_addr] : 32'h0;
  assign PREADY = 1'b1; // Np wait state
  assign PSLVERR = 1'b0; // No error
endmodule
```

Output and Read operation Logic Explanation:

- In this part, I am generating the APB Slave output signals.
- For read operation, when PSEL is high and PWRITE is low, the slave drives the data from the selected memory location onto PRDATA.
- Otherwise, PRDATA remains zero, which means no read is happening.
- PREADY is tied to 1, so the slave always responds in a single cycle (no wait states in my design).
- PSLVERR is always 0, meaning the slave never reports an error.
- So overall, this block handles sending read data back to the master and provides the standard APB response signals.

C. **APB TOP WRAPPER:** I break down design of APB Top in parts.

1. **Port List and TestBench Signal Declaration**:

```
module abp_top #(
  parameter ADDR_WIDTH = 32,
  parameter DATA_WIDTH = 32
) (
  input PCLK,
  input PRESETn,
  //Controller Interface
  input start_transfer,
  input write_read_n,
  input [ADDR_WIDTH -1 :0] address,
  input [DATA_WIDTH -1 :0] write_data,
  output reg [DATA_WIDTH -1 :0] read_data,
  output reg transfer_done
);
```

```
 wire [ADDR_WIDTH -1 :0] padder;
  wire[DATA_WIDTH -1 :0] pwdata,
prdata;
  wire psel, penable, pwrite,
pready, pslverr;
```

2. APB Master Instantiate And Signal Declaration:

```
apb_master #(
   .ADDR_WIDTH(ADDR_WIDTH),
   .DATA_WIDTH(DATA_WIDTH)
 ) inst_apb_master (
   .PCLK(PCLK),
   .PRESETn(PRESETn),
   //Controller Interface
   .start_transfer(start_transfer),
   .write_read_n(write_read_n),
   .address(address),
   .write_data(write_data),
   .read_data(read_data),
   .transfer_done(transfer_done),
```

```
//APB Interface
   .PSEL(psel),
   .PENABLE(penable),
   .PADDR(padder),
   .PWDATA(pwdata),
   .PWRITE(pwrite),
   .PREADY(pready),
   .PSLVERR(pslverr),
   .PRDATA(prdata)
 );
```

3. APB Slave Instantiate And Signal Declaration:

```
apb_slave #(
    .ADDR_WIDTH(ADDR_WIDTH),
    .DATA_WIDTH(DATA_WIDTH)
  ) inst_apb_slave (
    .PCLK(PCLK),
    .PRESETn(PRESETn),
```

```
//APB Interface
   .PSEL(psel),
   .PENABLE(penable),
   .PADDR(padder),
   .PWDATA(pwdata),
   .PWRITE(pwrite),
   .PREADY(pready),
   .PSLVERR(pslverr),
   .PRDATA(prdata)
 );
```

APB TOP WRAPPER Explanation:

- In this part, I created the top-level module that connects the APB Master and APB Slave together.
- First, I declared all the ports that come from the testbench, such as clock, reset, start signal, address, write data, etc.
- Then, I created internal wires for the APB bus signals like PADDR, PWDATA, PRDATA, PSEL, PENABLE, PWRITE, PREADY, and PSLVERR.
- After that, I instantiated the APB Master and connected all its ports to the top-level signals.
- Similarly, I instantiated the APB Slave and connected its ports to the same APB bus wires.
- Because both the Master and Slave share these wires, the data flow between them happens automatically according to the APB protocol.
- So overall, this top wrapper simply ties everything together and allows the testbench to drive the master, and the master to communicate with the slave through the APB bus.

C. **APB Testbench:** I break down design of APB Testbench in parts.

1. **Signal Declaration and ABP Top Module Instantiate :**

```
module tb_apb_top();
  parameter ADDR_WIDTH = 32;
  parameter DATA_WIDTH = 32;
  reg PCLK;
  reg PRESETn;
  //Controller Interface
  reg start_transfer;
  reg write_read_n;
  reg [ADDR_WIDTH -1 :0] address;
  reg [DATA_WIDTH -1 :0] write_data;
  wire [DATA_WIDTH -1 :0] read_data;
  wire transfer_done;
```

```
abp_top #(
   .ADDR_WIDTH(ADDR_WIDTH),
   .DATA_WIDTH(DATA_WIDTH)
 ) inst_apb_top (
   .PCLK(PCLK),
   .PRESETn(PRESETn),
   .start_transfer(start_transfer),
   .write_read_n(write_read_n),
   .address(address),
   .write_data(write_data),
   .read_data(read_data),
   .transfer_done(transfer_done)
 );
```

2. Starting Process and Clock Genration:

```
initial begin
   $display($time, " | --------------------Simulation is Started--------------------");
   PCLK = 1'b0;
   $display($time, " | --------------------Set 1'b0 to PRESETn--------------------");
   PRESETn = 1'b0; // Initially, PRESETn is Active
   #1000 $finish;
 end
 //Clock Genrator- TimePeriod = 10ns
 always #5 PCLK = ~PCLK;
```

3. Task for Write and Read Process:

```
 task write_transfer(output start, write,
output [31:0] addr, wdata);
    begin
      start = 1;
      write = 1;
      addr = 32'h 00000030;
      wdata = 32'hABABABAB;
    end
  endtask
```

```
 task read_transfer(output start, write,
output [31:0] addr);
    begin
      start = 1;
      write = 0;
      addr = 32'h 00000030;
    end
  endtask
  initial begin
```

4. Start Testing AMBA APB Design:

```
 initial begin
    $monitor($time, " | start_transfer = %b | write_read_n = %b | address = %h |
write_data = %h , read_data = %h", start_transfer, write_read_n, address, write_data,
read_data);

    repeat(3) @(posedge PCLK);
    $display($time, " | --------------------Set 1'b1 to PRESETn--------------------");
    PRESETn = 1'b1;

    $display($time, " | ------------------APB WRITE Transfer is Started------------------");
    write_transfer(start_transfer, write_read_n, address, write_data);
    wait(transfer_done);
    $display($time, " | ------------------APB WRITE Transfer is Completed------------------");
```

```
•  repeat(3) @(posedge PCLK);
•    $display($time, " | --------------------APB read Transfer is Started----------------------");
•    read_transfer(start_transfer, write_read_n, address);
•    repeat(2) @(posedge PCLK);
•    $display($time, " | start_transfer = %b | write_read_n = %b | address = %h |
   read_data = %h", start_transfer, write_read_n, address, read_data);
•    wait(transfer_done);
•    $display($time, " | --------------------APB read Transfer is Completed--------------------");
•   end
•   initial begin
•    $dumpfile("Apb.vcd");
•    $dumpvars(0, tb_apb_top);
•   end
• endmodule
```

APB Testbench Explanation:

- I created the APB testbench to verify the complete APB design.
- Then I instantiated the APB Top Wrapper, which connects the APB Master and APB Slave together.
- After that, I wrote an initial block to start the simulation, print messages, set the reset signal, and then release it after a few clock cycles.
- I also created a simple clock generator that toggles PCLK every 5 ns, giving me a 10 ns is complete clock period.
- To make the testbench cleaner, I used tasks for write and read transfers. Each task sets the start signal, operation type (read/write), and the address/data values.

- In the main testing block, I first perform a write transfer, wait until it completes, and print the status.
- Then I perform a read transfer from the same address and wait again until the read is completed.
- I used $monitor and $display to print the signal values during simulation so I can easily track what’s happening.
- Finally, I added $dumpfile and $dumpvars to generate the waveform file so I can view the complete operation in GTKWave.
- Overall, this testbench fully checks both write and read operations and verifies that the complete APB design is working correctly.

Verification of AMBA APB Design:

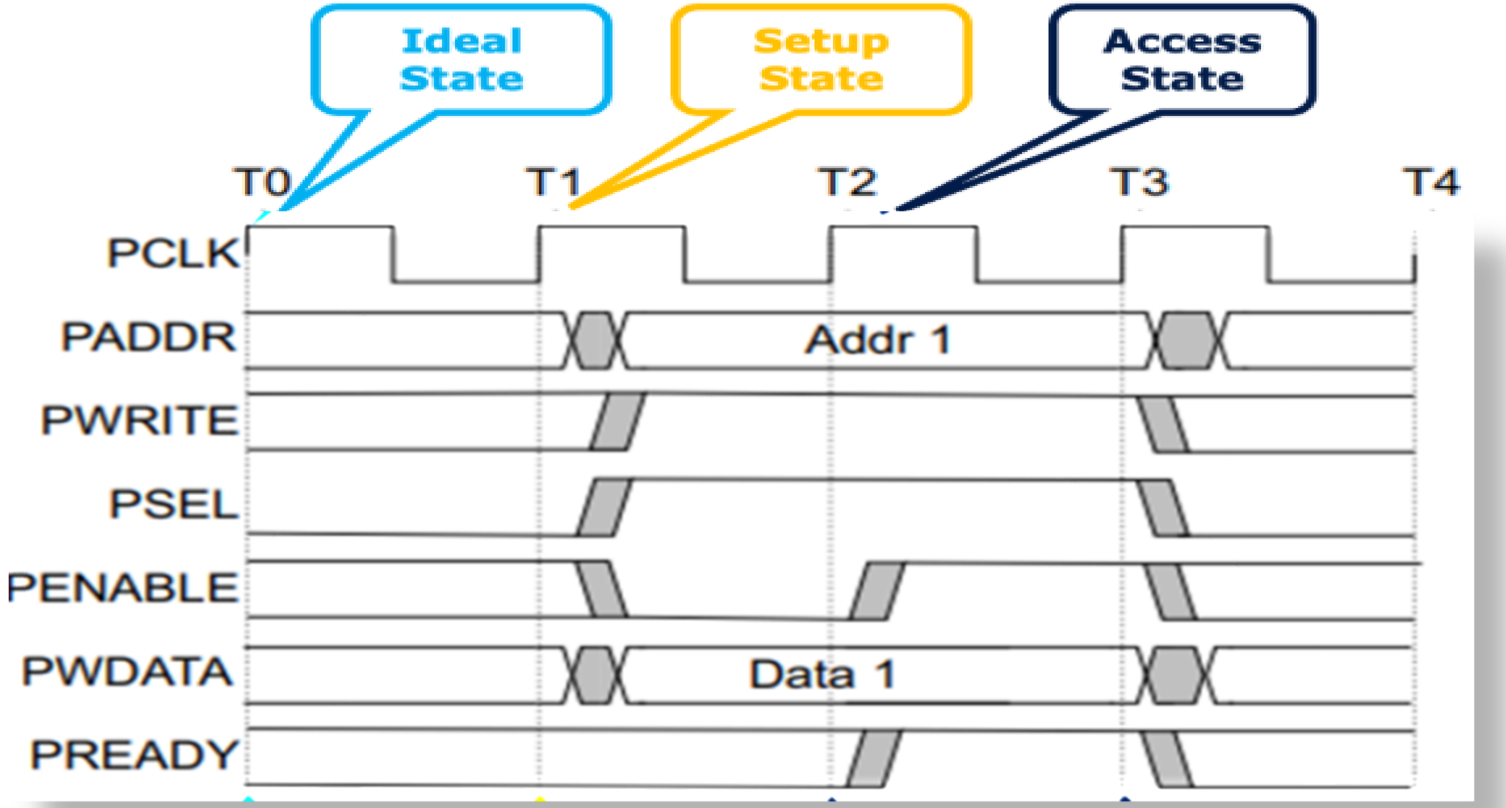
1. APB WRITE TRANSFER (Without Wait State):

SETUP Phase

- PSEL = 1
- PENABLE = 0
- PADDR & PWRITE & PWDATA must be stable

ACCESS Phase

- PSEL = 1
- PENABLE = 1
- Data written when PREADY = 1
- Transfer ends at rising edge when PREADY=1 & PENABLE=1



2. APB READ TRANSFER (Without Wait State):

SETUP Phase

•PSEL = 1

•PENABLE = 0

•PADDR valid

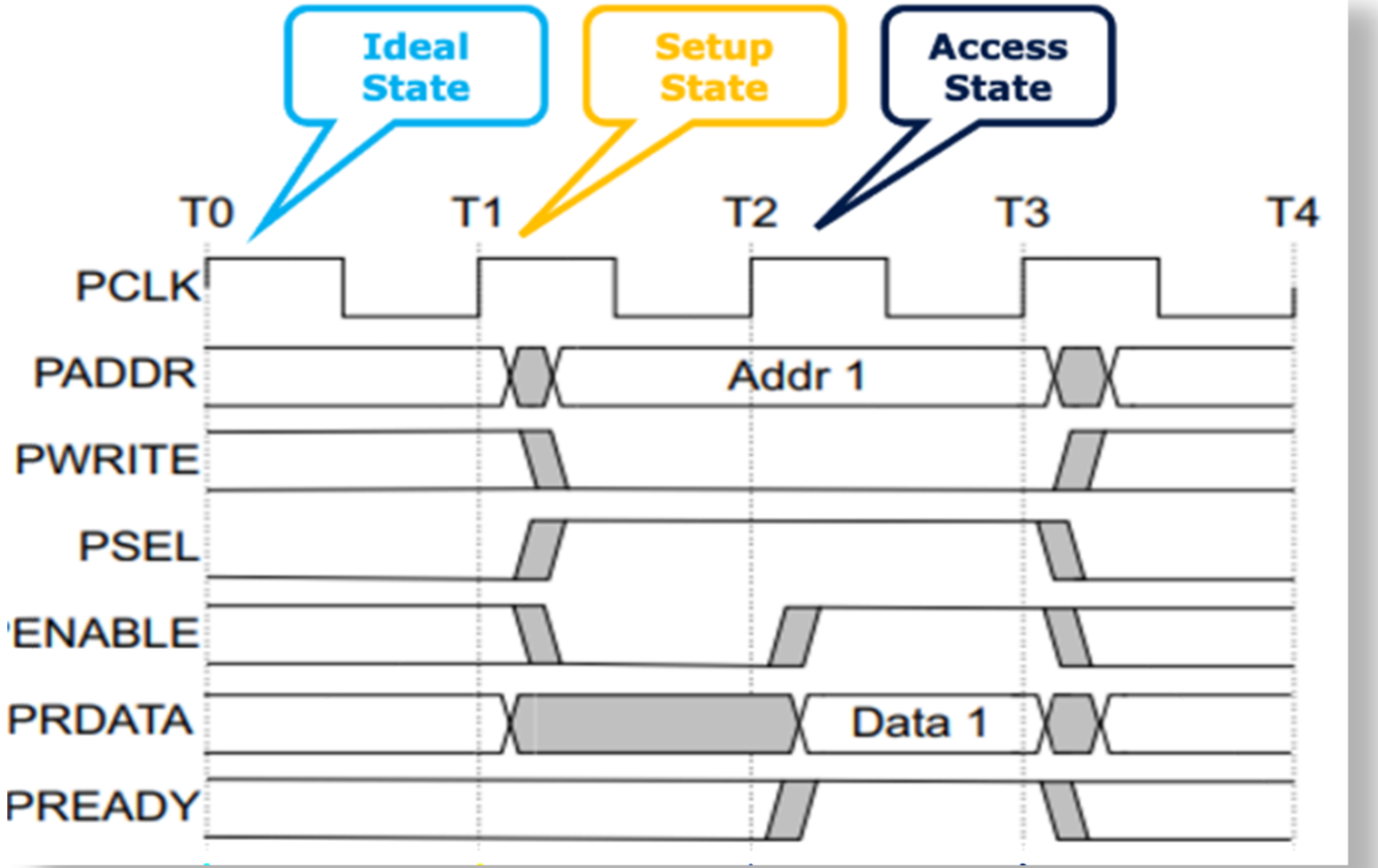
•PWRITE = 0

ACCESS Phase

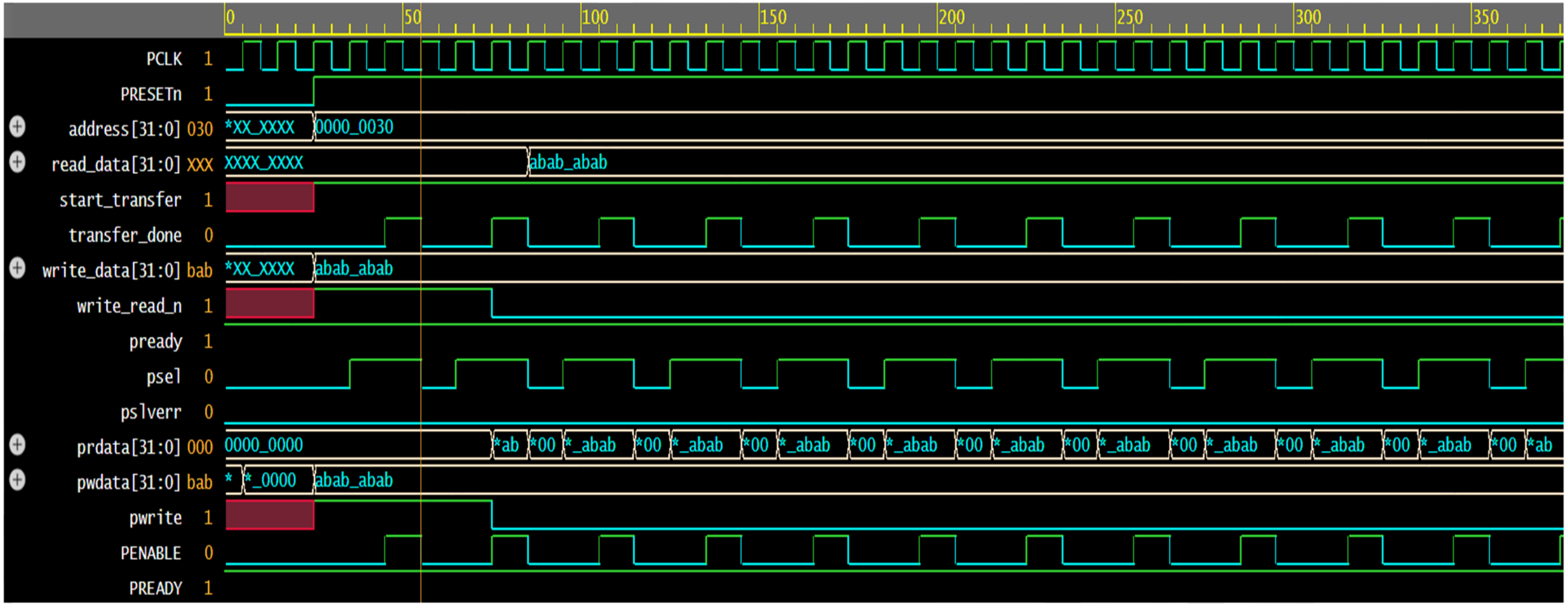
•PENABLE = 1

•PRDATA must be valid before end of this cycle

•Transfer ends at rising edge when PREADY=1



Waveform of my APB:



Conclusion:

- Designed a fully functional AMBA APB Master and Slave following ARM APB3 protocol rules.
- Implemented correct SETUP and ACCESS phases as defined in the official APB specification.
- Waveforms confirm APB-compliant timing for both read and write transfers.
- Proper behavior of PSEL, PENABLE, PWRITE, PADDR, PWDATA, PRDATA, and PREADY.
- APB Master, APB Slave, and Top Wrapper are cleanly separated.
- Can easily integrate more slaves or connect to an AXI/AHB → APB bridge.
- Writes and reads correctly update and fetch data from the slave memory.
- No protocol violations or timing conflicts observed in verification.
- Achieved a complete, verified APB system suitable for SoC peripheral communication
- Demonstrated understanding of AMBA APB protocol, RTL design, and waveform analysis

Repository & Reference Links:

AMBA APB Verilog Project Code:

AMBA APB Code = "**Link**" (Ctrl+Click For open the code "https://www.edaplayground.com/x/iBsh")

Reference:

- ARM Official AMBA APB Document "**IHI0024D_amba_apb_protocol_spec (5)**"

- **VLSI Mentor – AMBA APB Slides**
  Used to understand waveform behavior, setup/access phases, and visual FSM flow.