PROJECT REPORT ON

DESIGN AND VERIFICATION OF AHB TO APB BRIDGE

Submitted in Partial Fulfillment for the award of

Post Graduate Diploma in VLSI Design (PG-DVLSI)

from

C-DAC, ACTS (Pune)

Guided by:

Mr. AJIT JAIN

Presented By:

Mr. Chittibomma Surya Teja	**PRN: 240840133005**
Mr. Das Manoj Kumar Trilochan	**PRN: 240840133006**
Mr. Kakarla Harsha Sai	**PRN: 240840133012**
Mr. Kushwaha Abhishek Ajit	**PRN: 240840133016**
Mr. Satyam Yadav	**PRN: 240840133031**

Centre for Development of Advanced Computing

(C-DAC), Pune

ACKNOWLEDGEMENT

This project "Design and Verification of AHB To APB Bridge" was a great learning experience for us and we are submitting this work to Advanced Computing Training School (CDAC ACTS).

We are very glad to mention the name of Mr. Ajit Jain for their valuable guidance in working on this project. Their guidance and support helped me to overcome various obstacles and intricacies during project work.

Our heartfelt thanks go to Mrs. Swati Salunkhe (Course Coordinator, PG-DVLSI) who gave all the required support and kind coordination to provide all the necessities like required hardware, internet facility, and extra Lab hours to complete the project and throughout the course up to the last day here in C-DAC ACTS, Pune.

TABLE OF CONTENTS

ABSTRACT

This project focuses on the design and implementation of an AHB-to-APB bridge. The bridge serves as an intermediary component, facilitating communication between the high-performance Advanced High-performance Bus (AHB) and the low-power Advanced Peripheral Bus (APB).

This bridge is needed to integrate high-performance processors with lower-bandwidth peripherals within SoCs. The bridge ensures seamless communication between these components by adapting AHB transactions to the slower APB protocol. While data integrity is maintained, wait states might be introduced to synchronize the communication due to the speed difference.

The project involves designing the bridge architecture using Hardware Description Language (HDL) like Verilog, adhering to the respective AHB and APB specifications. Rigorous simulation techniques will thoroughly test the bridge's functionality in various scenarios, including read, write, and burst transfers. The bridge's performance metrics like latency, throughput, and resource utilization will also be analyzed.

The expected outcomes include a fully functional and verified AHB to APB bridge design, comprehensive documentation detailing the design process, verification methodology, and performance analysis results. This project will also contribute to a deeper understanding of bus protocols and their integration within SoCs.

The potential applications of this bridge lie in integrating low-power peripherals into high-performance SoCs and designing efficient and scalable communication architectures for different systems.

INTRODUCTION

AHB (Advanced High-Performance Bus): This high-performance bus excels in data transfer between core processors, memory controllers, and other high-bandwidth peripherals. It prioritizes speed and throughput, making it ideal for operations demanding real-time responsiveness.

APB (Advanced Peripheral Bus): In contrast, the APB focuses on optimized power consumption for low-power peripherals like timers, sensors, and I/O interfaces. It operates at lower frequencies than AHB but ensures efficient utilization of battery life in portable devices.

The AHB and APB Buses:

The ever-growing demand for performance and power efficiency in modern electronic devices presents a significant challenge for designers of System-on-Chip (SoC) architectures. SoCs integrate various processing units, memory controllers, and peripheral devices onto a single chip, each with distinct performance and power requirements. To address this heterogeneity, designers employ specialized communication protocols and interfaces tailored to specific needs.

While both AHB and APB serve crucial purposes within an SoC, their inherent differences in performance and power characteristics can create communication hurdles. This is where the AHB2APB bridge emerges as a critical component. It acts as an intermediary, seamlessly translating data between the high-performance AHB domain and the low-power APB domain.

RELEVANCE

The Arm Advanced Microcontroller Bus Architecture, or AMBA, is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. Essentially, AMBA protocols define how functional blocks communicate with each other. AMBA simplifies the development of designs with multiple processors and large numbers of controllers and peripherals. However, the scope of AMBA has increased over time, going far beyond just microcontroller devices.

Today, AMBA is widely used in a range of ASIC and SoC parts. These parts include application processors that are used in devices like IoT subsystems, smartphones, and networking SoCs.

AMBA provides several benefits:

• **Efficient IP reuse**

IP reuse is an essential component in reducing SoC development costs and timescales. AMBA specifications provide the interface standard that enables IP reuse. Therefore, thousands of SoCs, and IP products, are using AMBA interfaces.

• **Flexibility**

AMBA offers the flexibility to work with a range of SoCs. IP reuse requires a common standard while supporting a wide variety of SoCs with different power, performance, and area requirements. Arm offers a range of interface specifications that are optimized for these different requirements.

• **Compatibility**

A standard interface specification, like AMBA, allows compatibility between IP components from different design teams or vendors.

• **Support**

AMBA is well supported. It is widely implemented and supported throughout the semiconductor industry, including support from third-party IP products and tools. Bus interface standards like AMBA, are differentiated through the performance that they enable. The two main characteristics of bus interface performance are:

• **Bandwidth**

The rate at which data can be driven across the interface. In a synchronous system, the maximum bandwidth is limited by the product of the clock speed and the width of the data bus.

• **Latency**

The delay between the initiation and completion of a transaction. In a burst-based system, the latency figure often refers to the completion of the first transfer rather than the entire burst. The efficiency of your interface depends on the extent to which it achieves the maximum bandwidth with zero latency.

LITERATURE SURVEY

The Advanced Microcontroller Bus Architecture (AMBA) provides a set of standardized on-chip interconnect protocols for integrating various components within a system-on-chip (SoC). Two key AMBA buses are the Advanced High-Performance Bus (AHB) and the Advanced Peripheral Bus (APB). AHB is a high-performance bus designed for high-bandwidth communication between processors and memory controllers, while APB is a low-power bus ideal for connecting low-bandwidth peripherals.

An AHB2APB bridge plays a crucial role in bridging the communication gap between these two buses. It acts as an interface, translating AHB transactions into compatible APB transactions and vice versa. This report presents a literature survey on AHB2APB bridges, exploring their design principles, functionalities, and implementation aspects.

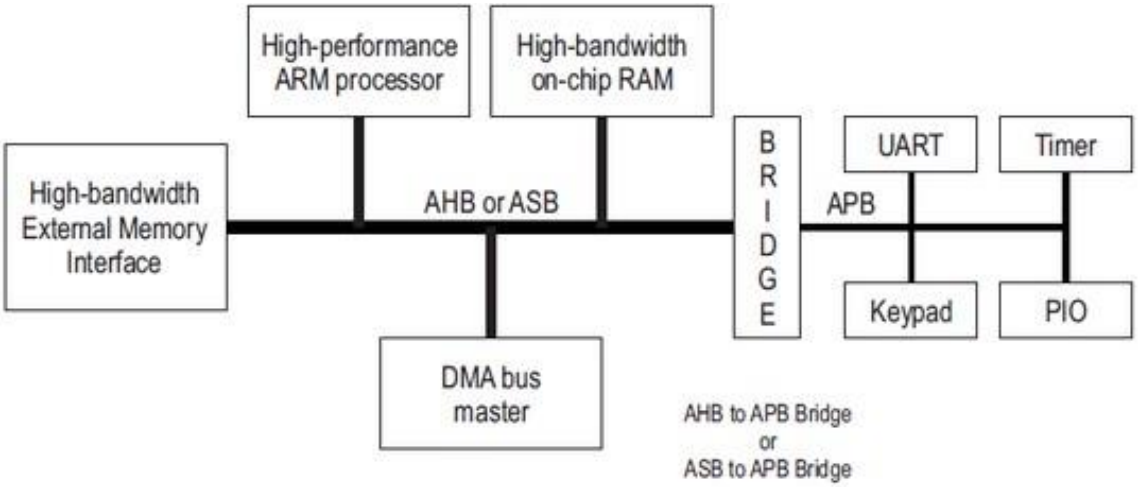


Fig.1.1 AHB TO APB BRIDGE

Design Principles:

Several key design principles govern the development of efficient AHB2APB bridges:

Protocol Conversion: The bridge must effectively convert AHB signals (address, control, and data) into their corresponding APB counterparts, ensuring compatibility and maintaining data integrity.

Clock Domain Crossing: AHB and APB often operate at different clock frequencies and phases. The bridge needs to implement proper clock domain crossing techniques to synchronize signals and avoid timing violations.

Buffering: To handle potential differences in data transfer rates between AHB and APB, the bridge may employ buffering mechanisms to decouple the two buses and optimize performance.

Error Handling: The bridge should incorporate error detection and correction mechanisms to ensure reliable data transfers and system stability.

Functionalities:

AHB2APB bridges typically offer the following functionalities:

Address Translation: Maps AHB addresses to equivalent APB addresses based on a pre-defined memory map.

Signal Conversion: Adapts AHB control signals to match APB requirements, including transfer type, burst size, and error indications.

Data Buffering: Provides temporary storage for data transfers between AHB and APB to manage potential speed mismatches.

Handshake Mechanism: Implements handshake protocols to coordinate data transfers between the bridge and connected peripherals on the APB bus.

Error Signaling: Detects and reports errors during data transfers, such as parity errors or timeouts.

Implementation Aspects:

AHB2APB bridges can be implemented using various hardware design methodologies:

Register Transfer Level (RTL) Design: This approach involves describing the bridge's functionality using hardware description languages like Verilog or VHDL. The RTL code is then synthesized into a gate-level netlist for further implementation.

Hardware IPs (Intellectual Property): Pre-designed and verified bridge IP cores are available from various vendors and can be integrated into the SoC design, reducing development time and effort.

Field-Programmable Gate Arrays (FPGAs): FPGAs offer flexibility for implementing custom bridge designs, although they may not achieve the same performance as dedicated ASIC implementations.

AHB2APB bridges play a critical role in facilitating communication between high-performance processors and low-power peripherals within SoCs. Understanding their design principles, functionalities, and implementation aspects is crucial for developing efficient and reliable embedded systems. This literature survey provides a starting point for further exploration of AHB2APB bridges and their role in SoC design.

AIM OF THE PROJECT

This project aims to design and implement an efficient and reliable AHB2APB bridge for integration within a System-on-Chip (SoC) architecture. The bridge will facilitate seamless communication between high-performance AHB masters (e.g., processors) and low-power APB peripherals.

Specific objectives of the project include:

Functional correctness: Ensure the bridge accurately translates AHB transactions into compatible APB transactions and vice versa, maintaining data integrity and adhering to AMBA specifications.

Performance optimization: Design the bridge to achieve optimal throughput and minimize latency while considering potential speed mismatches between AHB and APB.

Resource efficiency: Implement the bridge using hardware design techniques that minimize resource utilization (e.g., area, power) while meeting performance requirements.

Verification and validation: Develop a comprehensive test plan to thoroughly verify the bridge's functionality under various scenarios and validate its performance against set benchmarks.

Expected outcomes of the project:

A fully functional AHB2APB bridge design implemented using hardware description languages (HDL) or hardware IP cores. Detailed documentation outlining the bridge's design principles, functionalities, and implementation choices. Simulation results demonstrating the bridge's correct operation and adherence to AMBA specifications. Performance analysis evaluating the bridge's throughput, latency, and resource consumption.

This project contributes to the advancement of SoC design by providing a crucial component for efficient communication between different processing units and peripherals. The successful implementation of an AHB2APB bridge can be integrated into various SoC applications, enabling efficient data exchange and system functionality.

FUTURE SCOPE

Future research on AHB2APB bridges can explore several potential areas:

- **Performance optimization:** Investigating techniques to improve bridge throughput and reduce latency for high-bandwidth applications.
- **Low-power design:** Exploring power-efficient bridge architectures suitable for battery-powered devices.

THEORETICAL DESCRIPTION OF PROJECT

The Advanced High-performance Bus (AHB) protocol is a key component of the Advanced Microcontroller Bus Architecture (AMBA) suite, designed for high-bandwidth communication within System-on-Chip (SoC) architectures.

AHB PROTOCOL

1. Introduction:

The ever-increasing complexity of SoCs necessitates efficient on-chip communication mechanisms to connect various processing units, memory controllers, and peripherals. AMBA addresses this challenge by providing standardized buses, including AHB, for seamless data exchange within the SoC.

AHB is specifically designed for high-performance applications, offering features like:

High bandwidth: Supports burst transfers and pipelined operations for efficient data movement.

Scalability: Adaptable to various SoC sizes and configurations.

Low latency: Minimizes delays in data transfers, improving system responsiveness.

Flexibility: Supports a wide range of masters and slaves, including processors, memory controllers, and peripherals.

2. AHB Protocol Overview:

AHB operates on a master-slave architecture, where one master device initiates data transfers and multiple slave devices respond to these requests. The protocol defines various signal types for communication:

Address signals: Specify the memory location or peripheral register being accessed.

Control signals: Indicate the type of transfer (read/write), burst size, and error conditions.

Data signals: Carry the actual data being transferred between master and slave.

3. Key Features of AHB:

Burst Transfers: AHB supports burst transfers, allowing the transmission of multiple data items in a single transaction, improving efficiency compared to individual transfers.

Split Transactions: Complex transfers can be divided into address, data, and response phases, enabling concurrent processing, and reducing overall latency.

Pipeline Stages: AHB employs pipelining techniques to overlap different stages of a transaction, further optimizing data throughput.

Error Signaling: The protocol incorporates mechanisms to detect and report errors during transfers, ensuring data integrity.

Multiple Masters: AHB allows multiple masters to access the bus in a controlled manner, using arbitration mechanisms to prevent conflicts.

4. AHB Transaction Process:

A typical AHB transaction involves the following steps:

The master initiates transfer: The master asserts address and control signals on the bus, specifying the target slave, transfer type, and data size.

Slave selection: The arbiter grants bus access to the appropriate slave based on a predefined priority scheme.

Data transfer: The master drives data onto the bus for write operations or receives data from the slave for read operations.

Transfer completion: The slave acknowledges the transfer completion by asserting appropriate response signals.

5. Implementation Considerations:

Several factors need to be considered when implementing the AHB protocol:

Clock Domain Crossing: AHB often operates at different clock frequencies than connected peripherals. Asynchronous design techniques are essential to handle these clock domain differences and avoid timing violations.

Signal Integrity: The careful design of bus lines and drivers is crucial to ensure proper signal transmission and reception, especially at high speeds.

Verification and Validation: Rigorous testing methodologies are necessary to verify the functional correctness and performance of the AHB implementation.

6. Benefits of AHB:

The adoption of AHB in SoC design offers several advantages:

Improved performance: High bandwidth, burst transfers, and pipelining enable efficient data movement, enhancing system responsiveness.

Reduced design complexity: Standardized protocol simplifies integration of various components within the SoC.

Scalability: Adaptable to diverse SoC configurations with varying numbers of masters and slaves.

Interoperability: Compatibility with other AMBA protocols facilitates seamless communication between different on-chip components.

6. Applications of AHB:

AHB finds application in various SoC designs, including:

Microprocessors and microcontrollers: Connecting processors to memory controllers, caches, and peripherals.

Network-on-Chip (NoC) designs: Facilitating high-speed communication between different processing units within the NoC.

Multimedia and graphics processing systems: Enabling efficient data exchange between processors, memory, and dedicated hardware accelerators.

7. AHB Master:

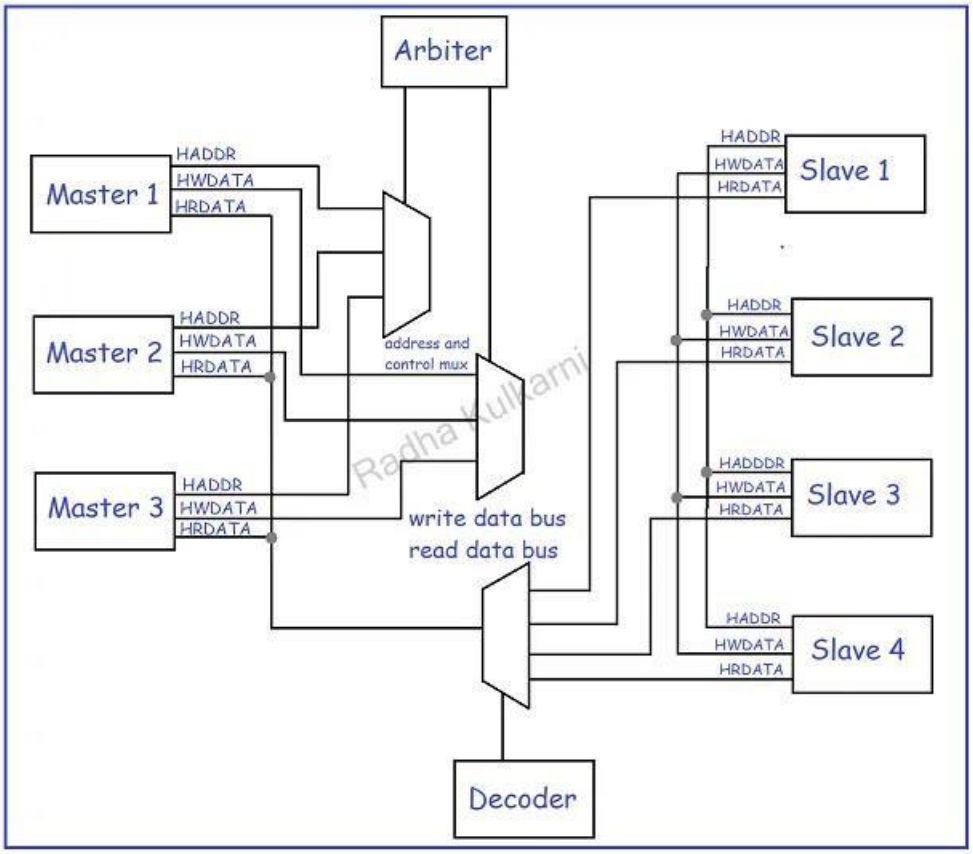


Fig.2.1 AHB MASTER

The fig2.1 shows 3 AHB Masters with its 3 slaves. The bus interconnect logic consists of one address decoder and a slave-to-master multiplexer. The decoder monitors the address of the master so that the appropriate slave must be selected. Multiplexer routes the appropriate slave

output data back to the Master. The arbiter is used to select the Master whose data is going to pass into the slave. The selection of a Master depends upon some factors.

8. AHB SLAVE

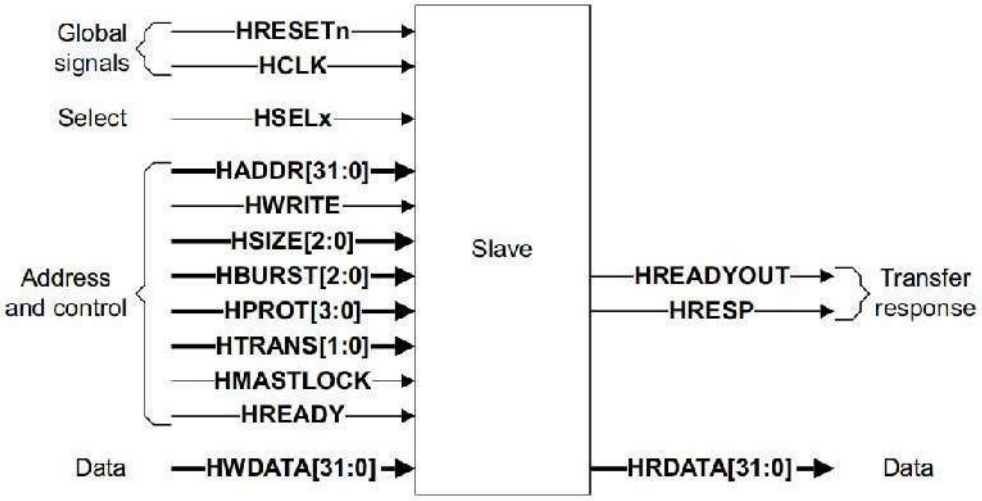


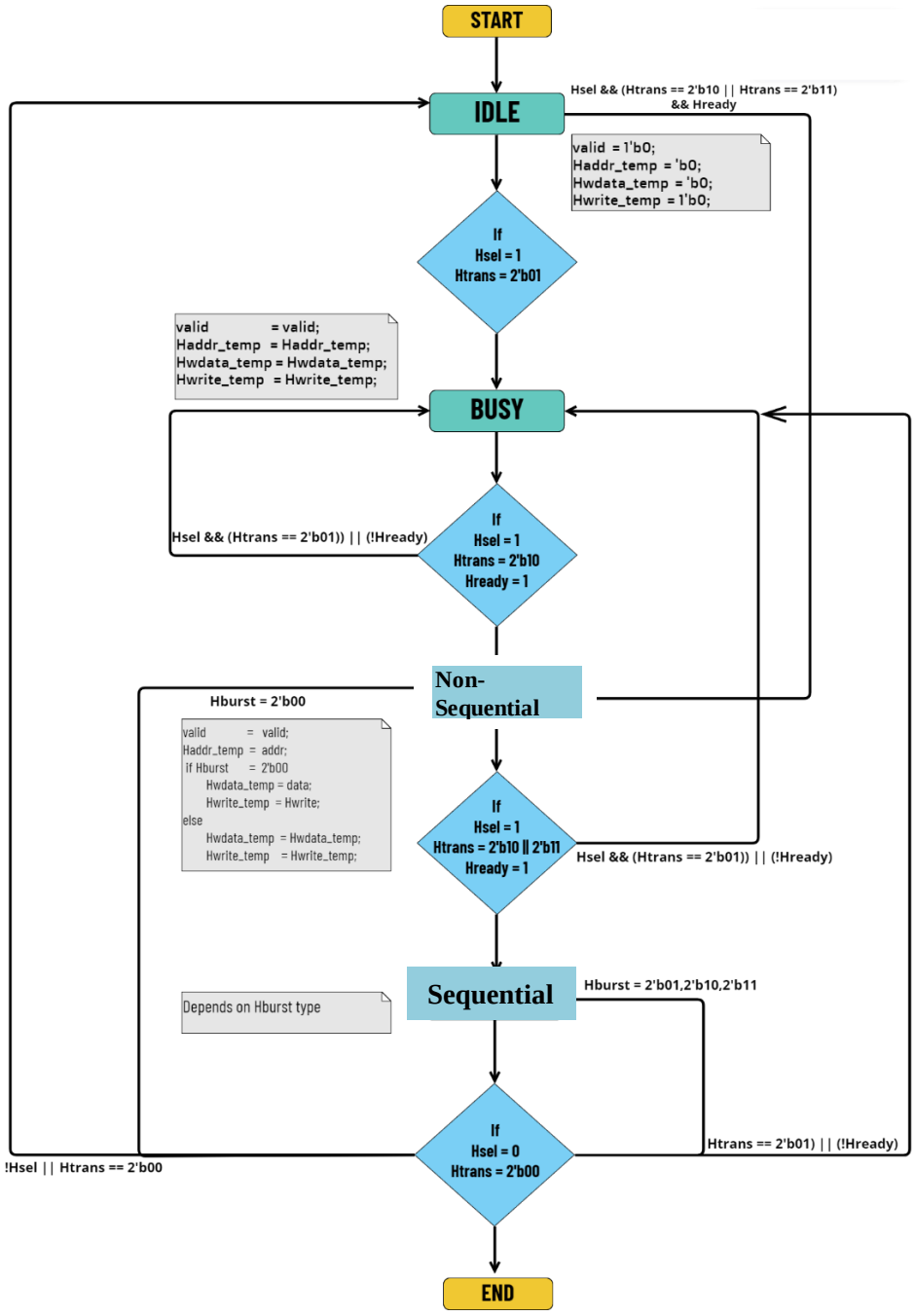
Fig.2.2 AHB SLAVE

The slave responds to the Transfer initiated by the Master. The slave uses its Hselx signal or input to perform its operation.

The slave also back signal to the Master:

- The completion and extension of Transfer.
- The success and failure of the Transfer.
- Any error occurred during the operation.

9. STATE DIAGRAM FOR AHB



FIFO CONTROLLER

1. Introduction

The Advanced Microcontroller Bus Architecture (AMBA) defines standardized protocols for on-chip communication within System-on-Chip (SoC) designs. The Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB) cater to different performance and power requirements. An APB2AHB bridge facilitates communication between these buses, ensuring seamless data exchange between high-performance AHB masters and low-power APB peripherals.

This report explores the potential of employing First-In-First-Out (FIFO) buffers within APB2AHB bridges, analyzing their benefits, implementation considerations, and trade-offs.

2. Challenges in APB2AHB Bridge Design:

Bridging between AHB and APB presents several challenges:

Speed Mismatch: AHB typically operates at higher frequencies than APB, leading to potential data transfer rate discrepancies.

Protocol Conversion: AHB and APB employ different signaling protocols, requiring translation for seamless communication.

Synchronization: Asynchronous clock domains of AHB and APB necessitate careful synchronization mechanisms to avoid timing violations.

3. Role of FIFOs in APB2AHB Bridges:

FIFOs offer a valuable solution to address these challenges:

Decoupling Clock Domains: By buffering data between AHB and APB domains, FIFOs decouple their clock frequencies, enabling asynchronous data transfers without compromising integrity.

Adapting Data Rates: FIFOs can act as temporary storage, allowing data accumulation from the slower APB and subsequent transmission at the faster AHB rate, smoothing out potential speed mismatches.

Handling Burst Transfers: FIFOs can efficiently handle AHB burst transfers by buffering data chunks and releasing them to the APB in smaller, manageable units.

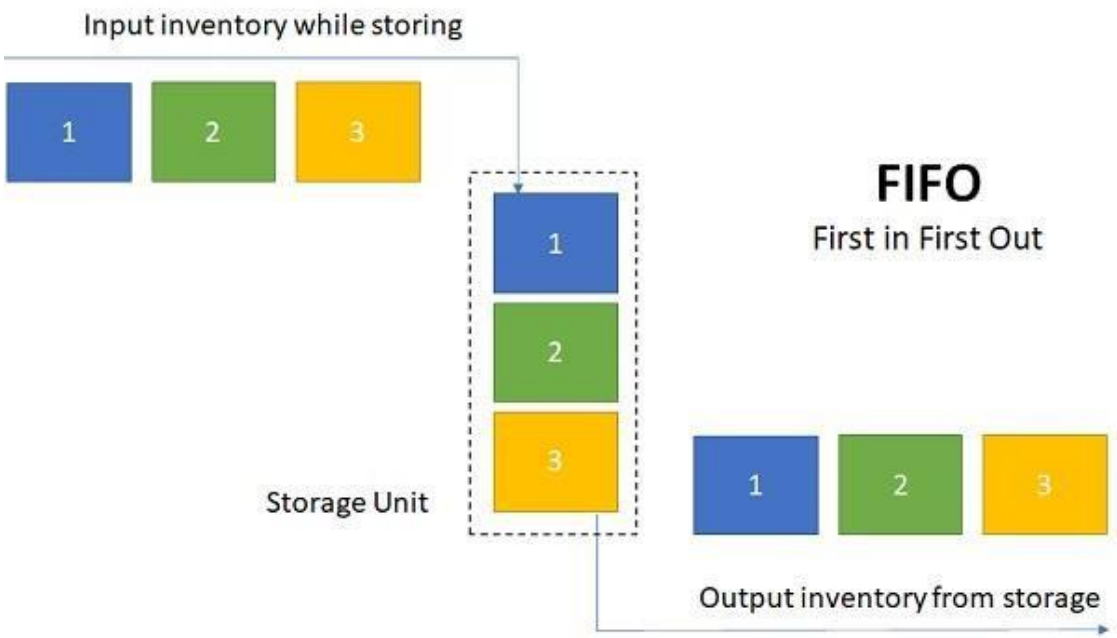
4. FIFO Implementation Considerations:

Integrating FIFOs in an APB2AHB bridge requires careful consideration of several factors:

FIFO Size: The size of the FIFO directly impacts its buffering capacity and ability to handle data rate differences. Choosing an appropriate size is crucial to prevent overflows and underflows.

Read/Write Latency: Introducing FIFOs adds additional latency to data transfers. Optimizing FIFO access mechanisms and minimizing read/write operations can help mitigate this impact.

Power Consumption: FIFOs introduce additional logic and consume power. Evaluating the trade-off between performance benefits and power overhead is essential.



Fi.2.3 A simple FIFO

5. **Benefits of Using FIFOs:**

Integrating FIFOs in APB2AHB bridges offers several advantages:

Improved Performance: FIFOs mitigate data rate mismatches potentially enhancing overall system throughput.

Simplified Design: By decoupling clock domains and handling burst transfers, FIFOs can simplify bridge design and reduce complexity.

Enhanced Reliability: Buffering data within FIFOs can help absorb transient errors and improve system robustness.

6. **Trade-offs and Challenges:**

Despite their benefits, using FIFOs in APB2AHB bridges comes with trade-offs and challenges:

Increased Latency: As discussed earlier, FIFOs introduce additional latency to data transfers, which may need to be carefully managed depending on application requirements.

Power Consumption: The inclusion of FIFOs adds to the overall power consumption of the bridge, requiring careful consideration in power-constrained designs.

Design Complexity: Integrating FIFOs increases the design complexity of the bridge, potentially requiring additional hardware resources and verification efforts.

FIFOs offer a valuable technique for addressing challenges in APB2AHB bridge design. Careful analysis of their benefits, trade-offs, and implementation considerations is crucial for determining their suitability in specific applications. By effectively utilizing FIFOs, designers can achieve efficient and reliable communication between AHB and APB domains within SoCs.

APB PROTOCOL

The Advanced Peripheral Bus (APB) protocol, a key component of the Advanced Microcontroller Bus Architecture (AMBA) suite, is specifically designed for low-power communication between peripherals and processors within System-on-Chip (SoC) architectures. This report provides a detailed exploration of the APB protocol, delving into its functionalities, features, and implementation aspects.

1. **Introduction**

   SoCs integrate various processing units, memory controllers, and peripherals, necessitating efficient on-chip communication mechanisms. AMBA addresses this need by offering standardized buses, including APB, optimized for low-power peripheral interactions.

   APB prioritizes low power consumption and simplicity, making it ideal for connecting slower peripherals that don't require high-bandwidth data transfers**.**

2. **APB Protocol Overview:**

   APB operates on a master-slave architecture, where one master device initiates data transfers and multiple slave devices respond to these requests. The protocol defines various signal types for communication:

   Address signals: Specify the memory location or peripheral register being accessed.

   Control signals: Indicate the type of transfer (read/write), size of data being transferred, and error conditions.

Data signals: Carry the actual data being exchanged between master and slave.

3. **Key Features of APB:**

Low Power Consumption: APB employs a simple design with minimal clocking overhead, minimizing power consumption compared to high-performance buses.

Scalability: The protocol adapts to diverse SoC configurations with varying numbers of masters and slaves.

Ease of Use: The simple design and standardized interface simplify the integration of peripherals into SoCs.

Error Signaling: APB incorporates mechanisms to detect and report errors during transfers, ensuring data integrity.

4. **APB Transaction Process**

A typical APB transaction involves the following steps:

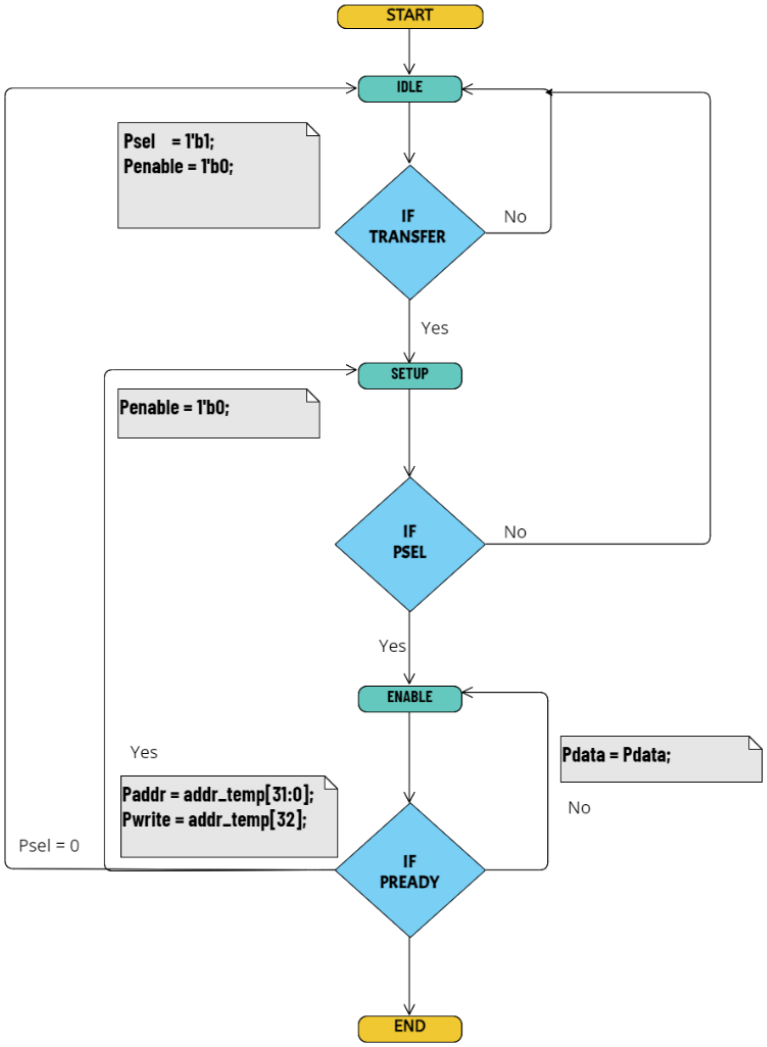
Master initiates transfer: The master asserts address and control signals on the bus, specifying the target slave, transfer type, and data size.

Slave selection: The arbiter grants bus access to the appropriate slave based on a predefined priority scheme.

Data transfer: The master drives data onto the bus for write operations or receives data from the slave for read operations.

Transfer completion: The slave acknowledges the transfer completion by asserting appropriate response signals.

5. STATE DIAGRAM FOR APB:



6. **States of APB:**

   Idle: This is the initial state where the bus is inactive and no transfer is ongoing.

   Address Setup: The master asserts the address and control signals on the bus, specifying the target slave, transfer type (read/write), and data size.

   Data Setup: The master drives data onto the bus (write operations) or prepares to receive data (read operations).

   Data Access: The slave processes the data, either reading from or writing to its internal registers.

   Data Transfer: The data is transferred between the master and slave.

   Response Wait: The master waits for the slave's response indicating transfer completion.

   Response Received: The slave asserts response signals, acknowledging successful transfer or reporting any errors.

   Post-Transfer: The bus returns to the idle state, and the master and slave can prepare for subsequent transactions.

7. **Transitions:**

   Idle to Address Setup: The master initiates a transfer by asserting address and control signals, triggering the transition.

   Address Setup to Data Setup: Once the slave receives the address and control information, the transition occurs, allowing the master to prepare data.

   Data Setup to Data Access: After the master prepares data, the transition enables the slave to process it.

   Data Access to Data Transfer: Once the slave finishes processing, the actual data transfer takes place.

   Data Transfer to Response Wait: Upon completing the transfer, the transition occurs, and the master awaits the slave's response.

Response Wait to Response Received: The slave asserts response signals, triggering the transition and indicating transfer completion.

Response Received to Idle: After receiving the response, the bus returns to the idle state, ready for new transactions.

8. PIN DIAGRAM FOR APB

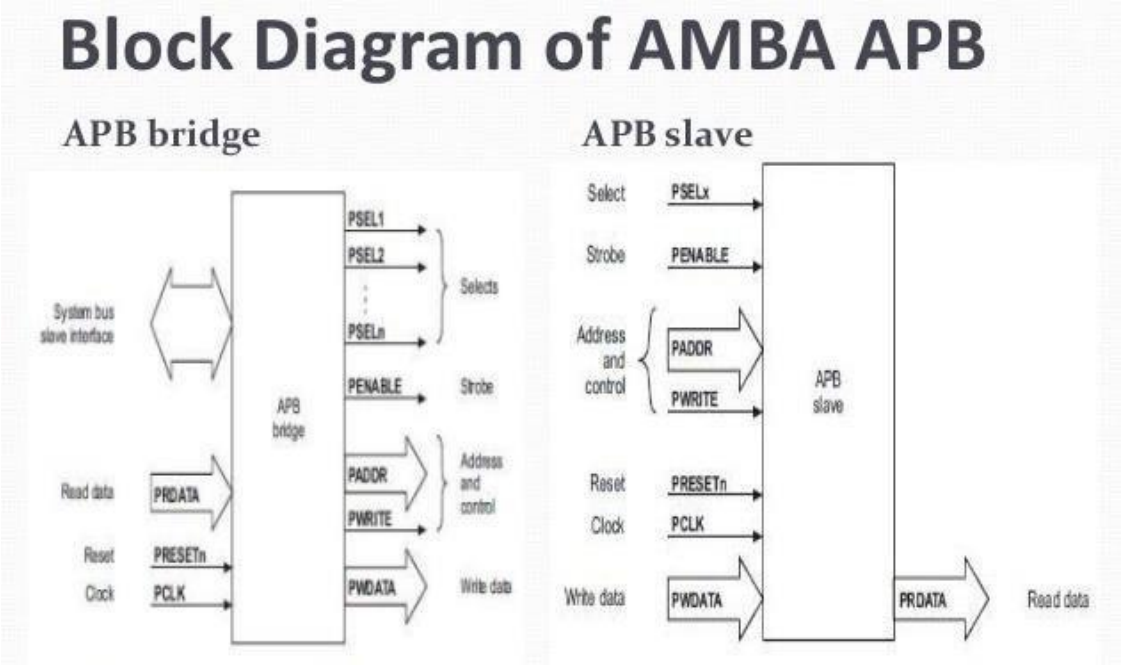


Fig 2.5 Block diagram for APB

Understanding the APB state diagram is crucial for comprehending the protocol's operation and analyzing potential timing issues during data transfers. It also aids in designing and verifying APB interfaces within SoCs.

SYSTEM DESIGN AND IMPLEMENTATION

1. Design Information

We are implementing the Top module by taking the instance of the APB master and AHB slave. For the Bridge, we have defined asynchronous FIFO on the Vivado platform. The verification is done using the Universal Verification Methodology. For implementation, we are using 3 IPs using Verilog hardware description language.

The work is focused on these steps:

a. AHB slave
b. APB master
c. FIFO controller
d. Top Module
e. Wrap Top

BASIC BLOCK DIAGRAM

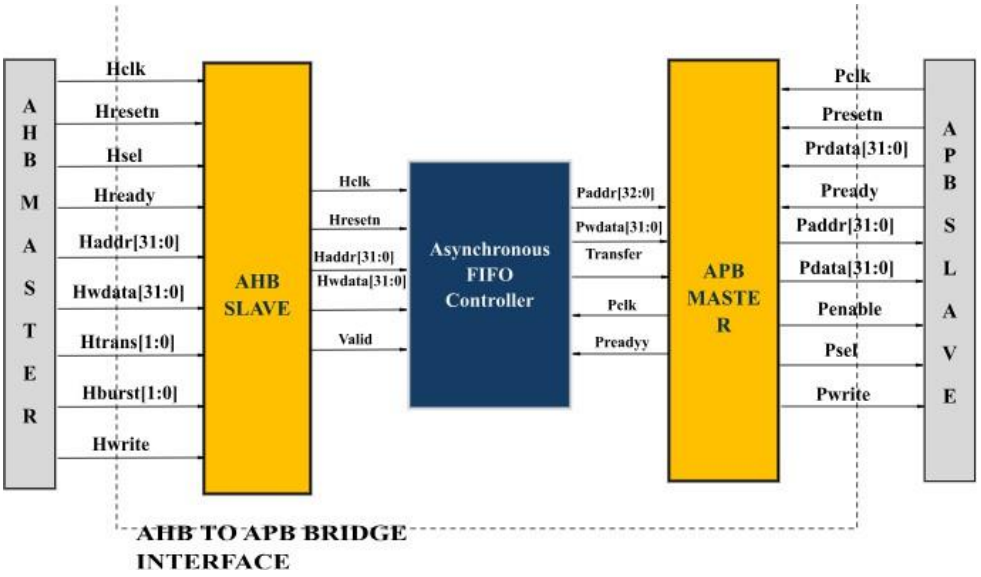


Fig.3.1 AHB To APB Bridge Interface

TOP BLOCK DIAGRAM

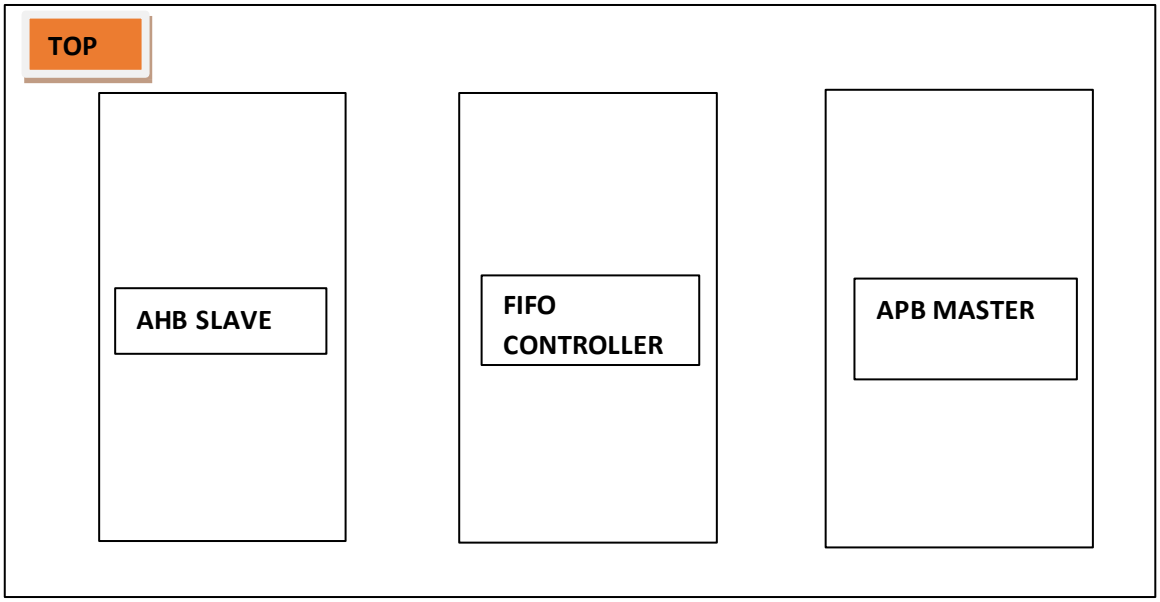


Fig.3.2 Top model for AHB2APB Design

SIGNAL DESCRIPTION

GLOBAL SIGNAL	AHB INPUT SIGNALS	AHB OUTPUT SIGNALS	FIFO INPUT SIGNALS	FIFO OUTPUT SIGNALS	APB INPUT SIGNAL	APB OUTPUT SIGNALS
Hclk	Htrans	Haddr_temp	Haddr_temp	Data_temp	Data_temp	Psel
Pclk	Haddr	Hwdata_temp	Hwdata_temp	Addr_temp	Addr_temp	Paddr
Hresetn	Hwdata	Valid	Valid	transfer	Prdata	Pdata
Presetn	Hburst	Hwrite_temp	Hwrite_temp	full	transfer	Rdata_temp
	Hwrite		Pready		Pready	Pwrite
	Hsel					Penable
	Hready					

1. **GLOBAL SIGNALS**
   a. **Hclk:** The primary clock for the design. This is used to drive the AHB Slave of the design. Here we are using 100MHz.
   b. **Pclk:** This clock is used to drive the APB master of the design. We are using a clock of 50MHz.
   c. Hresetn: This is used to synchronize the AHB slave. This is an active low reset for this design.
   d. Presetn: This is used to synchronize the APB slave. This is also an active low reset.
2. **AHB INPUT SIGNALS**
   a. Htrans[1:0]: This is used to indicate the current transfer, which can be used as IDEAL, BUSY, ADDRESS(NON-SEQ), and TRANSFER(SEQ).
   b. Haddr[31:0]: This is the 32-bit system address bus.
   c. Hwdata[31:0]: The write data bus is used to transfer data from the master to the bus slaves during the write operation. The data width of 32 bits is used here.

d. **Hburst[1:0]:** The signal used to select the burst mode, which is used as SINGLE TRANSFER, INCR(Infinite Incremental), INCR4, INCR8.

e. **Hwrite:** The signal used to perform read/write operation. WRITE operation is performed when Hwrite == 1 and when Hwrite == 0 READ operation is performed.

f. **Hsel:** This is used as an enable signal. The main task of this is to select the master in which we must perform the operation in multiple master-slave configurations. But in our design, we have only one master so we are using this as an enable signal.

g. **Hready:** signal used to check whether our AHB slave is ready to do operation or not.

3. **AHB OUTPUT SIGNALS**

a. Haddr_temp: To register the address for pipelining or APB.

b. Hwdata_temp: To register the data for pipelining or APB.

c. Valid: It is a condition checker. It checks whether the data and address packet are valid or not. If idle state, then valid is zero. For Busy and address the past value will be stored. And last for the Transfer valid is One.

d. Hwrite_temp: To register the write signal before sending it to FIFO.

4. **FIFO INPUT SIGNALS**

a. Haddr_temp: To register the address in input.

b. Hwdata_temp: To register the data in the input.

c. Hwrite_temp: To register the write signal.

d. **Valid:** It is a condition checker. It checks whether the data and address packet are valid or not. If idle state, then valid is zero. For Busy and address the past value will be stored. And last for the Transfer valid is One.

e. **Pready:** To check whether APB master is ready to take signal or not.

5. **FIFO OUTPUT SIGNALS**

   a. **Data_temp:** Input registered data for APM Master.

   b. **Addr_temp:** Input registered address for APB Master.

   c. **Transfer:** To check the condition for packet transfer to APB master.

   d. **Full:** When full is high the fifo won't take any new data/addr.

6. **APB INPUT SIGNAL**

   a. **Data_temp:** Input registered data from FIFO.

   b. **Addr_temp:** Input registered address from FIFO.

   c. **Prdata:** To read the data.

   e. **Transfer:** This is used to define the FSM status for the APB.

   d. **Pready:** This is used as an enable signal in APB master.

7. **APB OUTPUT SIGNALS**

   a. **Psel:** This signal is used to select the slave.

   b. **Paddr:** This signal contains the Address for the slave in which data is going to be written.

   c. **Pdata:** This signal contains Data that is going to be written on the selected address in slave.

   d. **rdata_temp:** Temporary register for read data.

   e. **Pwrite:** Signal which defines which operation is going to be performed i.e. read/write based on 0/1.

   f. **Penable:** This is the signal which enables the Peripherals.

MODES OF OPERATION(AHB)

The Advanced High-performance Bus (AHB) protocol offers several modes of operation to optimize data transfer efficiency and cater to diverse communication scenarios within SoCs. These modes address different requirements for speed, flexibility, and burst transfer handling. In our project, we are using the following modes:

- **Single Transfer:**
- This is the most basic mode, involving a single address, control, and data transfer per transaction.
- Suitable for simple data exchanges where high throughput is not critical.
- **Incremental Burst:**
- Enables transferring multiple consecutive data items using a single address and control phase.
- The address automatically increments for each subsequent data transfer within the burst.
- Offers improved efficiency compared to individual transfers for large data movements.

CODE

1. TOP WRAPPER:

```
module top_wrapper(
                input       Pclk,
                input       Pready,
                input [31:0]  HADDR_TEMP,
                input [1:0 ]  HBURST,
                input       Hclk,
                input       HREADY,
                input       Hresetn,
                input       HSEL,
                input [1:0 ]  HTRANS,
                input [31:0]  HWDATA,
                input       HWRITE,
                input       Presetn,
                input       transfer,
                input       write_enable,
                output  [31:0]    Paddr,
                output  [31:0]    Pdata,
                output      Penable,
                output      Psel,
                output      Pwrite,
                output  [31:0]    rdata_temp,
                output      rempty,
                output      wfull
                );



        top_bridge  DUT0 (
                    //input [31:0] wdata,
                      .HSEL(HSEL),
                      .HWRITE(HWRITE),
                      .HBURST(HBURST),
                      .HTRANS(HTRANS),
                      .HREADY(HREADY),
                      .write_enable(write_enable),
                      .Pready(Pready),
                      .HADDR_TEMP(HADDR_TEMP),
                      .HWDATA(HWDATA),      // Input data - data to be written
                    //input winc,               // write enable
                      .wclk(Hclk),
```

```
                .wrst_n(Hresetn),       // Write increment, write clock, write reset
                .transfer(transfer),
                .rclk(Pclk),
                .rrst_n(Presetn),       // Read increment, read clock, read reset
               //output [31:0] rdata,       // Output data - data to be read
                .wfull(wfull),                  // Write full signal
                .rempty(rempty),
                .Psel(Psel),            // Slave select
                .Paddr(Paddr), // Address to APB slave
                .Pdata(Pdata), // Data to write to the slave
                .rdata_temp(rdata_temp), // Data read from the slave
                .Pwrite(Pwrite),            // Write control signal
                .Penable(Penable)
               //output  [31:0] memory_data
  );


   endmodule
```

2. TOP BRIDGE:

```
`timescale  1ns / 1ps


module top_bridge(
                //input [31:0] wdata,
                input HSEL,
                input HWRITE,
                input [1:0] HBURST,
                input [1:0] HTRANS,
                input HREADY,
                input write_enable,
                input Pready,
                input [31:0] HADDR_TEMP,
                input [31:0] HWDATA,        // Input data - data to be written
                //input winc,                 // write enable
                input wclk,
                input wrst_n,       // Write increment, write clock, write reset
                input transfer,
                input rclk,
                input rrst_n,       // Read increment, read clock, read reset
                //output [31:0] rdata,       // Output data - data to be read
                output wfull,                  // Write full signal
                output rempty,
                output  Psel,            // Slave select
                output  [31:0] Paddr, // Address to APB slave
                output  [31:0] Pdata, // Data to write to the slave
                output  [31:0] rdata_temp, // Data read from the slave
```

```
                output  Pwrite,          // Write control signal
                output  Penable
                //output  [31:0] memory_data
);

  wire  [31:0] rdata;
  wire [31:0] ADDR_TEMP;
  wire [31:0]   HWDATA_TEMP;
  wire         VALID;
  wire       HWRITE_TEMP;
  wire [31:0] address_out;
  //wire  [31:0] Paddr;
  //wire  [31:0] Pdata;
  //wire   Pwrite;
  //wire   Penable;
 // wire  [31:0] memory_data;
  //wire  [31:0] rdata_temp;


       ahb_slave DUT(
                    .HRESETn(wrst_n),
                    .HCLK(wclk),
                    .HSEL(HSEL),
                    .HADDR(HADDR_TEMP),
                    .HWRITE(HWRITE),
                    .HBURST(HBURST),
                    .HTRANS(HTRANS),
                    .HREADY(HREADY),
                    .HWDATA(HWDATA),
                    //.HRDATA(rdata_temp),
                    .HADDR_TEMP(ADDR_TEMP),
                    .HWDATA_TEMP(HWDATA_TEMP),
                    .VALID(VALID),
                    .HWRITE_TEMP(HWRITE_TEMP)
                );

      top DUT1 (
            .address(ADDR_TEMP),
            .wdata(HWDATA_TEMP),       // Input data - data to be written
            .winc(HWRITE_TEMP),                // write enable
            .wclk(wclk),
            .wrst_n(wrst_n),       // Write increment, write clock, write reset
            .rinc(Penable),
            .rclk(rclk),
            .rrst_n(rrst_n),       // Read increment, read clock, read reset
            .rdata(rdata),       // Output data - data to be read
```

```
                .wfull(wfull),                  // Write full signal
                .rempty(rempty),
                .address_out(address_out)

    );

        APB_MASTER DUT2(
    // Global signals
                .Presetn(rrst_n),
                .Pclk(rclk),

    // Controller inputs
                .addr_temp(address_out),
                .data_temp(rdata),
                //.Prdata(memory_data),
                .transfer(transfer),
                .Pready(Pready),
                .write_enable(write_enable),

    // Output signals
                .Psel(Psel),
                .Paddr(Paddr),
                .Pdata(Pdata),      // output
                .rdata_temp(rdata_temp),
                .Pwrite(Pwrite),
                .Penable(Penable)
);
     endmodule
```

3. AHB CODE:

```
`timescale  1ns / 1ps

module ahb_slave(
                input           HRESETn,
                input           HCLK,
                input           HSEL,
                input [31:0]      HADDR,
                input           HWRITE,
                input [1:0]       HBURST,
                input [1:0]       HTRANS,
                input           HREADY,
                input [31:0]      HWDATA,
                //input [31:0]       HRDATA,
                output reg [31:0]   HADDR_TEMP,
                output reg [31:0]   HWDATA_TEMP,
                output reg        VALID,
```

```
            output reg          HWRITE_TEMP);

parameter IDLE = 2'b00;
parameter BUSY = 2'b01;
parameter NONSEQ = 2'b10;
parameter SEQ = 2'b11;

reg [1:0] present_state, next_state;
reg [31:0] addr, data;

// Capture HADDR and HWDATA on the clock edge
always @(posedge HCLK) begin
   if (~HRESETn) begin
      addr <= 0;
      data <= 0;
   end else begin
      addr <= HADDR;
      data <= HWDATA;
   end
end

// State transition logic
always @(posedge HCLK) begin
   if (~HRESETn)
      present_state <= IDLE;
   else
      present_state <= next_state;
end

// Next state logic
always @(*) begin
   case (present_state)
      IDLE: begin
         if (HSEL && HTRANS == 2'b01)
            next_state = BUSY;
         else if (HSEL && (HTRANS == 2'b10 || HTRANS == 2'b11) && HREADY)
            next_state = NONSEQ;
         else
            next_state = IDLE;
      end

      BUSY: begin
         if (HSEL && HTRANS == 2'b10 && HREADY)
            next_state = NONSEQ;
         else if (HSEL && HTRANS == 2'b01 || !HREADY)
            next_state = BUSY;
```

```
            else
                next_state = IDLE;
        end

        NONSEQ: begin
            if (HSEL && (HTRANS == 2'b10 || HTRANS == 2'b11) && HREADY)
                next_state = SEQ;
            else if (HSEL && HTRANS == 2'b01 || !HREADY)
                next_state = BUSY;
            else
                next_state = IDLE;
        end

        SEQ: begin
            if (~HSEL || HTRANS == 2'b00)
                next_state = IDLE;
            else if (HSEL && HTRANS == 2'b01 || !HREADY)
                next_state = BUSY;
            else begin
                case (HBURST)
                    2'b00: next_state = NONSEQ; // single transfer
                    2'b01: next_state = SEQ;   // increment transfer
                    2'b10: if (HADDR_TEMP < addr + 4)
                                next_state = SEQ;
                           else
                                next_state = NONSEQ;
                    2'b11: if (HADDR_TEMP < addr + 8)
                                next_state = SEQ;
                           else
                                next_state = NONSEQ;
                    default: next_state = IDLE;
                endcase
            end
        end
    endcase
end

// Output logic
always @(posedge HCLK) begin
    if (~HRESETn) begin
        VALID <= 1'b0;
        HADDR_TEMP <= 0; // Initialize to 0
        HWDATA_TEMP <= 0;
        HWRITE_TEMP <= 0;
    end else begin
        case (present_state)
```

```
        IDLE: begin
            VALID <= 1'b0;
            HADDR_TEMP <= 0; // Reset to 0 in IDLE state
            HWDATA_TEMP <= 0;
            HWRITE_TEMP <= 0;
        end

        BUSY: begin
            VALID <= 1'b0; // No valid data in BUSY state
            HADDR_TEMP <= HADDR_TEMP; // Hold address
            HWDATA_TEMP <= HWDATA_TEMP;
            HWRITE_TEMP <= HWRITE_TEMP;
        end

        NONSEQ: begin
            VALID <= 1'b1; // Valid data in NONSEQ state
            HADDR_TEMP <= addr; // Capture the current address
            HWDATA_TEMP <= data; // Capture the current data
            HWRITE_TEMP <= HWRITE; // Capture the write signal
            $display("NONSEQ: HADDR_TEMP = %0h, HADDR = %0h",
HADDR_TEMP, addr); // Debugging
        end

        SEQ: begin
            VALID <= 1'b1; // Valid data in SEQ state
            case (HBURST)
                2'b00: begin // Single transfer
                    HADDR_TEMP <= addr; // No increment
                    HWDATA_TEMP <= data;
                    HWRITE_TEMP <= HWRITE;
                end

                2'b01: begin // Increment by 1
                    HADDR_TEMP <= HADDR_TEMP + 1; // Increment address
                    HWDATA_TEMP <= data;
                    HWRITE_TEMP <= HWRITE;
                end

                2'b10: begin // Increment by 4
                    if (HADDR_TEMP < addr + 4) begin
                        HADDR_TEMP <= HADDR_TEMP + 1; // Increment address
                    end else begin
                        HADDR_TEMP <= addr; // Reset to base address
                    end
                    HWDATA_TEMP <= data;
                    HWRITE_TEMP <= HWRITE;
```

```
                end

                2'b11: begin // Increment by 8
                    if (HADDR_TEMP < addr + 8) begin
                        HADDR_TEMP <= HADDR_TEMP + 1; // Increment address
                    end else begin
                        HADDR_TEMP <= addr; // Reset to base address
                    end
                    HWDATA_TEMP <= data;
                    HWRITE_TEMP <= HWRITE;
                end

                default: begin
                    HADDR_TEMP <= 0;
                    HWDATA_TEMP <= 0;
                    HWRITE_TEMP <= 0;
                end
            endcase
            $display("SEQ: HADDR_TEMP = %0h, HADDR = %0h", HADDR_TEMP,
    addr); // Debugging
        end
      endcase
   end
end

    endmodule
```

4. TOP:

```
`timescale  1ns / 1ps

module top #(parameter DSIZE = 32,parameter ASIZE = 5)(
   input [DSIZE-1:0] wdata,        // Input data - data to be written
   input [DSIZE-1:0] address,
   input winc,                     // write enable
   input wclk,
   input wrst_n,       // Write increment, write clock, write reset
   input rinc,
   input rclk,
   input rrst_n,       // Read increment, read clock, read reset
   output [DSIZE-1:0] rdata,       // Output data - data to be read
   output wfull,                   // Write full signal
   output rempty,                   // Read empty signal
   output [31:0] address_out
   );

    wire [ASIZE-1:0] waddr;
```

```
  wire [ASIZE-1:0] raddr;
  wire [ASIZE:0] wptr;
  wire [ASIZE:0] rptr;
  wire [ASIZE:0] wq2_rptr;
  wire [ASIZE:0] rq2_wptr; //wq2_rptr: from read to write rq2_wptr : from write to
 read

synch #(ASIZE) sync_r2w (       // Read pointer syncronization to write clock
 domain
    .q2(rq2_wptr),    // read to write
    .din(rptr),
    .clk(wclk),
    .rst_n(wrst_n)
);

synch #(ASIZE) sync_w2r (       // Write pointer syncronization to read clock domain
    .q2(wq2_rptr),    // write to read
    .din(wptr),
    .clk(rclk),
    .rst_n(rrst_n)
);


memory #(DSIZE, ASIZE) fifomem(    // Memory module
    .rdata(rdata),
    .wdata(wdata),
    .address(address),
    .waddr(waddr),
    .raddr(raddr),
    .wclk_en(winc),
    .wfull(wfull),
    .wclk(wclk),
    .rclk(rclk),
    .rclk_en(rinc),
    .rempty(rempty),
    .address_out(address_out)
);

read_operation #(ASIZE) rptr_empty(         // Read pointer and empty signal
 handling
    .rempty(rempty),
    .raddr(raddr),
    .rptr(rptr),
    .wq2_rptr(wq2_rptr),
    .rinc(rinc),
    .rclk(rclk),
```

```
    .rrst_n(rrst_n)
  );

  write_operation #(ASIZE) wptr_full(          // Write pointer and full signal handling
    .wfull(wfull),
    .waddr(waddr),
    .wptr(wptr),
    .rq2_wptr(rq2_wptr),
    .winc(winc),
    .wclk(wclk),
    .wrst_n(wrst_n)
  );
   endmodule
```

5. MEMORY CODE:

```
`timescale  1ns / 1ps


module memory #(parameter DSIZE = 32,parameter ASIZE = 5)(
  input [DSIZE-1:0] wdata,
  input [ASIZE-1:0] waddr,
  input [ASIZE-1:0] raddr,
  input [31:0] address,
  input wclk_en,
  input wfull,
  input wclk,
  input rclk,
  input rclk_en,
  input rempty,
  output reg [DSIZE-1:0] rdata,
  output reg [31:0] address_out
   );

   localparam DEPTH = 1 << ASIZE;    //declaring memory locations
   reg [DSIZE-1:0] mem [0:DEPTH-1];     // initating memory locations

   /// performing write operation
    always @(posedge wclk)
     begin
          if (wclk_en && !wfull)
            begin
              mem[wdata] = wdata; // Write data
              $display("data=%0h,address=%0h",wdata,waddr);
             end
//           else
//              begin
```

```
//                    mem[waddr] = 0;
//                  end
      end
       //$display("mem= %0p,waddr=%h,full=%0b",mem,waddr,wfull);

      /// performing read operation
      always @(*)
      begin
       if(rclk_en && !rempty)
         begin
           rdata=mem[raddr];
           address_out = raddr+1;
            $display("data output=%0h,address output=%0h",rdata,raddr);
         end
        else
          begin
            rdata=0;
          end
      end
     endmodule
```

6. SYNCH:

```
`timescale  1ns / 1ps


module synch #(parameter ASIZE =4)(
   input [ASIZE:0] din,
   input clk,
   input  rst_n,
   output reg [ASIZE:0] q2
   );

   reg [ASIZE:0] q1;

   always @(posedge clk or negedge rst_n) begin
      if (!rst_n)
         {q2, q1} <= 2'b0;
      else
         {q2, q1} <= {q1, din};
   end

    endmodule
```

7. READ-OPERATION

```
`timescale  1ns / 1ps
```

```
module read_operation#(parameter SIZE = 4)(
    input [SIZE :0] wq2_rptr,      // write to read  this is also gray pointer which is
     coming from write
    input rinc,
    input rclk,
    input rrst_n,
    output reg rempty,
    output [SIZE-1:0] raddr,
    output reg [SIZE :0] rptr    // this is sending to write
    );

    wire [SIZE:0] rgray_next;   // Next read pointer in gray and binary code
    wire rempty_next;

    // fifo empty condition
    assign rempty_next = (rgray_next == wq2_rptr);
    always @(posedge rclk or negedge rrst_n) begin
        if (!rrst_n)
            rempty <= 1'b0;   // FIFO starts empty     // rempty <= 1'b1 previously
        else
            rempty <= rempty_next;  // Update empty flag
    end


    reg [SIZE:0] rbin;                    // Binary read pointer
    wire [SIZE:0] rbin_next;

  // inceremented
    assign rbin_next  = rbin + (rinc & ~rempty);

    always @(posedge rclk or negedge rrst_n) begin
        if (!rrst_n) begin
            rbin  <= 0;    // Reset Binary Read Pointer
        end
        else begin
          //@(negedge rclk)
            rbin  <= rbin_next;   // Update Binary Read Pointer
        end
     end

    assign raddr = rbin[SIZE-1:0];

// converting  binary to gray

    assign rgray_next = (rbin_next >> 1) ^ rbin_next;
```

```
    always @(posedge rclk or negedge rrst_n) begin
    if (!rrst_n) begin
        rptr  <= 0;    // Reset Binary Read Pointer
    end
    else begin
        rptr  <= rgray_next;   // Update Binary Read Pointer
    end
  end

endmodule
```

8. WRITE-OPERATION:

```
`timescale  1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 01.02.2025 17:56:02
// Design Name:
// Module Name: write_operation
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module write_operation#(parameter SIZE = 4)(
    input [SIZE:0] rq2_wptr,      //read to write this is also gray pointer which is coming
     from rptr_empty
    input winc,
    input wclk,
    input wrst_n,
    output reg wfull,
    output  [SIZE-1:0] waddr,
    output reg [SIZE:0] wptr   // this is sending to read
    );

    reg [SIZE:0] wbin;          // Binary Write Pointer
```

```
  wire [SIZE:0] wbin_next;     // Next value of Binary Write Pointer
  wire [SIZE:0] wgray_next;    // Next value of Gray-coded Write Pointer
  wire wfull_next;

 // incrementing
  assign wbin_next  = wbin + (winc & ~wfull);

    always @(posedge wclk or negedge wrst_n) begin
            if (!wrst_n) begin
               wbin  <= 0;    // Reset Binary Write Pointer
            end
            else begin
              //@(negedge wclk)
               wbin  <= wbin_next;   // Update Binary Write Pointer
            end
    end

// converting binary to gray

  assign waddr = wbin[SIZE-1:0];
  assign wgray_next = (wbin_next >> 1) ^ wbin_next;

     always @(posedge wclk or negedge wrst_n) begin
            if (!wrst_n) begin
               wptr  <= 0;
            end
            else begin
               wptr  <= wgray_next;
            end
     end

// fifo full condition

 assign wfull_next = rq2_wptr == {!wgray_next[SIZE:SIZE-1],wgray_next[SIZE-
   2:0]};

    always @(posedge wclk or negedge wrst_n) begin
         if (!wrst_n)
            wfull <= 1'b0;   // FIFO starts empty (not full)
         else
            wfull <= wfull_next;  // Update full flag
        end

   endmodule
```

9. APB CODE:

```
`timescale 1ns / 1ps


module APB_MASTER(
//global signals
input Presetn,
input Pclk,

//controller input
input [32:0] addr_temp,
input [31:0] data_temp,
input [31:0] Prdata,
```

```
input transfer,
input Pready,

//output signals
output reg    Psel,
output reg [31:0] Paddr,
output reg [31:0] Pdata,
output reg [31:0] rdata_temp,
output reg Pwrite,
output reg Penable
);

reg [2:0] present_state, next_state;

parameter idle  = 2'b01;
parameter setup  = 2'b10;
parameter enable = 2'b11;

always@(posedge Pclk, negedge Presetn)
begin
  if(!Presetn)
     present_state <= idle;
  else
     present_state <= next_state;
end

always@(*)
begin
case(present_state)
```

```
idle:
   begin
     if(!transfer)
        next_state = idle;
     else
        next_state = setup;
   end
setup:
    begin
     if(Psel)
     next_state = enable;
     else
     next_state = idle;
    end

enable:
    begin
     if(Psel)
     if(transfer)
         begin
          if(Pready)
          begin
            next_state = setup;
            end
          else
            next_state = enable;
         end
     else
       next_state = idle;
```

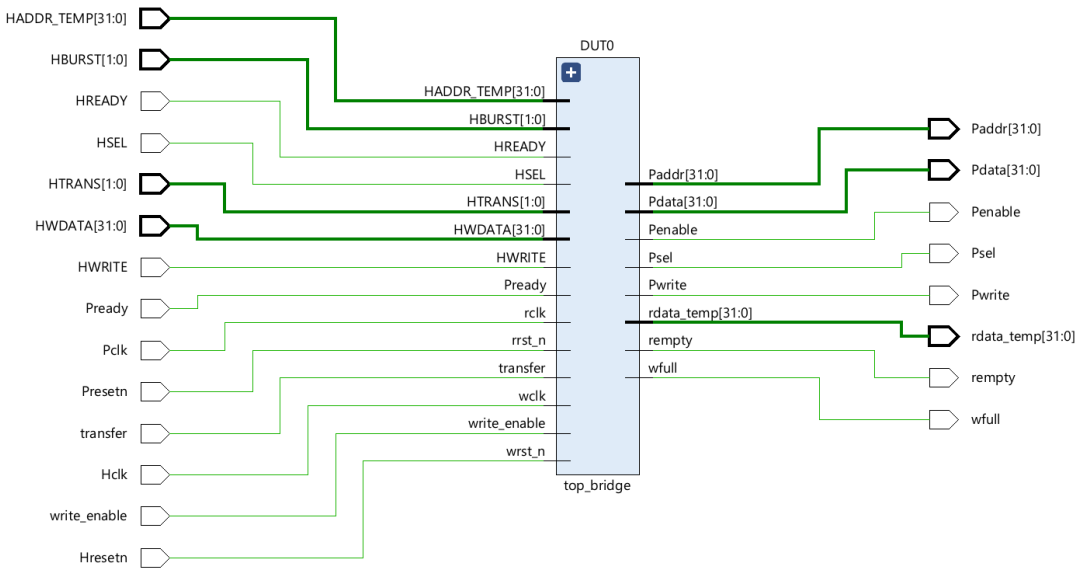
```
   end
endcase
end

always @(posedge Pclk)
begin
case(present_state)
idle:
begin
 Psel    = 1'b1;
Penable = 1'b0;
end
setup:
 Penable = 1'b0;
enable:
     begin
     if(Psel)
       Penable = 1'b1;
       if(transfer)
       begin
          if(Pready)
            begin
            Paddr = addr_temp[31:0];
            Pwrite = addr_temp[32];
            if(addr_temp[32])
              begin
                Pdata  = data_temp;
              rdata_temp  = 'b0;
              end
```

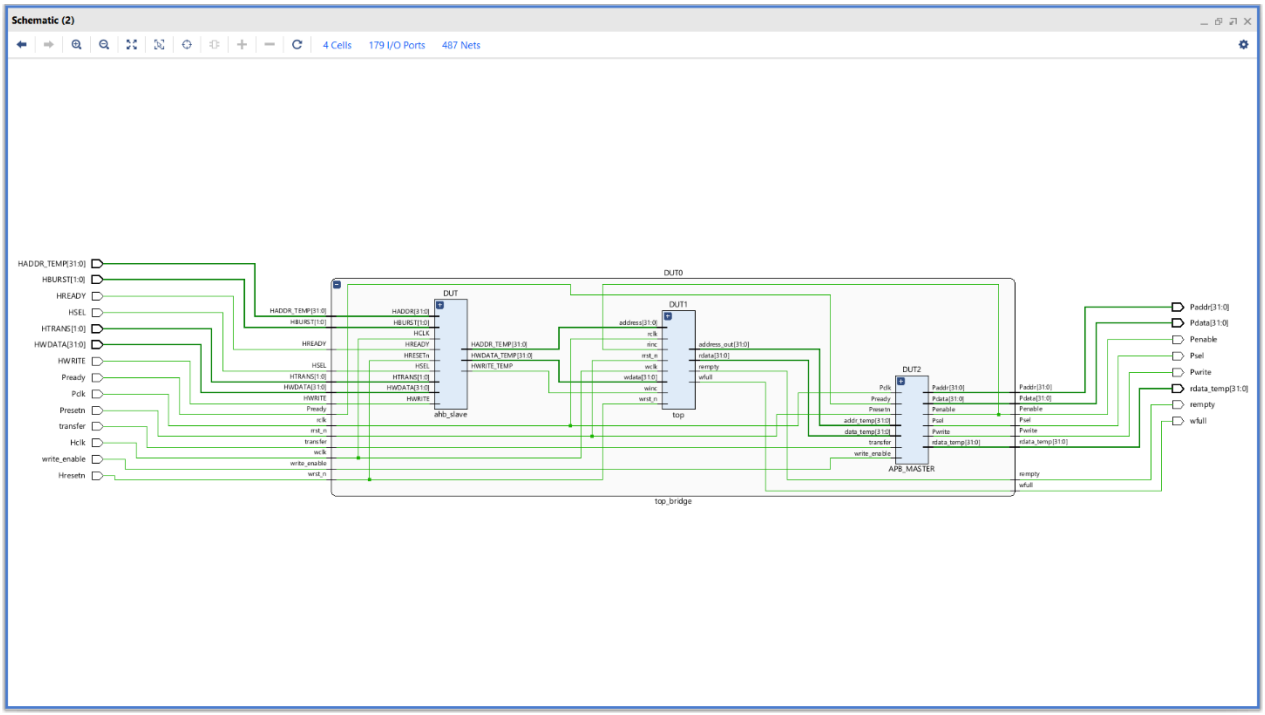
```
            else
            begin
              Pdata  = Pdata;
            rdata_temp  = Prdata;
            end
          end
          end
end
endcase
end
endmodule
```

SIMULATION RESULTS AND DESIGN

1. Elaborated Design (Wrapper)

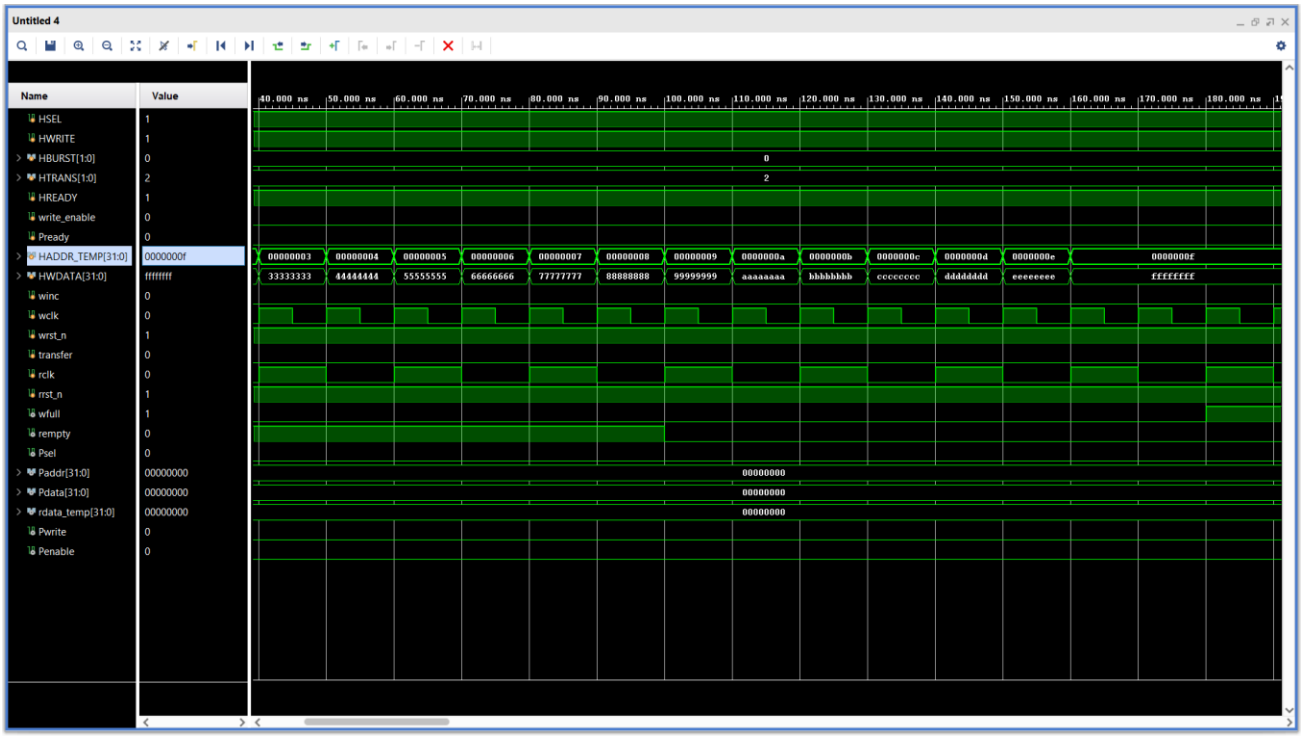


2. Elaborated Design (Schematic):

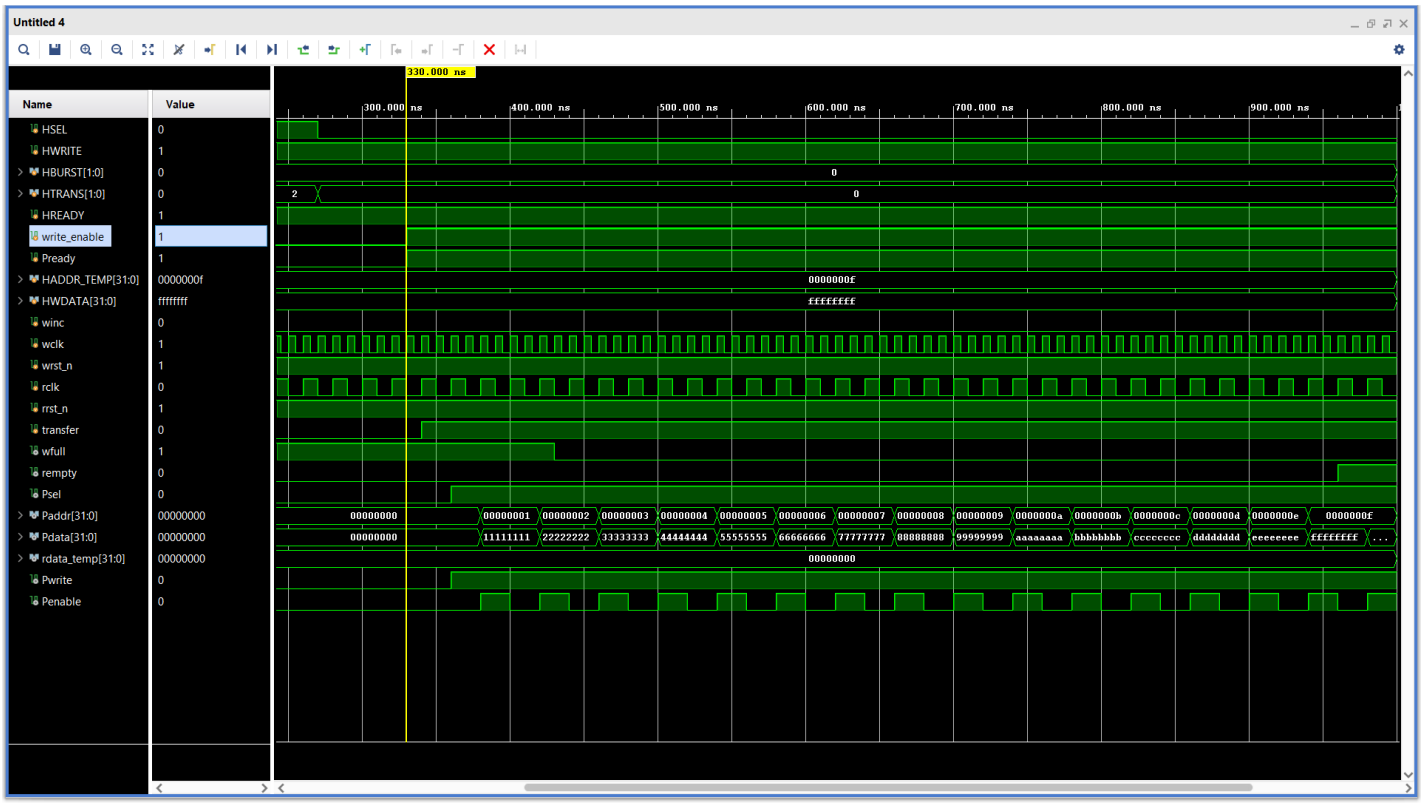


3. Simulation Result

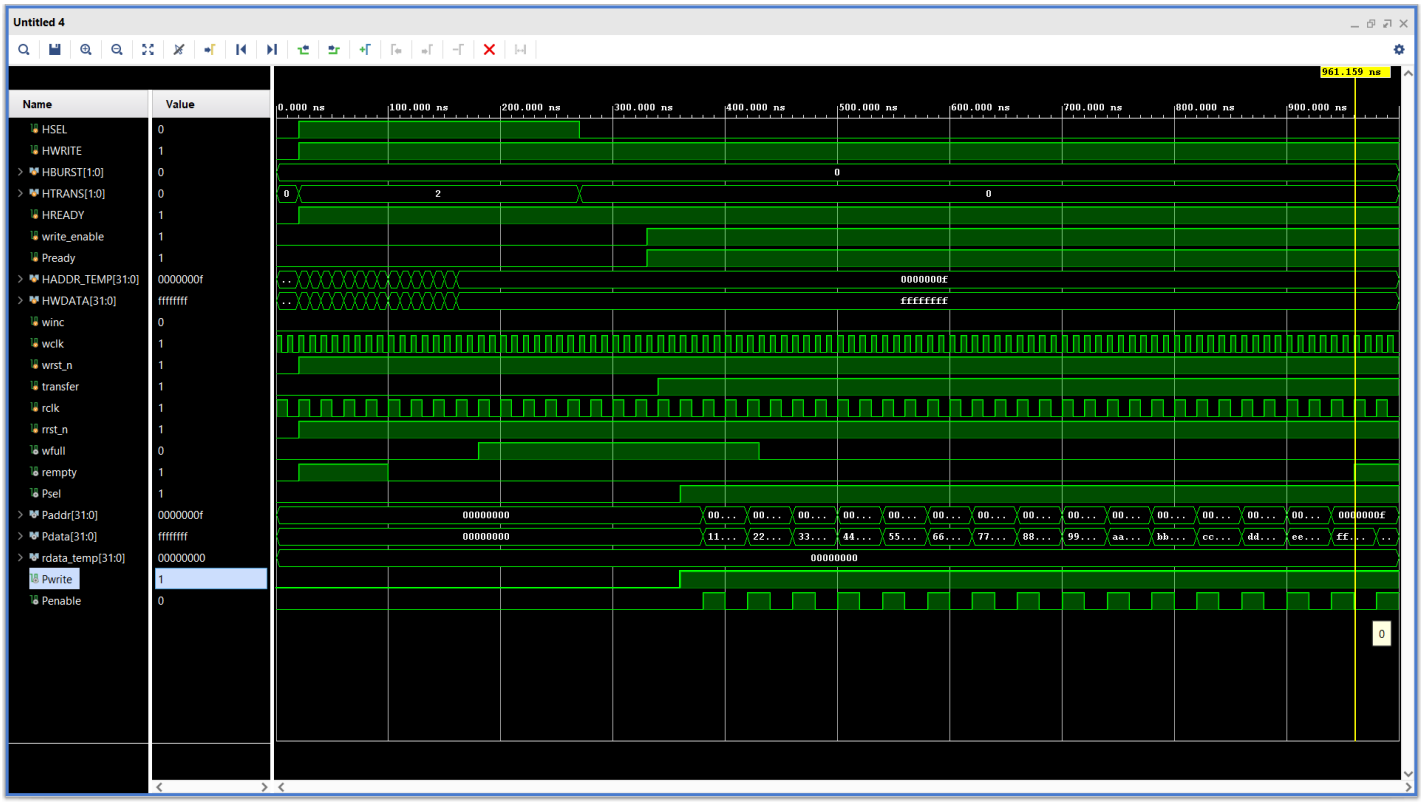
a. WRITE OPERATION MODE:



b. READ OPERATION



c. TOTAL SIMULATION



VERIFICATION USING UVM

1. Verification Technology – UVM

UVM(Universal Verification Methodology) is a Standard Verification Methodology that uses System Verilog constructs based on which a fully Functional Testbench can be built to verify the functional correctness of the Design Under Test(DUT). It is an IEEE standard/methodology based on System Verilog language.

Some of the Salient Features are

- A library of base classes for building testbench components (Agent, Sequencer, Driver, Monitor, Scoreboards, Environment class, etc.)
- Provides Verification phases for synchronizing concurrent processes.
- Provides a reporting mechanism for a consistent way of printing and logging results
- Provides a Factory, for constructing objects and substituting objects

2. Basic Test bench Functionality

The purpose of a test bench is to determine the correctness of the DUT. This is accomplished by the following steps.

- Generate stimulus
- Apply stimulus to the DUT
- Capture the response
- Check for correctness
- Measure progress against the overall verification goals

Some steps are accomplished automatically by the testbench, while others are manually determined by you. The methodology you choose determines how the preceding steps are carried out.

3. Methodology Basics

- Constrained-random stimulus
- Functional coverage
- Layered test bench using transactors

- Common test bench for all tests
- Test case-specific code kept separate from the testbench

4. CODE

Tb_top:

```
`include "uvm_macros.svh"
import uvm_pkg::*;


`include "top_wrapper.v"
`include "ahb_intf.sv"
`include "apb_intf.sv"

`include "ahb_seq_item.sv"
`include "ahb_sequence.sv"
`include "ahb_driver.sv"
`include "ahb_sequencer.sv"
`include "ahb_agent.sv"

`include "apb_driver.sv"
`include "apb_agent.sv"

`include "bridge_env.sv"
`include "test.sv"




typedef class bridge_test;
typedef class bridge_env;
typedef class ahb_agent;
typedef class apb_agent;
typedef class ahb_driver;
typedef class apb_driver;
typedef class ahb_sequencer;
typedef class ahb_seq_item;
typedef class ahb_sequence;

module testbench_top;
```

```
logic HRESETn;
logic HCLK;
logic Presetn;
logic Pclk;

initial begin
  HCLK = 0;
  forever #5 HCLK = ~HCLK;
end

initial begin
  Pclk = 0;
  forever #10 Pclk = ~Pclk;
end


initial begin
  HRESETn = 0;
  Presetn = 0;
  #20;
  HRESETn = 1;
  Presetn = 1;
end

initial begin
  #1000 $finish;
end

ahb_if ahb_if0();
apb_if apb_if0();


top_wrapper dut (

  .Pclk(apb_if0.Pclk),
  .Pready(apb_if0.Pready),
  .HADDR_TEMP(ahb_if0.HADDR_TEMP),
  .HBURST(ahb_if0.HBURST),
  .Hclk(ahb_if0.Hclk),
  .HREADY(ahb_if0.HREADY),
  .Hresetn(ahb_if0.Hresetn),
```

```
    .HSEL(ahb_if0.HSEL),
    .HTRANS(ahb_if0.HTRANS),
    .HWDATA(ahb_if0.HWDATA),
    .HWRITE(ahb_if0.HWRITE),
    .Presetn(apb_if0.Presetn),
    //.rinc(ahb_if0.rinc),
    //.write_enable(ahb_if0.write_enable),
    .Paddr(apb_if0.Paddr),
    .Pdata(apb_if0.Pdata),
    .Penable(apb_if0.Penable),
    .Psel(apb_if0.Psel),
    .Pwrite(apb_if0.Pwrite),
    .rdata_temp(apb_if0.rdata_temp)
    //.rempty(apb_if0.rempty),
    //.wfull(apb_if0.wfull)

  );

  assign ahb_if0.Hclk=HCLK;
  assign ahb_if0.Hresetn=HRESETn;
  assign apb_if0.Pclk=Pclk;
  assign apb_if0.Presetn=Presetn;
  initial begin

    uvm_config_db#(virtual ahb_if)::set(null, "uvm_test_top.env.ahb_agnt.driver", "ahb_vif",
ahb_if0);

    uvm_config_db#(virtual apb_if)::set(null, "uvm_test_top.env.apb_agnt.driver", "apb_vif",
apb_if0);

    run_test("bridge_test");
  end



endmodule
```

Test:

```
`ifndef TEST_
`define TEST_
```

```
class bridge_test extends uvm_test;

  `uvm_component_utils(bridge_test)

  bridge_env env;
  ahb_sequence ahb_seq;
  //apb_sequence apb_seq;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = bridge_env::type_id::create("env", this);
    ahb_seq = ahb_sequence::type_id::create("ahb_seq");
    //apb_seq = apb_sequence::type_id::create("apb_seq");
  endfunction

  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    ahb_seq.start(env.ahb_agnt.sequencer);
    //apb_seq.start(env.apb_agnt.sequencer);
    phase.drop_objection(this);
  endtask

endclass
`endif
```

bridge env:

```
 `ifndef ENV_
`define ENV_
class bridge_env extends uvm_env;

  `uvm_component_utils(bridge_env)

  ahb_agent ahb_agnt;
  apb_agent apb_agnt;

  function new(string name, uvm_component parent);
```

```
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ahb_agnt = ahb_agent::type_id::create("ahb_agnt", this);
    apb_agnt = apb_agent::type_id::create("apb_agnt", this);
  endfunction

endclass

`endif
```

AHB Agent:

```
`ifndef AGENT_
`define AGENT_
class ahb_agent extends uvm_agent;

  `uvm_component_utils(ahb_agent)

  ahb_driver    driver;
  ahb_sequencer sequencer;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    driver = ahb_driver::type_id::create("driver", this);
    sequencer = ahb_sequencer::type_id::create("sequencer", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    driver.seq_item_port.connect(sequencer.seq_item_export);
  endfunction

endclass
`endif
```

AHB Driver:

```
`ifndef DRIVER_
`define DRIVER_
class ahb_driver extends uvm_driver#(ahb_seq_item);

  `uvm_component_utils(ahb_driver)

  virtual ahb_if  ahb_if0;


  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction


  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (uvm_config_db#(virtual ahb_if)::get(this, "", "ahb_vif", ahb_if0))
      `uvm_info(get_full_name(),"got interface in ahb driver",UVM_NONE)
     else
      `uvm_fatal(get_full_name(),"interface not found in ahb driver")


  endfunction


  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      drive_transaction(req);
      seq_item_port.item_done();
    end
  endtask


  virtual task drive_transaction(ahb_seq_item item);
  `uvm_info(get_full_name(),"DATA from AHB Driver",UVM_NONE)
    item.print();
    wait (ahb_if0.Hresetn == 1);
      `uvm_info(get_full_name(),"**************************  AHB RESET
*****************  ",UVM_NONE)
    @(posedge ahb_if0.Hclk);
```

```
    ahb_if0.HADDR_TEMP   <= item.HADDR_TEMP;
    ahb_if0.HBURST       <= item.HBURST;
    ahb_if0.HREADY       <= 1'b1;
    ahb_if0.HSEL         <= 1'b1;
    ahb_if0.HTRANS       <= item.HTRANS;
    ahb_if0.HWRITE       <= 1'b1;

    @(posedge ahb_if0.Hclk);

     ahb_if0.HWDATA       <= item.HWDATA;

  endtask

endclass

`endif
```

AHB Intf:

```
`ifndef INTF_
`define INTF_

interface ahb_if();

                 //logic          Pclk;
                // logic          Pready;
                 logic [31:0]   HADDR_TEMP;
                 logic [1:0 ]   HBURST;
                 logic          Hclk;
                 logic          HREADY;
                 logic          Hresetn;
                 logic          HSEL;
                 logic [1:0 ]   HTRANS;
                 logic [31:0]   HWDATA;
                 logic          HWRITE;
                // logic          Presetn;
                //  logic          rinc;
                //  logic          write_enable;

endinterface

`endif
```

AHB Sequence Item:

```
`ifndef SEQ_
`define SEQ_

class ahb_seq_item extends uvm_sequence_item;



              rand   bit [31:0]   HADDR_TEMP;
              rand bit [1:0 ]   HBURST;
              rand bit [1:0 ]   HTRANS;
              rand   bit [31:0]   HWDATA;
              rand bit          HWRITE;
             // rand bit [2:0]   HSIZE;



  `uvm_object_utils_begin(ahb_seq_item)

    `uvm_field_int(HADDR_TEMP, UVM_ALL_ON)
    `uvm_field_int(HBURST, UVM_ALL_ON)
    `uvm_field_int(HTRANS,UVM_ALL_ON)
    `uvm_field_int(HWDATA,UVM_ALL_ON)
    `uvm_field_int(HWRITE,UVM_ALL_ON)
    //`uvm_field_int(HSIZE, UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name = "ahb_seq_item");
    super.new(name);
  endfunction

endclass

`endif
```

AHB Sequence:

```
`ifndef SEQUENCE_
`define SEQUENCE_

class ahb_sequence extends uvm_sequence#(ahb_seq_item);
```

```
  `uvm_object_utils(ahb_sequence)

  function new(string name = "ahb_sequence");
    super.new(name);
  endfunction

  virtual task body();
    ahb_seq_item item;

    repeat (100)
     begin
      item = ahb_seq_item::type_id::create("item");
      start_item(item);
      assert(item.randomize());
      `uvm_info("","sequence random in ahb",UVM_NONE)
      item.print();
      finish_item(item);
    end
  endtask

endclass


`endif
```

AHB Sequencer:

```
`ifndef SEQUENCER_
`define SEQUENCER_
class ahb_sequencer extends uvm_sequencer#(ahb_seq_item);

  `uvm_component_utils(ahb_sequencer)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

endclass

`endif
```

APB Agent:

```
`ifndef AGENT_H
 `define AGENT_H
class apb_agent extends uvm_agent;

  `uvm_component_utils(apb_agent)

  //apb_sequencer sequencer;
  apb_driver    driver;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
   // sequencer = apb_sequencer::type_id::create("sequencer", this);
    driver    = apb_driver::type_id::create("driver", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
   // driver.seq_item_port.connect(sequencer.seq_item_export);
  endfunction

endclass

 `endif
```

APB Driver:

```
`ifndef AGENT_H
 `define AGENT_H
class apb_agent extends uvm_agent;

  `uvm_component_utils(apb_agent)

  //apb_sequencer sequencer;
  apb_driver    driver;

  function new(string name, uvm_component parent);
    super.new(name, parent);
```

```
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
   // sequencer = apb_sequencer::type_id::create("sequencer", this);
    driver    = apb_driver::type_id::create("driver", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
   // driver.seq_item_port.connect(sequencer.seq_item_export);
  endfunction

endclass

 `endif
```

APB Interface:

```
`ifndef INTF_H
  `define INTF_H
interface apb_if();
   logic            Pclk;
   logic            Presetn;
   logic             Pready;
  logic [31:0]       Paddr;
  logic [31:0]       Pdata;
  logic            Penable;
  logic            Psel;
  logic            Pwrite;
  logic [31:0]       rdata_temp;
  logic            rempty;
  logic            wfull;

endinterface
`endif
```

APB Sequence Item:

```
`ifndef SEQUENCE_ITEM
`define SEQUENCE_ITEM
class apb_seq_item extends uvm_sequence_item;
```

```
        bit [31:0]      Paddr;
        bit [31:0]      Pdata;
        bit         Penable;
        bit         Psel;
        bit         Pwrite;
        bit [31:0]      rdata_temp;
        bit         rempty;
        bit         wfull;



  `uvm_object_utils_begin(apb_seq_item)
    `uvm_field_int(Paddr, UVM_ALL_ON)
    `uvm_field_int(Pdata, UVM_ALL_ON)
    `uvm_field_int(Penable, UVM_ALL_ON)
    `uvm_field_int(Psel, UVM_ALL_ON)
    `uvm_field_int(Pwrite,UVM_ALL_ON)
    `uvm_field_int(rdata_temp, UVM_ALL_ON)
    `uvm_field_int(rempty, UVM_ALL_ON)
    `uvm_field_int(wfull, UVM_ALL_ON)


  `uvm_object_utils_end


  function new(string name = "apb_seq_item");
    super.new(name);
  endfunction

endclass

`endif
```

CONCLUSION

This project successfully designed and implemented an **APB-to-AHB bridge**, achieving the goal of enabling efficient communication between **low-power peripherals** and **high-performance processors** within a **System-on-Chip (SoC)** environment. The bridge effectively integrates different bus protocols, ensuring seamless data transfer while adhering to **AMBA specifications**.

The project underscores the key benefits of **APB-to-AHB bridges**, including **optimized communication**, **power efficiency**, **scalability**, and **standardized interfaces**. By leveraging these strengths, the bridge enhances the development of **high-performance and power-conscious SoCs**.

Moving forward, this work sets the stage for future improvements in **APB-to-AHB bridge design**. Areas such as **performance optimization, power reduction techniques, security enhancements,** and **support for evolving protocols** present promising directions for research and innovation. Continuous refinement of the bridge's functionality will ensure **more efficient, reliable, and secure communication** within next-generation SoCs, contributing to the advancement of **complex and power-efficient system-on-chip architectures**.

FUTURE SCOPE

While this project successfully implemented a functional APB2AHB bridge, there are several promising avenues for further exploration:

1. **Performance optimization**: The bridge's performance can be further enhanced by investigating techniques like pipeline stages, burst transfer optimization, and advanced buffering strategies. These improvements could potentially reduce latency and increase overall throughput.
2. **Low-power design:** Exploring alternative design methodologies and low-power techniques like clock gating and power management within the bridge can contribute to minimizing power consumption without compromising functionality. This is crucial for battery-powered devices and power-constrained SoC applications.
3. **Security considerations:** Integrating security features into the bridge design is becoming increasingly important. Future work could involve exploring encryption and authentication mechanisms to safeguard data transfers between different bus domains, mitigating potential security vulnerabilities within the SoC.
4. **Adaptability and scalability:** The bridge's design can be further explored to adapt to emerging AMBA protocols or integrate with other on-chip communication standards. architectureprove the bridge's versatility and enable its application in a wider range of SoC architectures.

By delving deeper into these areas, future research can refine the APB2AHB bridge design, pushing the boundaries of its capabilities and ensuring its continued relevance in the evolving landscape of SoC communication solutions.

REFRENCES

1. **ARM AMBA Specification v3.0:** https://www.arm.com/architecture/system-architectures/amba/amba-specifications
2. **“Advanced High-performance Bus (AHB) Protocol - A Tutorial" by Xilinx:** https://support.xilinx.com/s/question/0D52E000078SIT1SAO/create-ahb-ip?language=en_US
3. **"Advanced Peripheral Bus (APB) Protocol - A Tutorial" by Xilinx:** https://docs.xilinx.com/r/en-US/am011-versal-acap-trm/APB-Programming-Interface
4. **"A Survey of On-Chip Communication Architectures in System-on-Chip Designs" by Na-Young Kim, et al.:** https://ieeexplore.ieee.org/document/982487
5. **"Low-Power Design Techniques for Application-Specific Integrated Circuits" by Chandrakasan, Anantha P., and Jieh-Wen Tseng:** https://link.springer.com/book/10.1007/978-1-4419-9973-3
6. **"Bridge Design for On-Chip Communication Architectures" by Peter Piguet, et al.:** https://eps.ieee.org/images/files/HIR_2021/ch13_co-d.pdf
7. **"A High-Performance and Scalable Network-on-Chip Architecture for Next-Generation SoCs" by Guohua He, et al.:** https://ieeexplore.ieee.org/document/9824621

8. **"A Low-Power Design Methodology for Application-Specific Integrated Circuits" by Anantha Chandrakasan, et al.:**
   http://ieeexplore.ieee.org/document/8073688/