ARM AMBA 3 APB PROTOCOL

Aaditya Shah
Department of Electrical and Computer Engineering
Portland State University
Portland,Oregon,USA
aadityas@pdx.edu

Bhargav Chunduri
Department of Electrical and Computer Engineering
Portland State University
Portland,Oregon,USA
bhargavc@pdx.edu

Dhushyanth Dharmavarapu
Department of Electrical and Computer Engineering
Portland State University
Portland,Oregon,USA
dharmava@pdx.edu

Sahil Khan
Department of Electrical and Computer Engineering
Portland State University
Portland,Oregon,USA
sahilk@pdx.edu

Abstract

The AMBA 3 APB protocol, part of the AMBA 3 family, is optimized for low-bandwidth peripherals, focusing on simplicity and low power consumption. It operates with a three-state machine (IDLE, SETUP, ACCESS), supports read/write transfers, and allows error handling via a minimal, unpipelined interface. Designed for seamless integration with AHB-Lite and AXI systems, the APB ensures synchronized signal transitions and supports wait states for flexibility. Its efficiency and scalability make it essential for ARM-based embedded systems.

I. INTRODUCTION

The Advanced Peripheral Bus (APB) is a fundamental component of ARM's AMBA (Advanced Microcontroller Bus Architecture), designed to connect low-bandwidth and low-performance peripherals to the main system bus. It is widely used in applications where simplicity, low power consumption, and minimal latency are essential, such as interfacing with peripherals like GPIOs, UARTs, timers, and interrupt controllers. Unlike high-performance buses, APB operates as a non-pipelined, single-clock domain protocol, ensuring straightforward data transfers without the complexity of advanced synchronization or handshaking. It employs a basic address, control, and data phase structure, making it easy to integrate and use. Each transaction is completed in a predictable manner, which is critical for peripherals that require deterministic timing. Furthermore, APB's low-complexity design reduces silicon area and power consumption, making it ideal for embedded systems, microcontrollers, and SoC designs where energy efficiency is paramount. Its widespread adoption highlights its importance in achieving efficient and reliable communication with control-oriented peripherals.

II. BACKGROUND

The Advanced Peripheral Bus (APB) was introduced as part of ARM's AMBA (Advanced Microcontroller Bus Architecture) in the mid-1990s to address the need for a simple, low-cost, and low-power solution for connecting peripheral devices in embedded systems and SoCs (System-on-Chips). Unlike its high-performance counterparts, the Advanced High-Performance Bus (AHB) and Advanced eXtensible Interface (AXI), which were designed for high-speed data throughput and complex operations, APB focuses on peripherals that have less stringent performance requirements but demand efficient communication and minimal complexity.The development of APB was driven by the growing need for efficient communication between the CPU and peripheral devices like timers, GPIOs, UARTs, and interrupt controllers, which typically operate in control-oriented applications. Its non-pipelined architecture and single clock domain simplify its implementation, reducing silicon area and power consumption. This design makes APB particularly well-suited for low-power devices and embedded applications where energy efficiency and reliability are critical.APB's straightforward structure, using sequential data transfers and a simple address, control, and data bus interface, ensures predictable performance, which is critical for control and status operations. Its initial adoption in early embedded systems has expanded significantly over time, making APB an integral part of modern SoC designs. As a lightweight complement to AHB and AXI in the AMBA hierarchy, APB has played a vital role in shaping the design of peripheral subsystems in energy-efficient and cost-sensitive applications.

III. OVERVIEW OF ARCHITECTURE

A. AMBA ARCHITECTURE

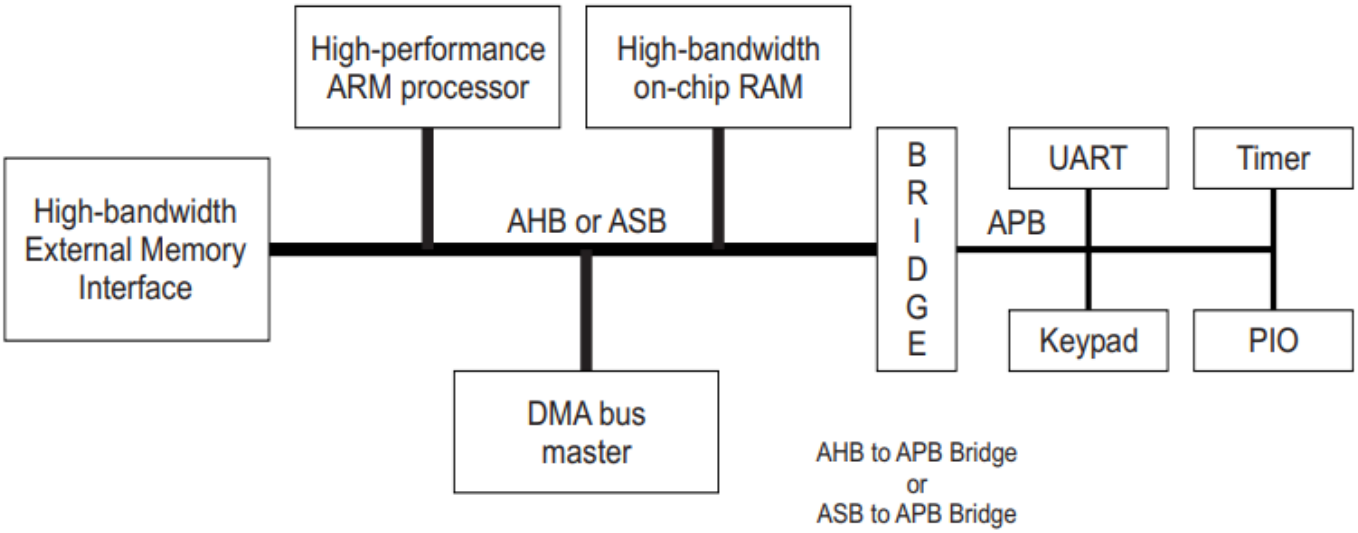

Figure 1. Block Diagram AMBA Architecture

The block diagram represents the AMBA (Advanced Microcontroller Bus Architecture), which is widely used in System-on-Chip (SoC) designs to facilitate communication between different components of a system. It organizes high-bandwidth, high-performance components like processors, memory, and DMA controllers, and connects them to low-bandwidth peripherals like UART, GPIO, and timers. The architecture is designed to optimize performance while maintaining simplicity, scalability, and low power consumption.

In this architecture:

- The AHB/ASB (Advanced High-Performance/System Bus) handles high-speed communication between high-performance components like the CPU, on-chip RAM, and external memory.
- The APB (Advanced Peripheral Bus) is used for low-speed peripherals, reducing power consumption and complexity.
- The Bridge acts as an intermediary between the AHB/ASB and APB, converting signals and protocols for seamless communication.

This organization allows efficient separation of high-speed and low-speed communication, ensuring that each component interacts optimally without unnecessary power or bandwidth overhead.

B. OVERVIEW OF EACH COMPONENT

- High-Performance ARM Processor:

The CPU is the core of the system, responsible for executing instructions and managing overall system operations.
It communicates with other components over the high-speed AHB/ASB, ensuring low-latency access to memory and peripherals.

- High-Bandwidth External Memory Interface:

This module provides access to off-chip external memory, such as DRAM or flash memory.
It is designed for high-speed data transfers required by the CPU for storing large amounts of data or instructions.

- High-Bandwidth On-Chip RAM:

On-chip RAM is a fast-access memory block located within the SoC.
It stores frequently accessed data and instructions, reducing the need to access slower external memory.

- DMA Bus Master:

The Direct Memory Access (DMA) controller allows data to be transferred between memory and peripherals without CPU intervention.This offloads the processor, improving overall system performance and reducing bottlenecks.

- Bridge (AHB/APB or ASB/APB):

The Bridge connects the high-speed AHB/ASB to the low-speed APB.
It handles protocol conversion, ensuring that data can move seamlessly between the high-performance and low-power domains of the system.

- APB (Advanced Peripheral Bus):

APB connects control-oriented peripherals like UART, keypad, timers, and PIO (Programmable Input/Output).
It is optimized for low power and simplicity, providing the necessary interface for peripherals that don't require high-speed data transfers.

- Peripherals (UART, Timer, Keypad, PIO):

UART (Universal Asynchronous Receiver/Transmitter): Used for serial communication between the SoC and external devices.
Timer: Provides precise timing control for various tasks in the system.
Keypad: Handles input from a connected keypad or button interface.

PIO (Programmable Input/Output): Manages general-purpose digital input/output pins for interacting with external devices.

C. APB BRIDGE ARCHITECTURE

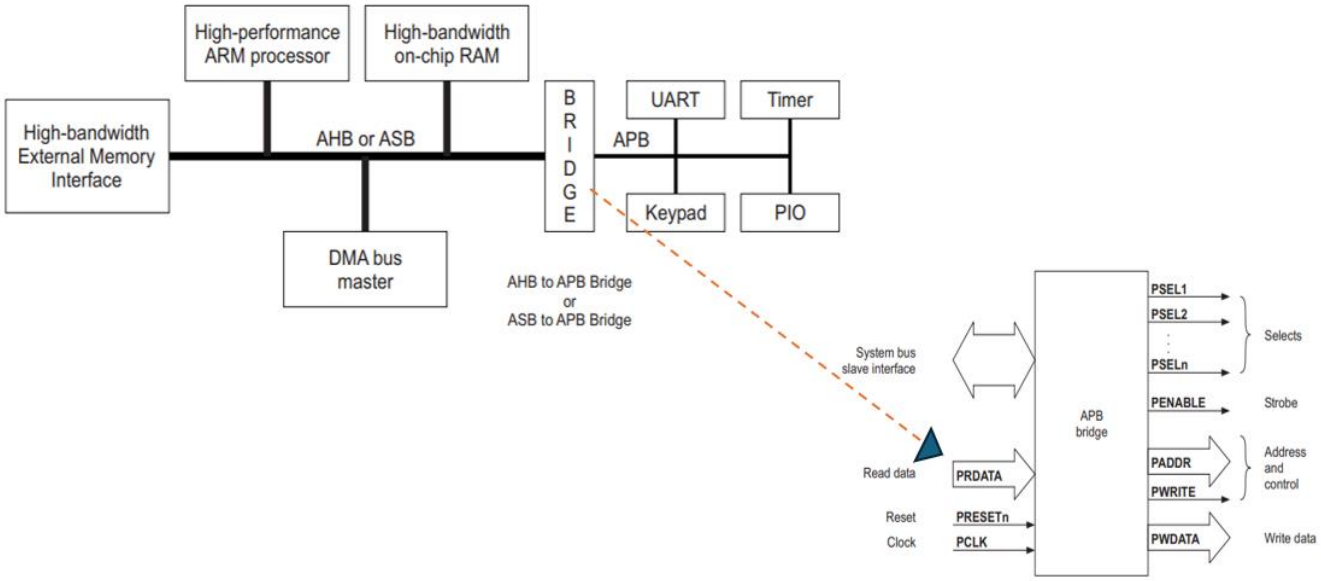

Figure 2. APB Bridge Architecture

The APB Bridge acts as an intermediary between the high-speed, high-performance bus (AHB or ASB) and the low-power, simple APB (Advanced Peripheral Bus). Its primary role is to manage the translation of signals and protocols to ensure that low-speed peripherals, which do not require high bandwidth, can communicate with the rest of the system effectively.
The APB Bridge has two major responsibilities:

1. Protocol Conversion:It converts the complex, high-speed transaction signals of the AHB/ASB to simpler APB signals. This includes adapting address, data, and control signals.
2. Performance Optimization:By isolating the APB from high-speed buses, the APB Bridge ensures that peripheral devices can operate at a slower clock rate, reducing power consumption while maintaining proper functionality.

Signals Used in the APB Bridge
The Figure 2 provides details on several key signals involved in the APB Bridge's operation:
Input Signals to the APB Bridge
System Bus Slave Interface Signals (From AHB or ASB):

- Read Data (PRDATA): Data sent from the peripheral to the system bus during a read operation.
- Write Data (PWDATA): Data sent from the system bus to the peripheral during a write operation.
- Address (PADDR): Specifies the address of the peripheral being accessed.
- Write/Read Control (PWRITE): Determines whether the operation is a write (1) or read (0).
- Reset (PRESETn): Active-low reset signal that initializes the APB and all connected peripherals.
- Clock (PCLK): The clock signal that drives the APB. It typically runs at a lower frequency than the AHB/ASB clock to save power.

APB Output Signals to Peripherals
Select Signals (PSEL1, PSEL2, ..., PSELn):

- These signals determine which peripheral is being addressed in the current transaction.
- Only one peripheral is selected (1) at any given time, while all others remain unselected (0).

Strobe Signal (PENABLE):

- Indicates the start of the access phase for a selected peripheral.
- Ensures that the peripheral begins processing the transaction after all address and control signals are stable.

Data Signals:

- PRDATA (Peripheral Read Data): Data sent back to the system bus from the selected peripheral during a read transaction.
- PWDATA (Peripheral Write Data): Data sent to the selected peripheral from the system bus during a write transaction.

Address and Control Signals:

- PADDR (Peripheral Address): Carries the address for the selected peripheral to specify the register or memory location to be accessed.
- PWRITE: Indicates whether the operation is a read (0) or write (1).

D. APB MASTER SLAVE ARCHITECTURE

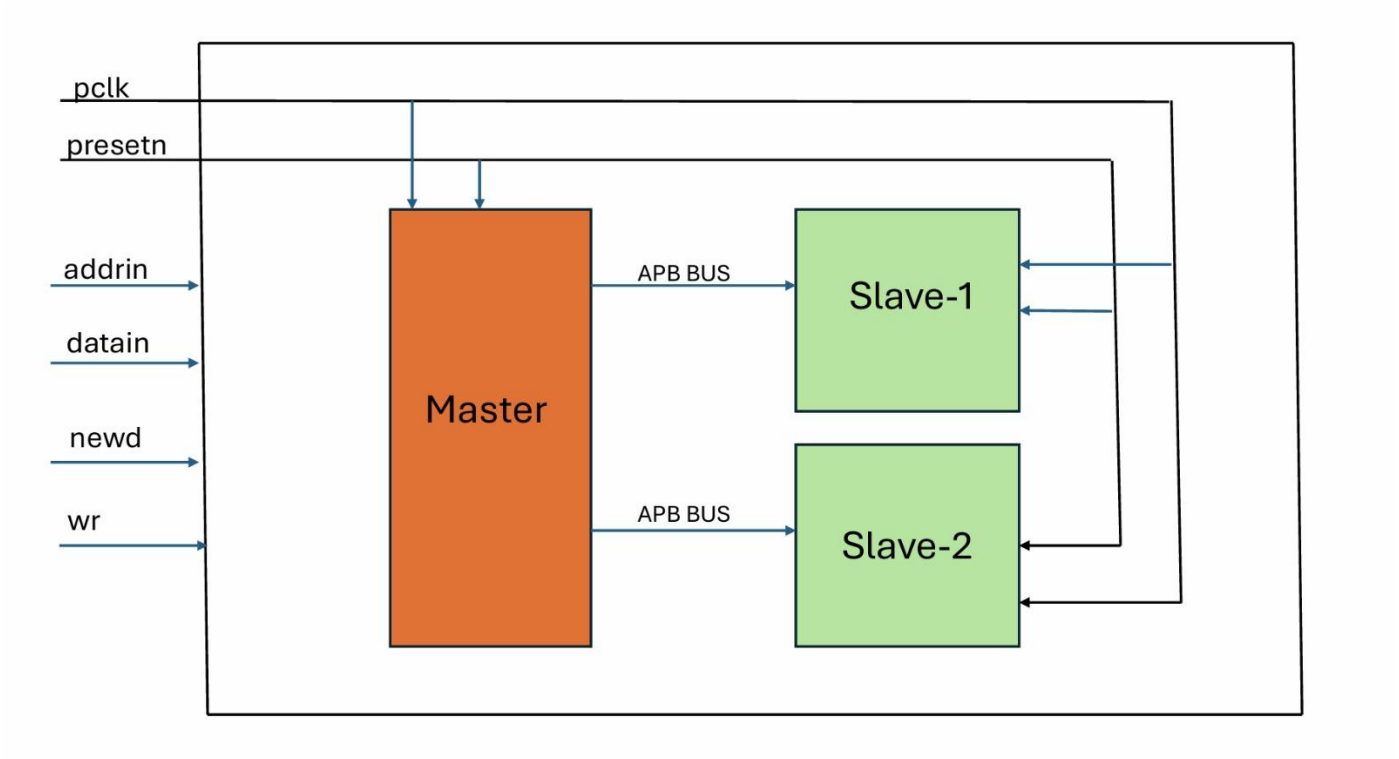

Figure 3. APB Black Box

In the Figure 3 showing the Advanced Peripheral Bus (APB) communication, we can see the interface between the APB Master and APB Slave. The communication is facilitated by a set of control and data signals that ensure proper data transactions across the bus. Here's a detailed description of each signal:
Signals between APB Master and APB Slave:

I. PSEL
- Purpose: Selects a specific slave device on the APB.
- Function: The PSEL signal is used by the APB Master to address a peripheral or slave device. When PSEL is asserted, it indicates that a transaction should be performed with a corresponding APB Slave.

II. PENABLE
- Purpose: Used to enable the access phase of a read or write transaction.
- Function: The PENABLE signal must be asserted by the APB Master to indicate the start of a valid transaction phase. It indicates that the address, control, and data signals are stable and valid.

III. PADDR[31:0]
- Purpose: Address signal for accessing a register or memory location.
- Function: It specifies the address where the data will be read from or written to in the APB slave. It indicates the exact location of a register or memory.

IV. PWRITE
- Purpose: Indicates whether the operation is a read or write transaction.
- Function:
  If PWRITE = 1, it signifies a write operation.
  If PWRITE = 0, it signifies a read operation.

V. PWDATA[31:0]
- Purpose: Write Data Bus
- Function: During a write transaction, this bus carries the 32-bit data that the APB Master sends to the APB Slave.

VI. PRDATA[31:0]
- Purpose: Read Data Bus
- Function: During a read transaction, the PRDATA signal is sent back from the APB Slave to the APB Master, containing the data that the master requested.

VII. PSTB[3:0]
- Purpose: Write strobe signal.
- Function: Used to identify valid byte lanes in the 32-bit write operation, ensuring that only the intended bytes are written.

VIII. PPROT[2:0]

- Purpose: Protection signal for transaction access.
- Function: Used to specify the type of access (e.g., privileged or non-privileged). It ensures proper security and access control.

IX. PREADY

- Purpose: Ready signal.
- Function: This handshake signal indicates whether the slave is ready to accept the transaction request. It tells the master that the slave can perform the operation.

X. PSLVER

- Purpose: Slave Error signal.
- Function: It indicates an error in communication. If PSLVER is asserted by the slave, it informs the master that an error has occurred during a read or write transaction.

SIGNALS WORKING:

Transaction Initiation

- The APB Master first asserts the PSEL signal to select a target slave device.
- Then, it provides the PADDR, PWRITE, and PWDATA signals, specifying the address and data for a read or write operation.
- Finally, PENABLE is asserted to indicate a valid access phase.

Data Communication

- In a Read Transaction:
  The slave responds with PRDATA[31:0].
- In a Write Transaction:
  Data is sent to the slave via the PWDATA[31:0] bus.
  The slave uses PREADY to confirm it is ready for communication.

Error Handling

- If there is an error during a transaction, the slave asserts the PSLVER signal, which communicates issues back to the master, ensuring proper error detection and handling.

IV. OPERATION

The operation of the AMBA 3 APB protocol revolves around its *finite state machine* (FSM) and the specific sequence of signal transitions to facilitate read and write transfers. The protocol defines three operational states—*IDLE, **SETUP, and **ACCESS*—to streamline communication between the master (typically the processor or bus bridge) and one or more slaves (peripherals).

A. TYPES OF OPERATIONS:

1. IDLE State:
   Purpose: Indicates that no transfer is currently active on the bus.
   Behavior:
   Control signals like PSELx, PENABLE are deasserted.
   The bus is free, waiting for a new transaction request from the master.

2. SETUP State:
   Purpose: Prepares the bus for data transfer.
   Behavior:
   The master asserts PSELx to select the peripheral to communicate with.
   Address (PADDR) and control signals (PWRITE) are set.
   Data (PWDATA) is prepared for write transfers.
   The PENABLE signal remains deasserted during this phase.

3. ACCESS State:
   Purpose: Executes the data transfer between the master and the slave.
   Behavior:
   The master asserts PENABLE, signaling the slave to perform the operation.
   In a *read operation*, the slave drives data onto PRDATA.
   In a *write operation*, the slave receives data from PWDATA.
   If the slave requires more time to complete the operation, it deasserts PREADY to introduce wait states.

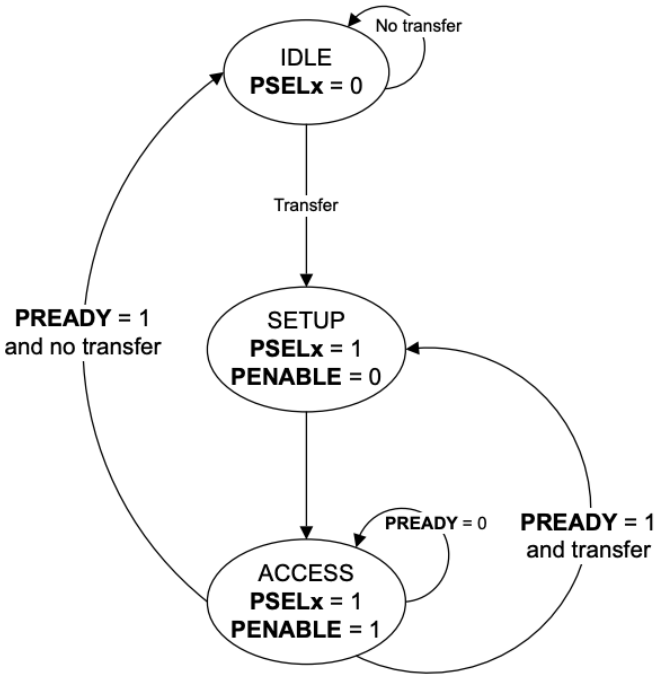

Figure 4. OPERATION STATES

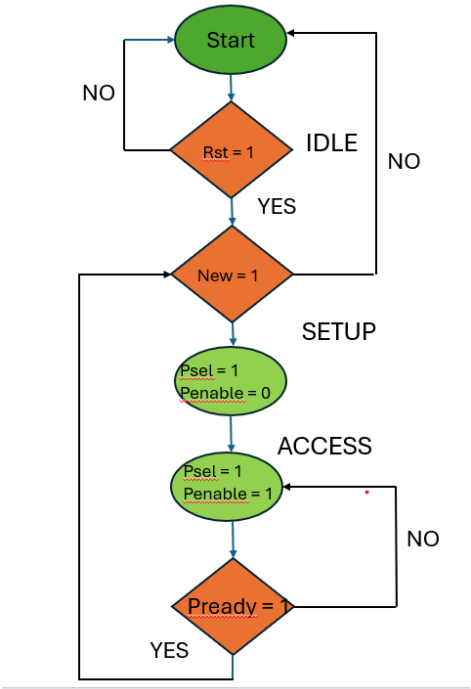

Figure 5. Simple Flowchart of Operations

B. TYPES OF TRANSFERS

1. READ OPERATION (WITHOUT WAIT STATES):
    - At T1, a READ transfer with address PADDR, PWRITE and PSEL starts.They will be registered at rising edge of PCLK.This is SETUP Phase of the transfer.

- After T2, PENABLE and PREADY are registered at the rising edge of PCLK.When asserted, PENABLE indicates the starting of ACCESS phase.
- When asserted, PREADY indicates that slave can complete the transfer at next rising edge of PCLK by providing the data on PRDATA.Slave must provide the data before the end of read transfer. i.e. before T3.

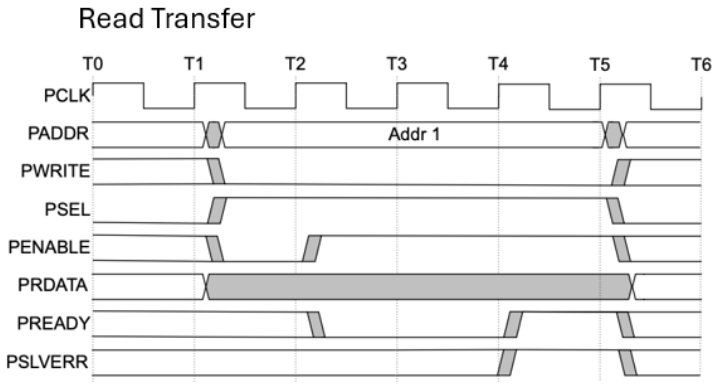

Figure 6. Read Transfer

2. WRITE OPERATION (WITHOUT WAIT STATES):

- At T1, a write transfer with address PADDR,PWDATA,PWRITE and PSEL starts.They will registered at the next rising edge of PCLK, T2.This is Setup Phase of Transfer.
- After T2, PENABLE and PREADY are registered at the rising edge of PCLK.When asserted, PENABLE indicates starting of ACCESS Phase
- When asserted, PREADY indicates that slave can complete the transfer at the next rising edge of PCLK.PADDR, PDATA and control signals all should remain valid till the transfer is completed at T3. PENABLE signal will be de-asserted at the end of the transfer.

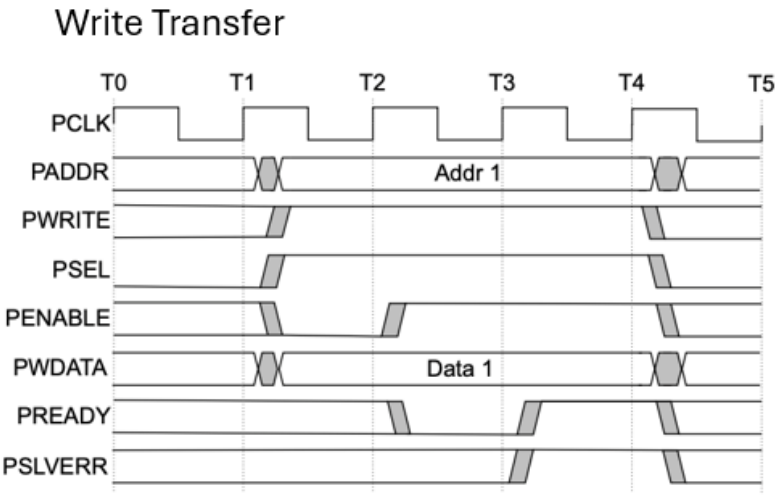

Figure 7. Write Transfer

3. ERROR RESPONSE:
   Whenever there is a problem in the transfer, Slave indicates the error response for the transfer by asserting the PSLVERR signal. PSLVERR is only considered valid during the last cycle f and APB transfer, when PSEL, PENABLE, and PREADY are all HIGH.
   It is recommended, but not mandatory that you drive PSLVERR low when it is not being sampled.

Transactions that receive an error response, might or might not have changed the state of peripheral. For example, If the APB master performs a write transaction to an APB slave and receives an error response, it is not guaranteed that the data is not written on the slave peripheral.

Error Response for a read transfer:

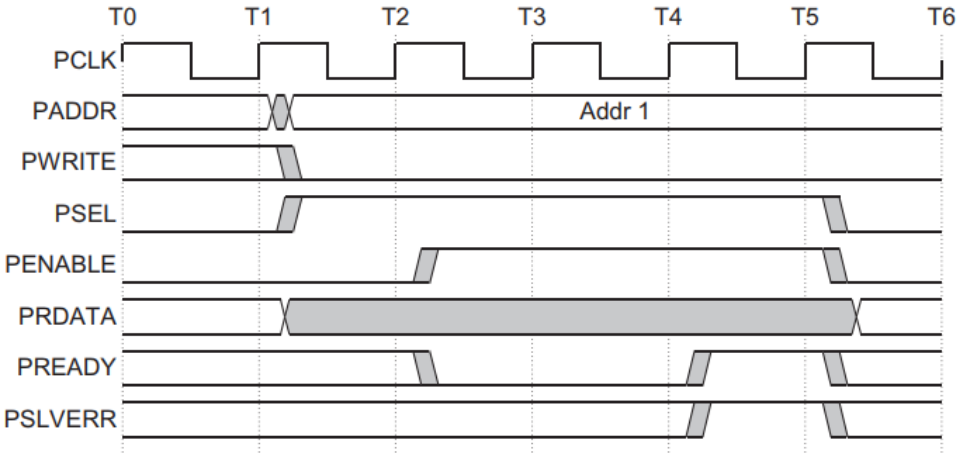

Figure 8. Read Transfer Error Timing Diagram

Error Response for a write transfer:

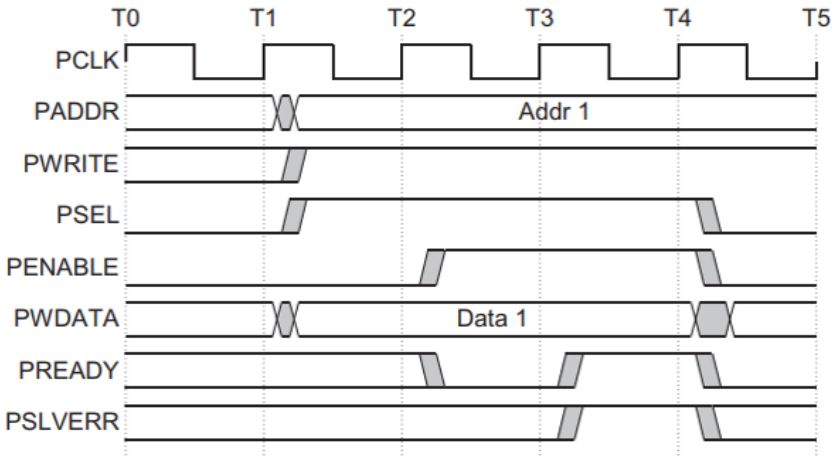

Figure 9. Write Transfer Error Timing Diagram

V. IMPLEMENTATION

This project demonstrates a robust implementation of an **APB master-slave system** with error detection and verification through a testbench. It highlights essential digital design concepts like hierarchical modules, FSM-based control, and bus protocols.
This project implements a **multi-slave Advanced Peripheral Bus (APB) system** consisting of:

1. **Master Module (apb_m_top):**

   Controls transactions and directs them to one of two slaves based on the slave address.

   Handles APB signals (pwrite, psel, penable, etc.).

```
// Reset decoder
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn)
        state <= IDLE;
    else
        state <= nstate;
end

// State machine logic
always_comb begin
    case (state)
        IDLE: nstate = newd ? SETUP : IDLE;
        SETUP: nstate = ENABLE;
        ENABLE: nstate = (newd && slave_if.pready) ? SETUP : ENABLE;
        default: nstate = IDLE;
    endcase
end

// Address decoding
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn) begin
        master_out.psel1 <= 1'b0;
        master_out.psel2 <= 1'b0;
    end else if (nstate == IDLE) begin
        master_out.psel1 <= 1'b0;
        master_out.psel2 <= 1'b0;
    end else if (nstate == SETUP || nstate == ENABLE) begin
        master_out.psel1 <= (slv_addr_in == 2'b01);
        master_out.psel2 <= (slv_addr_in == 2'b10);
    end else begin
        master_out.psel1 <= 1'b0;
        master_out.psel2 <= 1'b0;
    end
end
```

Figure 10. Master Module

2. **Slave Modules (apb_s_top):**

Two slaves (s1 and s2), each managing memory and responding to the master's requests.

Performs read and write operations and detects errors (e.g., invalid addresses or data).

```
// Reset decoder
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn)
        state <= IDLE;
    else
        state <= nstate;
end

// Next state and output decoder
always_comb begin
    prdata = '0;
    pready = 1'b0;

    case (state)
        IDLE: begin
            if (psel && pwrite)
                nstate = WRITE;
            else if (psel && !pwrite)
                nstate = READ;
            else
                nstate = IDLE;
        end
        WRITE: begin
            if (psel && penable) begin
                if (!addr_err && !addv_err && !data_err) begin
                    pready = 1'b1;
                    mem[paddr] = pwdata;
                end
                nstate = IDLE;
            end
        end
        READ: begin
            if (psel && penable) begin
                if (!addr_err && !addv_err && !data_err) begin
                    pready = 1'b1;
                    prdata = mem[paddr];
                end
                nstate = IDLE;
            end
        end
        default: nstate = IDLE;
    endcase
end
```

Figure 11. Slave Module

3. **Top-Level Wrapper (mul_slave):**

Integrates the master and slaves into a complete APB system.

Ensures proper data and signal flow between the master and selected slave.

```
// Master module
apb_m_top m1 (
    .pclk(pclk),
    .presetn(presetn),
    .slv_addr_in(slv_addr_in),
    .addrin(addrin),
    .datain(datain),
    .wr(wr),
    .newd(newd),
    .slave_if(selected_slave_if), // Pass selected slave interface
    .master_out(master_out),
    .dataout(dataout)
);

// Slave 1
apb_s_top s1 (
    .pclk(pclk),
    .presetn(presetn),
    .paddr(master_out.addr),
    .psel(master_out.psel1),
    .penable(master_out.penable),
    .pwdata(master_out.wdata),
    .pwrite(master_out.pwrite),
    .prdata(slave_if1.rdata),
    .pready(slave_if1.pready),
    .pslverr(slave_if1.pslverr)
);

// Slave 2
apb_s_top s2 (
    .pclk(pclk),
    .presetn(presetn),
    .paddr(master_out.addr),
    .psel(master_out.psel2),
    .penable(master_out.penable),
    .pwdata(master_out.wdata),
    .pwrite(master_out.pwrite),
    .prdata(slave_if2.rdata),
    .pready(slave_if2.pready),
    .pslverr(slave_if2.pslverr)
);
```

Figure 12. Master Slave Instantiation

4. **Interface :**

```
import apb_pkg::*;

interface apb_if (
    input logic pclk,           // Clock signal
    input logic presetn,        // Reset signal
    input logic [3:0] paddr,    // Address bus
    input logic psel,           // Select signal
    input logic penable,        // Enable signal
    input logic [7:0] pwdata,   // Write data
    input logic pwrite,         // Write control signal
    output logic [7:0] prdata,  // Read data
    output logic pready,        // Ready signal
    output logic pslverr        // Slave error signal
);

    // The APB slave interface signals
    apb_slave_if_t slave_if;

    // The APB master interface signals
    apb_master_if_t master_if;

endinterface
```

Figure 13. Interface

```
   /* // Error handling
    assign addr_err = (paddr <= 16);
    assign addv_err = (paddr <= 0);  // Replace with address validation logic
    assign data_err = (pwdata == '0); // Replace with data validation logic

    assign pslverr = (psel && penable) ? (addr_err || addv_err || data_err) : 1'b0; */

// Checking valid values of address
  logic av_t;
  always_comb begin
    av_t = (paddr >= 0) ? 1'b0 : 1'b1;
  end

  // Checking valid values of data
  logic dv_t;
  always_comb begin
    dv_t = (pwdata >= 0) ? 1'b0 : 1'b1;
  end

  assign addr_err = ((nstate == WRITE|| nstate == READ) && (paddr > 4'hF)) ? 1'b1 : 1'b0;
  assign addv_err = (nstate == WRITE || nstate == READ) ? av_t : 1'b0;
  assign data_err = (nstate == WRITE || nstate == READ) ? dv_t : 1'b0;

  assign pslverr = (psel == 1'b1 && penable == 1'b1) ? (addv_err || addr_err || data_err) : 1'b0;
```

Figure 14. Error Detection

```
// Output assignments
always_ff @(posedge pclk or negedge presetn) begin
    if (!presetn) begin
        master_out.penable <= 1'b0;
        master_out.addr <= '0;
        master_out.wdata <= '0;
        master_out.pwrite <= 1'b0;
    end else if (nstate == SETUP) begin
        master_out.addr <= addrin;
        master_out.pwrite <= wr;
        if (wr)
            master_out.wdata <= datain;
    end else if (nstate == ENABLE) begin
        master_out.penable <= 1'b1;
    end
end

assign dataout = (master_out.penable && !wr) ? slave_if.rdata : 8'h00;
```

Figure 15. Output Logic

VI. SIMULATION RESULTS

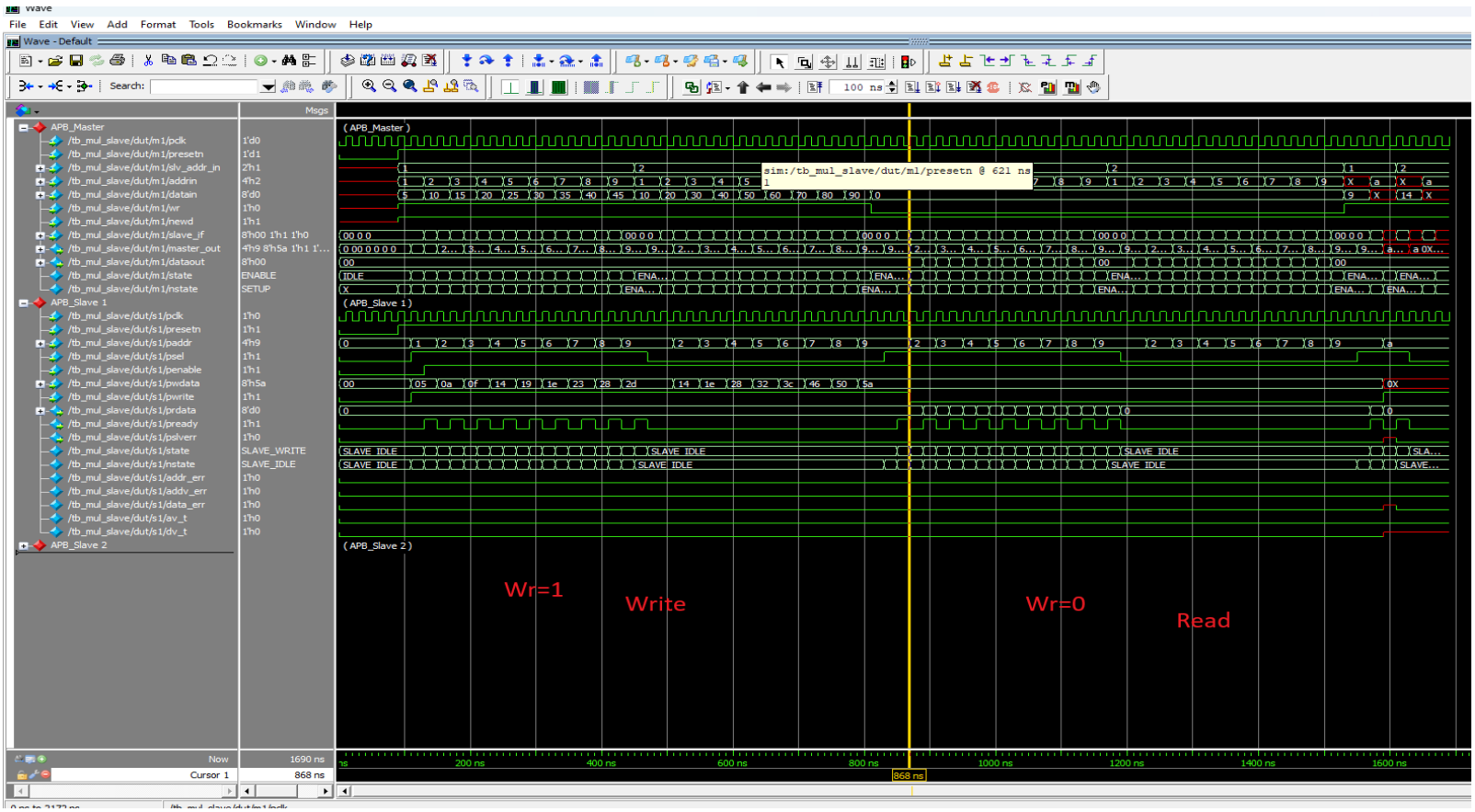

Figure 16.. Output Slave 1

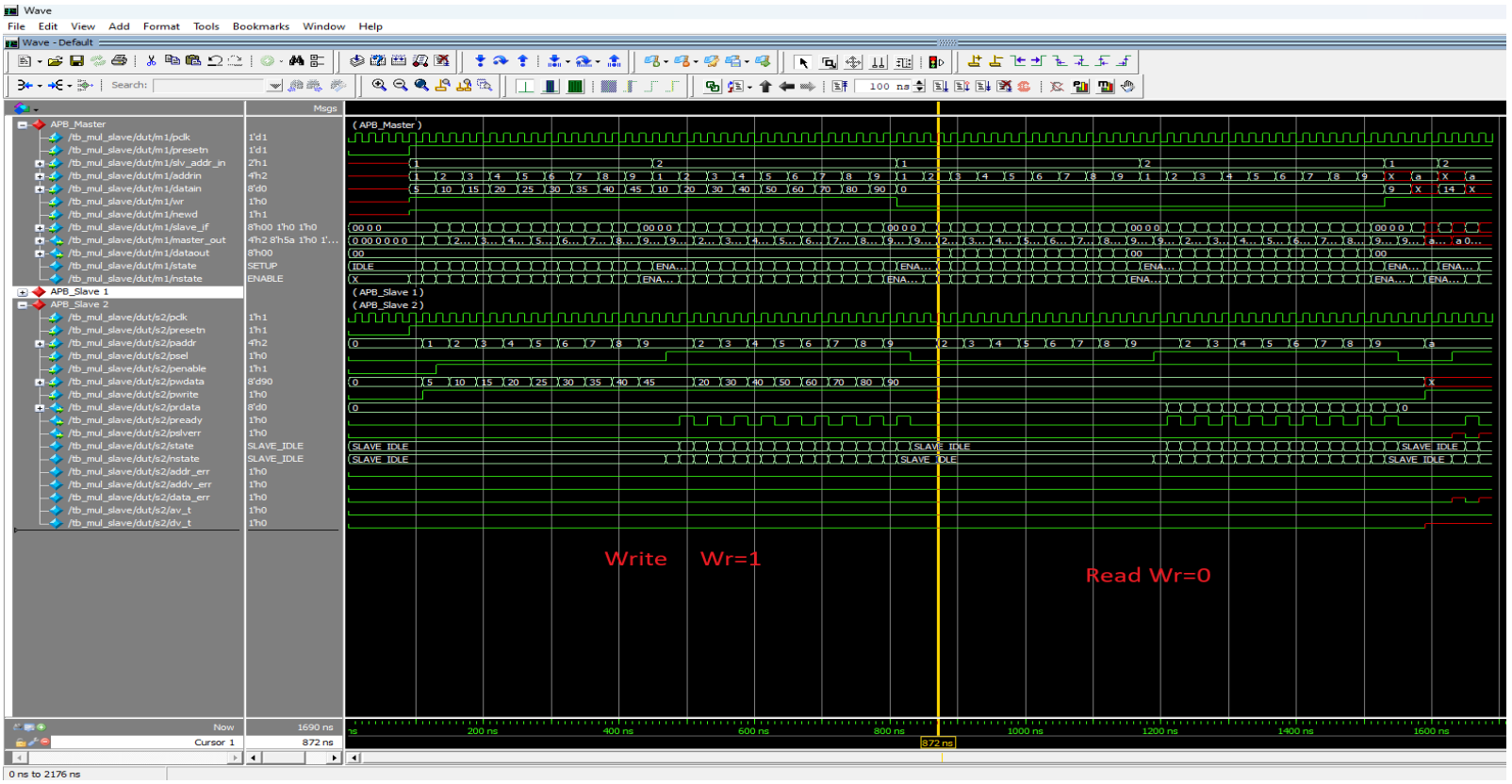

Figure 17.. Output Slave 2

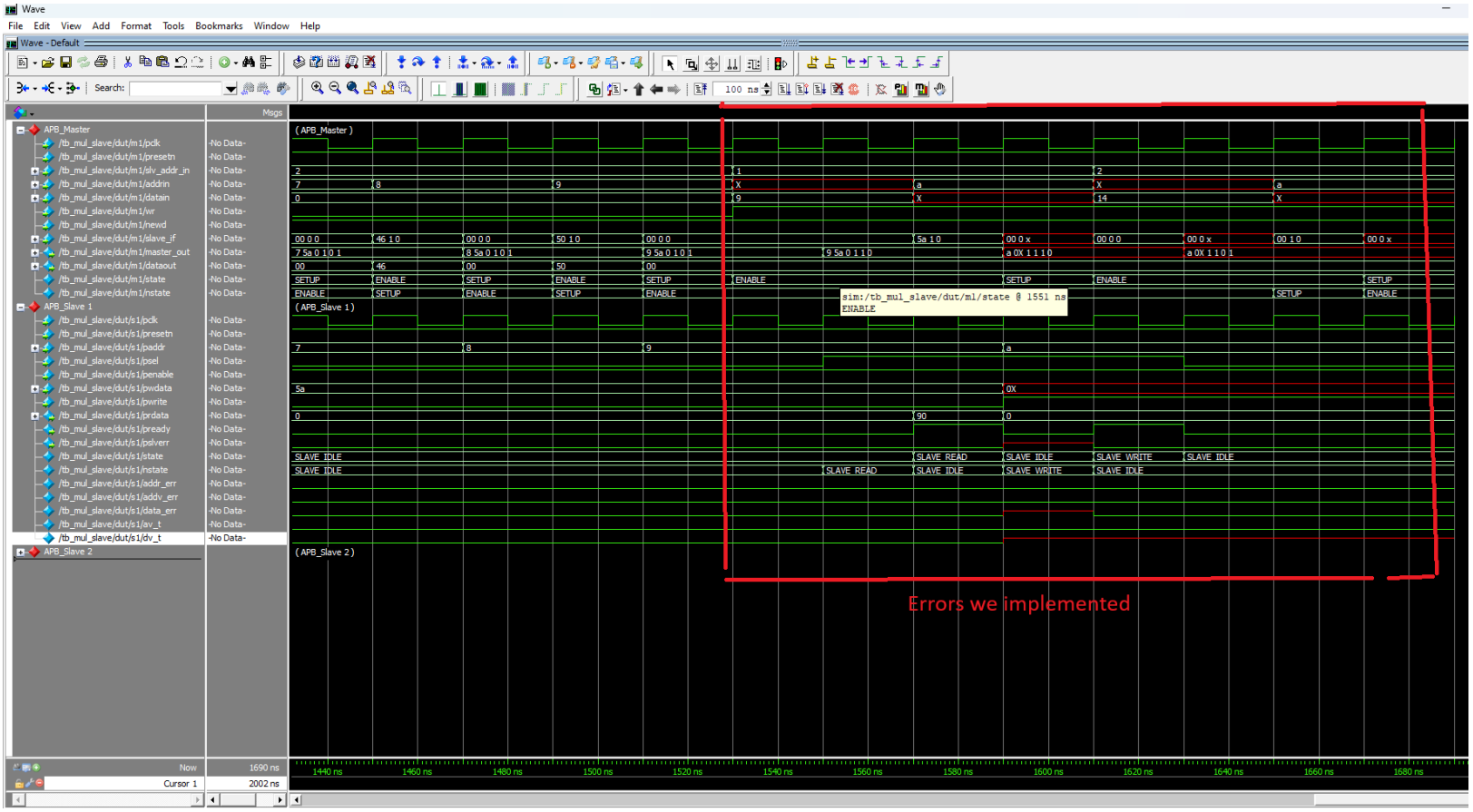

Figure 18. Slave1 Error.

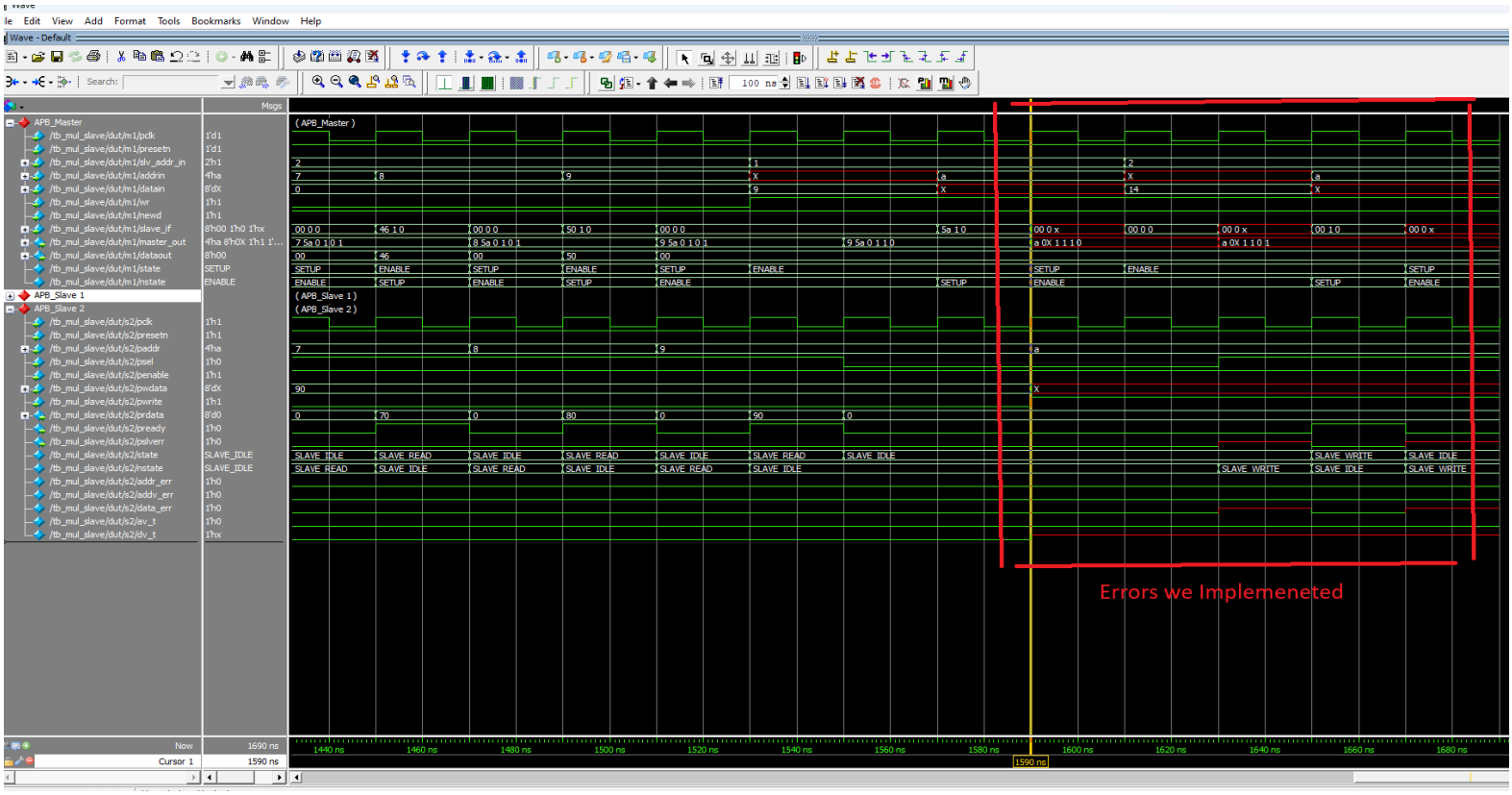

Figure 19. Slave 2 Error

VII. CHALLENGES

1. We started by focusing on the design phase, where we concentrated on developing a clean and efficient RTL architecture. This phase involved in-depth discussions on the design's requirements and specifications, ensuring that we had a solid foundation before moving forward. Once the initial design was in place, we moved on to verification, which went smoothly and successfully. We created a robust verification environment, executing functional and corner-case tests to ensure the correctness and reliability of the design.

2. After completing the verification phase, we turned our attention to further improving the efficiency of the codebase. One of the major steps involved leveraging SystemVerilog constraints in our testbenches. By utilizing constrained random generation, we were able to generate a wide range of test scenarios while minimizing the number of lines of code required. This not only made the code more compact and maintainable but also increased the professional quality of the verification process.

3. SystemVerilog constraints allowed us to control complex relationships between signal values, reducing the need for manually written test cases. The constraints enabled automatic generation of highly diverse and meaningful stimulus, which improved our ability to cover more functional scenarios with fewer lines of code. This approach not only optimized code efficiency but also helped in identifying potential issues early in the verification cycle, further improving the design's robustness.

VIII. References

[1] https://verificationforall.wordpress.com/apb-protocol/

[2] https://ieeexplore.ieee.org/document/9388549

[3] https://ieeexplore.ieee.org/document/6825442