# INTERNSHIP

# REPORT

# ON

# DESIGN OF

# AHB-TO-APB BRIDGE


**BY**

**SRINIDHI M R**

**III YEAR,B.E ECE**

**SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY,CHENNAI**

# BONAFIDE CERTIFICATE

This is to certify that the **"Internship Report"** submitted by **SRINIDHI M R** is the work done by her and submitted during the internship period at **Maven Silicon.** The report is being submitted in partial fulfilment of the internship requirements for Maven Silicon

The work and findings presented in this report are original and reflect the knowledge and skills acquired by me during the course of the internship.

# ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to the Board of Management of **SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY** for their kind encouragement in allowing me to undertake this training and for supporting its successful completion. I am truly grateful to them.

I convey my thanks to **Dr. N.M. NANDHITHA., M.E., Ph.D., Prof. & Dean, School of Electrical and Electronics Engineering**, and **Dr. T. RAVI., M.E., Ph.D., Head of the Department of Electronics and Communication Engineering**, for providing the necessary support that enabled me to pursue this Design Internship with an esteemed organization.

I would like to express my sincere and deep sense of gratitude to **Ms. Sathyapriya**, my mentor at **Maven Silicon**, for her valuable guidance, suggestions, and constant encouragement throughout the internship. Her support and mentorship played a crucial role in my learning and the successful completion of the internship. I am extremely thankful to her for sharing her knowledge and expertise with me.

I also wish to extend my thanks to all the teaching and non-teaching staff members of **Maven Silicon**, whose help and assistance were invaluable during my internship.

## TABLE OF CONTENTS


| CHAPTER | TITLE | PAGE No |
|---|---|---|
| | **ABSTRACT** | **1** |
| | **OBJECTIVE** | **2** |
| 1 | INTRODUCTION | 3 |
| | 1.1 DATA TYPES IN VERILOG | 3 |
| | 1.2 MODELLING TECHNIQUES | 3 |
| | 1.3 OPERATORS IN VERILOG | 4 |
| | 1.4 ASSIGNMENT | 5 |
| 2 | SEQUENTIAL FLIP FLOPS AND COUNTERS | 6 |
| | 2.1 FLIP FLOPS | 6 |
| | 2.2 COUNTERS | 7 |
| 3 | FINITE STATE MACHINES | 10 |
| | 3.1 TYPES OF FSMs | 10 |
| | 3.2 STATE REPRESENTATION | 11 |
| | 3.3.STATE TRANSITION LOGIC | 11 |
| | 3.4 APPLICATIONS OF FSMs | 11 |
| | 3.5 ADVANTAGES OF FSMs | 12 |
| 4 | DESIGN OF AHB TO APB BRIDGE | 13 |
| | 4.1 AMBA OVERVIEW | 13 |
| | 4.2 AHB TO APB BRIDGE FUNCTIONALITY | 13 |
| | 4.3 PROJECT DESCRIPTION | 14 |
| | 4.4 KEY OPERATIONS OF THE AHB TO APB BRIDGE | 16 |
| | 4.5 KEY CONCEPTS IN THE AHB TO APB BRIDGE | 17 |
| | 4.6 TYPES OF SIGNALS | 17 |
| | 4.7 TRANSFER TYPES | 18 |
| | 4.8 STATE TRANSITIONS DONE IN THE PROJECT | 19 |
| 5 | SYNTHESIS (RTL VIEWER) | 23 |
| 6 | SIMULATED OUTPUT | 24 |
| 7 | CONCLUSION | 29 |

# LIST OF FIGURES


| FIGURE No | TITLE | PAGE No. |
|---|---|---|
| 1 | BLOCK DIAGRAM OF AHB-TO-APB BRIDGE | 22 |
| 2 | SYNTHESIS PART OF AHB-TO-APB BRIDGE | 23 |

# ABSTRACT

The AHB (Advanced High-performance Bus) to APB (Advanced Peripheral Bus) bridge is a key component in AMBA-based SoC (System on Chip) designs. It acts as an interface between the high-performance AHB and low-power APB, enabling seamless communication and data transfer between these two bus protocols. The AHB is optimized for speed and bandwidth, catering to high-performance operations, while the APB is designed for simple and power efficient.

This project focuses on designing and verifying an AHB to APB bridge, ensuring proper protocol conversion and data integrity. The design is implemented using Verilog HDL and simulated using ModelSim. Synthesis is performed using Quartus Prime to analyze the design's performance metrics. The bridge enables the AHB master to communicate with APB slaves by converting high-speed pipelined AHB signals into non-pipelined APB signals, meeting industry standards for interoperability in complex SoC architectures.

# OBJECTIVE

The objective of this project is to:

1. Develop a Functional Bridge: Design a Verilog-based AHB to APB bridge to facilitate protocol conversion and efficient communication.
2. Maintain Data Integrity: Ensure error-free data transmission between the AHB master and APB slave devices.
3. Optimize Performance: Design the bridge with minimal latency and optimized resource utilization to enhance SoC performance.
4. Protocol Compliance: Validate adherence to AMBA specifications for both AHB and APB protocols.
5. Verification and Testing: Simulate the design in ModelSim for functional verification and perform synthesis in Quartus Prime to check for timing, area, and power metrics.
6. Enhance Understanding: Gain practical experience in HDL coding, protocol design, and SoC development processes.

# CHAPTER 1 : INTRODUCTION

Verilog is a hardware description language (HDL) used to model, design, and simulate digital systems. Developed in the 1980s, it has become one of the most widely used HDLs for designing and verifying ASICs, FPGAs, and SoC architectures. Verilog allows designers to describe the behaviour and structure of electronic circuits at various abstraction levels, from high-level functional specifications to gate-level implementations.

## 1.1 Data Types in Verilog

Verilog offers a variety of data types tailored for hardware modelling:

1. Net Types:
    - Wire: Represents a physical connection. It carries values driven by gates or continuous assignments.
2. Variable Types:
    - Reg: Stores a value across simulation cycles and is used in procedural blocks (always and initial).
    - Integer: Used for general-purpose integer calculations.
    - Time: Stores time values for simulation purposes.
    - Real: Stores real numbers (floating-point values).
3. Vectors:
    - Verilog supports bit vectors like reg [7:0] and wire [31:0] for multi-bit signals.
4. Constants:
    - parameter and localparam are used to define constants in designs.

## 1.2 Modelling Techniques

Verilog supports different modelling techniques for describing hardware:

1. Behavioral Modeling:
    - Describes the circuit's functionality using high-level constructs like always and initial blocks.
    - Example:

```
always @(posedge clk)
  out = a + b;
```

2. Dataflow Modeling:
    - Describes data flow using continuous assignments (assign statements).
    - Example:

```
assign y = a & b;
```

3. Structural Modeling:
    - Represents the design as a connection of basic components (e.g., gates, modules).
    - Example:

```
and g1(y, a, b);
```

4. Switch-Level Modeling:
    - Models digital circuits at the transistor level using primitives like pmos, nmos, etc.

## 1.3 Operators in Verilog

Verilog supports a rich set of operators categorized as:

i. Arithmetic Operators: +, -, *, /, %

ii. Relational Operators: ==, !=, <, <=, >, >=

iii. Logical Operators: &&, ||, !

iv. Bitwise Operators: &, |, ^, ~

v. Reduction Operators: &, |, ^ applied to all bits of a vector.

vi. Shift Operators: <<, >>

vii. Concatenation and Replication:

- Concatenation: {a, b}
- Replication: {n{a}}

## 1.4 Assignment

1.Continuous Assignment:

- Used for dataflow modelling to assign values to wire
- Example:

```
assign sum = a+b;
```

2.Procedural Assignment:

Used in procedural blocks ( always, initial) to assign values to reg variables.

Example:

```
always @(posedge clk)
   q <= d;  // Non-blocking assignment
```

3.Blocking and Non-Blocking Assignments

Blocking : Executes sequentially within a block

```
a=b+c;
```

# CHAPTER 2: SEQUENTIAL FLIP-FLOPS AND COUNTERS

Sequential circuits are an essential part of digital electronics where the output depends not only on the current input but also on the history of past inputs. Flip-flops and counters are fundamental building blocks in sequential logic circuits. During our learning, we were also taught how to code these components, enabling us to implement them in real-world applications and simulations.

## 2.1 Flip-Flops

Flip-flops are bistable devices, meaning they have two stable states, representing binary 0 and 1. They store a single bit of information and are used as memory elements in digital systems. Flip-flops are triggered by clock signals, which synchronize changes in their state. The common types of flip-flops include:

1. **SR Flip-Flop (Set-Reset Flip-Flop)**:
    - It has two inputs: Set (S) and Reset (R).
    - When S = 1 and R = 0, the output is set to 1.
    - When S = 0 and R = 1, the output is reset to 0.
    - When S = 0 and R = 0, the output remains unchanged.
    - When S = 1 and R = 1, the state is undefined (in basic SR latch).
    - *We learned how to code this flip-flop using Verilog, including its truth table and state transitions.*
2. **D Flip-Flop (Data or Delay Flip-Flop)**:
    - It has a single input, D (Data), and a clock signal.
    - The output (Q) follows the input (D) on the triggering edge of the clock.

- This is widely used in shift registers and data storage applications.
- *We implemented this in code to understand its real-time behaviour.*

3. **JK Flip-Flop**:
    - It eliminates the undefined state of the SR flip-flop.
    - When J = 1 and K = 0, it sets the output to 1.
    - When J = 0 and K = 1, it resets the output to 0.
    - When J = 1 and K = 1, the output toggles.
    - *Coding this flip-flop taught us the importance of managing toggle conditions.*
4. **T Flip-Flop (Toggle Flip-Flop)**:
    - It toggles the output state when the input (T) is 1 and a clock edge occurs.
    - When T = 0, the output remains unchanged.
    - Used in counters and toggle operations.
    - *We learned how to write code to simulate its toggle functionality.*

## 2.2 Counters

Counters are sequential circuits that count pulses and represent the count as a binary number. They are built using flip-flops and are widely used in timers, frequency dividers, and event counting.

1. **Types of Counters**:
    - **Asynchronous (Ripple) Counter**:
        - In an asynchronous counter, the output of one flip-flop drives the clock input of the next flip-flop.

- They are simpler but slower due to propagation delays.
- *We coded asynchronous counters to understand how delays accumulate in ripple configurations.*

- **Synchronous Counter**:
  - All flip-flops are triggered by the same clock signal.
  - They are faster and more reliable than asynchronous counters.
  - *Coding synchronous counters helped us grasp clock synchronization in circuits.*

2. **Common Counter Types**:
   - **Binary Counter**:
     - Counts in binary sequence (e.g., 0000, 0001, 0010, ...).
     - *We coded binary counters to observe binary progression in real-time.*
   - **Decade Counter**:
     - Counts from 0 to 9 (a mod-10 counter) and then resets.
   - **Up/Down Counter**:
     - Counts up or down based on control input.
   - **Ring Counter**:
     - A circular shift register where only one flip-flop is set at a time.
   - **Johnson Counter**:
     - A twisted ring counter where the inverted output of the last flip-flop is fed back as input to the first flip-flop.

3. **Applications of Counters**:
    - Digital clocks
    - Frequency measurement
    - Memory address sequencing
    - Event counting in embedded systems

*Learned how to code these counters enhanced our understanding of their applications and functioning, especially in simulation environments like ModelSim and Quartus Prime.*

# CHAPTER 3: FINITE STATE MACHINES (FSMs)

A **Finite State Machine (FSM)** is a computational model used to design systems that transition through a finite number of states based on inputs and predefined rules. FSMs are a cornerstone of sequential logic design and are widely employed in hardware design, software algorithms, and automation systems.

1. **Definition**:
    - An FSM is a system with a set of distinct states.
    - The machine transitions from one state to another based on external inputs and transition logic.
2. **Components of an FSM**:
    - **States**: A finite number of distinct configurations the system can be in.
    - **Inputs**: External signals or events that trigger state changes.
    - **Transitions**: Defined paths between states based on inputs or conditions.
    - **Outputs**: System responses or actions, which may depend on the state or both the state and inputs.
    - **Initial State**: The state in which the FSM begins operation.

## 3.1 TYPES OF FSMs

1. **Moore Machine**:
    - The outputs depend only on the current state.
    - **Advantages**:
        - Simpler design since outputs are decoupled from inputs.
        - Easier to debug.
        - **Example**: A vending machine dispensing a product after receiving the correct payment.

2. **Mealy Machine**:
    - The outputs depend on both the current state and the inputs.
    - **Advantages**:
        - Can respond faster to inputs since outputs are input-dependent.
        - **Example**: A digital lock system that triggers an alert based on the entered key sequence.

## 3.2 STATE REPRESENTATION

- States are often represented using binary encoding. For example:
    - **S0**: 00
    - **S1**: 01
    - **S2**: 10
    - **S3**: 11

## 3.3 STATE TRANSITION LOGIC

- Transitions are governed by rules or conditions.
- These conditions are typically based on input signals.
- Example: In a simple FSM with states S0, S1, and S2:
    - If in S0 and input = 1 → Transition to S1.
    - If in S1 and input = 0 → Transition back to S0.
    - If in S1 and input = 1 → Transition to S2.

## 3.4 APPLICATIONS OF FSMs

1. **Digital Electronics**:
    - Control units in CPUs.
    - Protocol design for communication systems.
    - Sequence detectors.

2. **Embedded Systems**:
    - Traffic light controllers.
    - Elevator control systems.
    - Washing machine cycles.
3. **Software Development**:
    - Lexical analyzers in compilers.
    - Decision-making algorithms.
    - Video game character behavior modeling.

## 3.5 ADVANTAGES OF FSMs

1. **Predictable Behavior**:
    - Each state and transition is well-defined, ensuring deterministic operation.
2. **Modularity**:
    - FSMs can be broken down into smaller, manageable subsystems.
3. **Scalability**:
    - Easily scalable for complex systems with many states.
4. **Debugging and Testing**:
    - State diagrams make it easy to identify and fix issues.

# CHAPTER 4: DESIGN OF AHB-APB BRIDGE

The AHB to APB Bridge is a critical component of the AMBA (Advanced Microcontroller Bus Architecture) developed by ARM. It serves as a bridge to connect the high-performance AHB (Advanced High-Performance Bus) with the low-power APB (Advanced Peripheral Bus). The bridge allows for seamless communication between these two buses, enabling efficient data transfers in System-on-Chip (SoC) designs.

## 4.1 AMBA Overview

- **Developer**: ARM Ltd.
- **Purpose:** To define a set of standard buses for SoC designs, enabling component interoperability and design reuse.
- **AMBA Specifications:**
  - AHB (Advanced High-Performance Bus): High-speed, pipelined, multi-master bus for memory and high-performance peripherals.
  - APB (Advanced Peripheral Bus): Simple, low-power, non-pipelined bus for low-bandwidth peripherals.

## 4.2 AHB to APB Bridge Functionality

- **Purpose**: Acts as a protocol converter between AHB and APB.
- **Key Features**:
  1. Data Transfer: Converts high-speed AHB transactions to the simpler APB protocol.
  2. Address Translation: Decodes AHB addresses to APB addresses.
  3. Clock Domain Crossing: Handles differences in clock domains between AHB and APB.
  4. Pipeline and Buffering: Ensures data integrity during high-speed transfers.

## 4.3 Project Description

This project involves designing the AHB to APB Bridge to facilitate communication between:

1. AHB Master Devices: high-speed peripherals.
2. APB Slave Devices: GPIO, UART, and low-speed peripherals.

- Purpose: The bridge acts as an interface between the high-speed AHB (used for high-performance system-level interconnects) and the low-power APB (used for connecting slower peripherals like UART, GPIO, etc.).
- Functionality:
    - Transfers address, control, and data between AHB and APB.
    - Decodes addresses and generates peripheral select signals for APB devices.
    - Ensures compatibility between AHB burst operations and APB's simpler protocol (which does not support burst transfers).

1. System Bus → APB Transfer:
    - The bridge decodes the address received from the AHB bus and generates an appropriate peripheral select signal for the target APB peripheral.
    - It holds the address and keeps it valid throughout the transfer.
2. Data Transfer:
    - Only one peripheral is active at a time on the APB. The bridge ensures that the correct peripheral receives the transfer based on the decoded address.

3. Read and Write Transfers:
    - For read transfers, data from the APB peripheral is driven onto the AHB system bus.
    - For write transfers, data is written from the AHB system bus to the selected APB peripheral.
4. Wait States:
    - When AHB burst transfers are converted into individual APB transactions, the bridge may introduce wait states. These ensure that AHB masters wait while the slower APB completes its current operation.
5. High-Speed AHB, Low-Power APB:
    - AHB operates at high speed and supports pipelining and burst transfers.
    - APB operates at lower power and supports simpler, non-pipelined single transactions. The bridge ensures synchronization between these two buses.
6. Control Signals:
    - The bridge generates the PENABLE signal, which activates the transfer on the APB.
    - The peripheral select signal is valid only during the transfer to ensure proper data flow.

- AHB Compatibility:
    - The bridge ensures that AHB burst transfers are handled properly, even though the APB does not support bursts. Each burst transfer is broken down into multiple single transfers on the APB side.

- Peripheral Access:
    - Only one peripheral is active on the APB at any given time. The bridge generates select signals to activate the required peripheral.

### 4.4 Key Operations of the AHB to APB Bridge

1. Latch Address:
    - The bridge latches the address from the AHB bus and holds it until the transfer is completed on the APB.
2. Generate Peripheral Select:
    - Based on the latched address, the bridge activates the specific APB peripheral using a select signal.
3. Wait for Transfer Completion:
    - Wait states may be added to allow slower APB peripherals to complete their operations while ensuring that AHB masters are synchronized.

**AHB Slave Interface**: Receives transactions from the AHB master. Receives requests such as single or burst operations from the AHB master.Decodes the request and ensures compatibility with the simpler APB protocol.

**APB Controller Interface** :Generates the required APB control signals (PSEL, PENABLE, PWRITE, etc.).Handles the setup and access phases for each transaction.

**State Machine:** Manages the transition between AHB operations and APB phases. Ensures that AHB burst requests are handled as individual APB transactions.

## 4.5 Key Concepts in the AHB to APB Bridge

### Bus Cycle

- A bus cycle refers to a single operation (e.g., read or write) performed over the bus.
- **In AHB:**
    - Operations are pipelined for higher throughput.
    - Multiple transfers can overlap, reducing wait states.
- **In APB:**
    - Operations are single-cycle and non-pipelined.
    - Each transfer requires a setup phase and an access phase.

### Burst Operation

- **Burst Operation**: A series of consecutive data transfers initiated by a single address phase, commonly used in AHB for high-speed memory transactions.
- **In AHB:**
    - Supported: Can perform burst operations like single, incremental, and wrapping bursts.
    - Efficient: Reduces address overhead by transferring multiple data beats after one address phase.
- **In APB:**
    - Not Supported: APB is designed for simple, low-power peripherals that do not require high-speed burst transfers.
    - Transactions are performed one at a time with no burst capability.

## 4.6 TYPES OF SIGNAL

In this project, the **Hburst** signal determines the type of burst transfer being performed. Each burst transfer involves multiple data beats sent consecutively. The burst type is encoded using a 3-bit field.

1. **Single Transfer (000)**:
    - A single data beat is transferred.
2. **Incrementing Burst (001, 011, 101, 111)**:
    - The address increases sequentially for each data beat.
    - The length of the burst depends on the specific burst type:
        - INCR4: 4 beats.
        - INCR8: 8 beats.
        - INCR16: 16 beats.
    - Useful for accessing contiguous memory locations.
3. **Wrapping Burst (010, 100, 110)**:
    - Similar to incrementing bursts but the address wraps back to a lower boundary when it crosses a burst boundary (e.g., for WRAP4, the boundary is 4 words).
    - Ensures that data is fetched efficiently when the memory subsystem operates on fixed-size blocks.

## 4.7 TRANSFER TYPES

In the AMBA AHB-APB protocol, the Htrans (Transfer Type) signal is a 2-bit field used to specify the type of transaction being performed on the bus. This signal indicates whether the transfer is active, idle, or part of a burst operation. It is crucial for the operation of the AHB and helps manage pipelining and the sequence of data transfers.

1. **Idle (00)**:

- Indicates that no data transfer is required on the bus.
- The address and control signals are not valid during an idle state.
- Used when the bus is not actively transferring data.

2. **Busy (01)**:
   - The master is busy with internal operations but is not performing a data transfer.
   - Useful for managing pipelining or when the master requires additional cycles before initiating the next transfer.
   - Keeps the master in control of the bus while deferring the actual data transfer.
3. **Non-Sequential (10)**:
   - Indicates the start of a new transaction, unrelated to the previous one.
   - The address on the bus is valid and does not follow any specific pattern from the previous address.
   - Used for single transfers or the start of a burst.
4. **Sequential (11)**:
   - Indicates the next transfer in a burst sequence.
   - The address is an increment of the previous address, following the burst pattern (e.g., incremental or wrapping).
   - Used for efficient block transfers like memory accesses.

## 4.8 STATE TRANSITIONS DONE IN THE PROJECT

1. **ST_IDLE (3'b000)**:
   - This is the idle state where the controller is waiting for a valid signal.
   - If a valid signal is received and the operation is a write (hwrite_reg is 1), the controller transitions to the ST_WWAIT state to prepare for a write operation.
   - If the operation is a read (hwrite_reg is 0), it transitions to the ST_READ state.

2. **ST_READ (3'b001)**:
   - In this state, the controller is ready to initiate a read operation.
   - If the operation is still a read (!hwrite_reg), the controller moves to ST_RENABLE for read enable processing.
3. **ST_RENABLE (3'b010)**:
   - This state is for enabling the read operation.
   - Once enabled, the controller returns to the ST_IDLE state, completing the read cycle.
   - The prdata output is set to a placeholder value (32'hDEADBEEF in this case) as an example of the read data.
4. **ST_WWAIT (3'b011)**:
   - This state is a wait state for write operations.
   - If valid and hwrite_reg are true, the controller transitions to the ST_WRITE state.
   - The controller also sets the address (haddr1) for the write operation.
5. **ST_WRITE (3'b100)**:
   - In this state, the controller prepares for the write operation.
   - It transitions to ST_WENABLE to enable the write process.
   - The write data (hwdata1) is captured for the upcoming write.
6. **ST_WENABLE (3'b101)**:
   - The ST_WENABLE state is responsible for enabling the write operation.
   - Once enabled, the controller moves back to the ST_IDLE state, completing the write operation.

- The controller ensures that Hreadyout is low while the write data is being processed, and then goes high once the write is completed.

7. **ST_WRITEP (3'b110)**:
   - Similar to ST_WWAIT, this state handles the second phase of the write operation but with different data (hwdata2).
   - It waits for the valid signal and the write flag to indicate a second write, preparing the system to move to the next state.
8. **ST_WENABLEP (3'b111)**:
   - In this state, the controller enables the second phase of the write operation.
   - Once enabled, the controller returns to the ST_IDLE state, completing the second write operation.

# BLOCK DIAGRAM OF AHB-APB BRIDGE

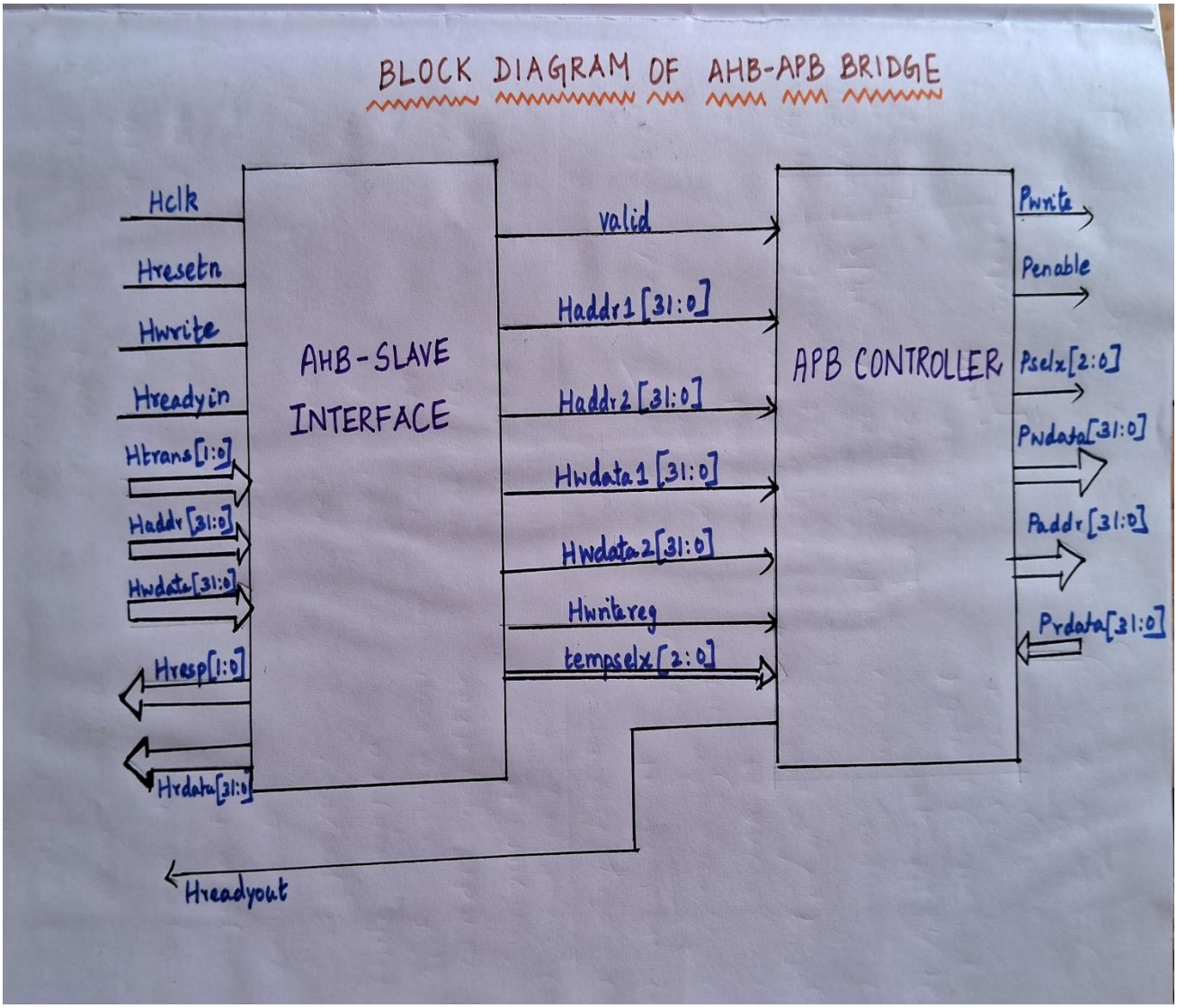



***Figure 1: Ahb-to-Apb-bridge design***

# CHAPTER 5: SYNTHESIS (RTL VIEWER)

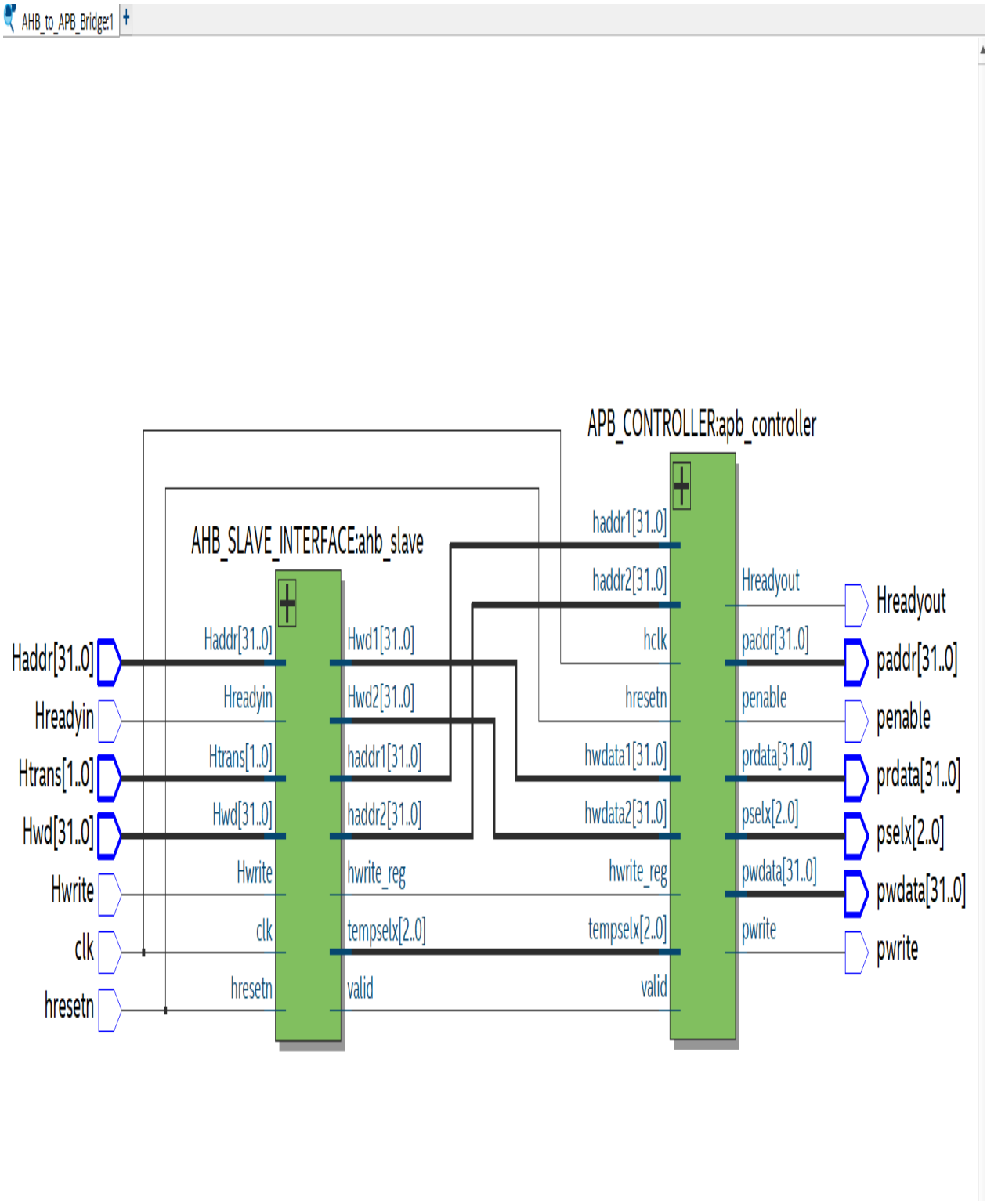



***Figure 2: Synthesis part of AHB-TO-APB BRIDGE***

# CHAPTER 6 : SIMULATED OUTPUT

- The AHB-to-APB Bridge converts high-speed AHB signals to slower APB protocol signals, ensuring proper handshaking.
- Control Signals (pselx, penable) in the APB domain synchronize and control read/write operations.

1. **Single Write Operation**

- **Waveform Observation:**
  - The AHB signals (Haddr, Hwdata, Hwrite, Htrans) indicate a valid single write transaction.
  - Haddr shows the address of the write operation.
  - Hwdata shows the data to be written at the specified address.
  - Hwrite is high (1), indicating a write operation.
  - Htrans transitions to a non-idle state (e.g., St1), indicating the start of a valid transaction.
- **APB Interface:**
  - The paddr signal receives the address from the AHB interface.
  - pwdata contains the data to be written.
  - pwrite is high (1), signifying a write.
  - penable and pselx signals become active to ensure the write occurs in the APB domain.

2. **Single Read Operation**

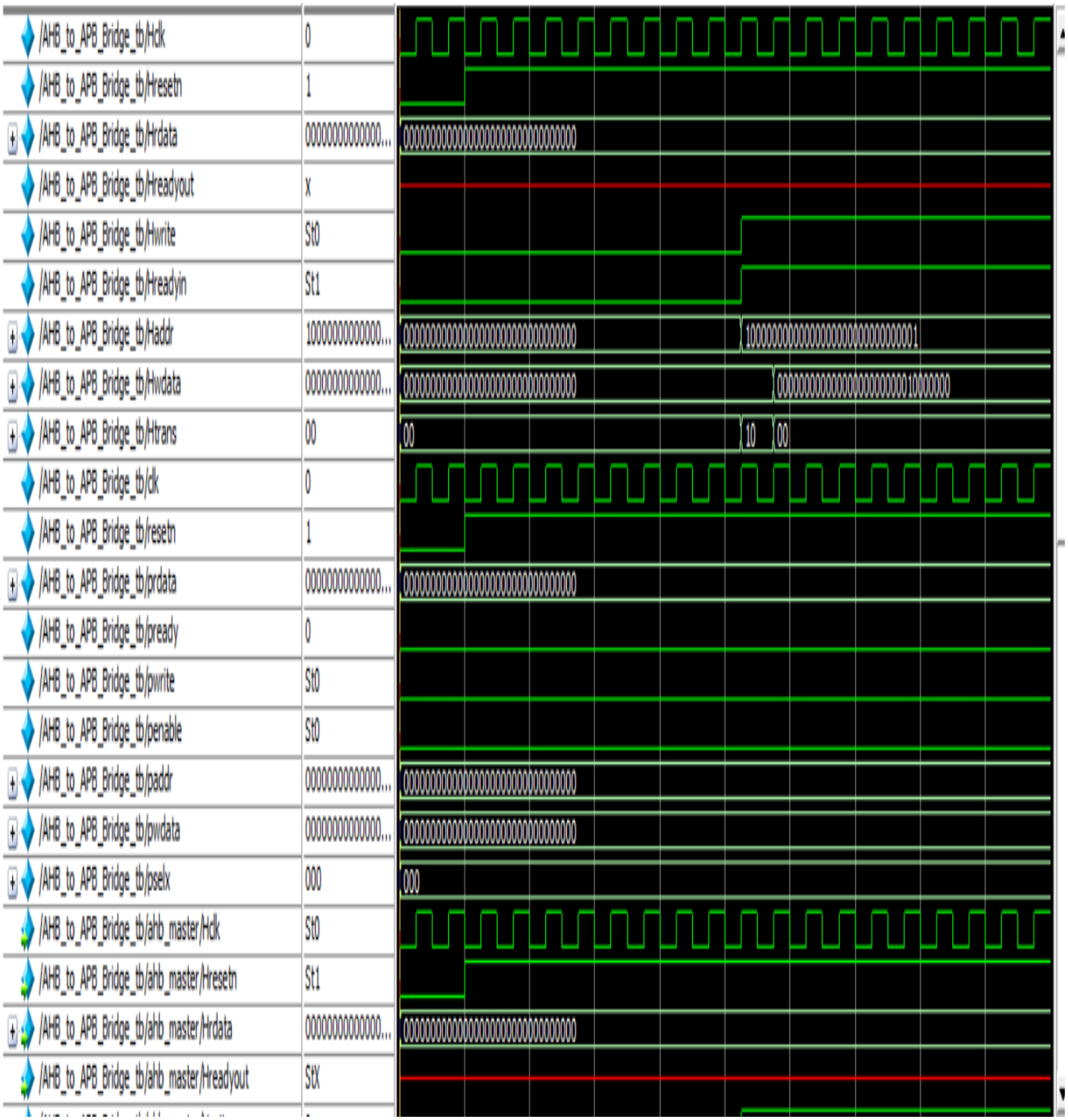
- **Waveform Observation:**
  - Similar to the write operation, Haddr and Htrans indicate a valid read transaction.
  - Hwrite is low (0), denoting a read operation.

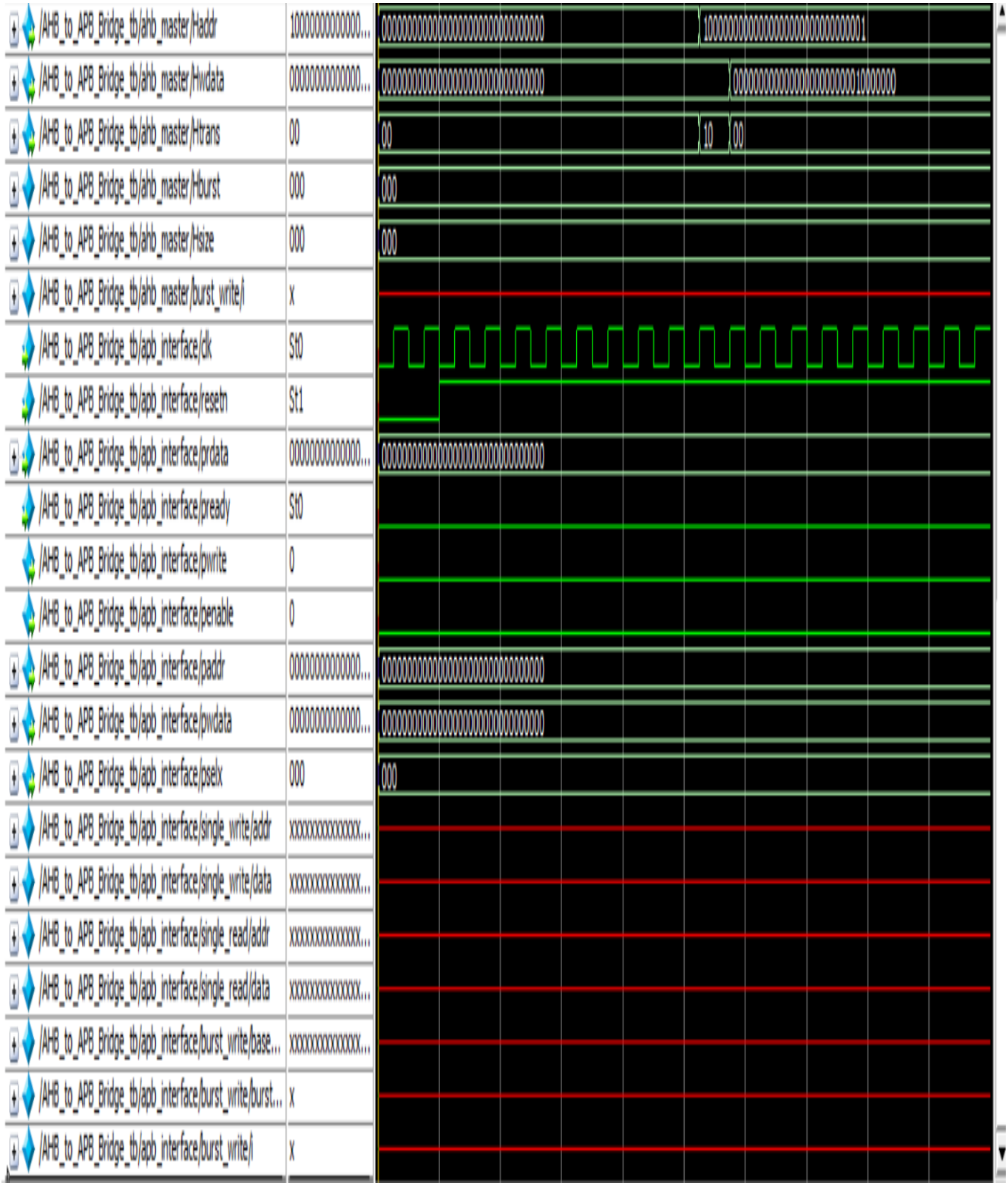
    - The read data (prdata) appears on the APB interface after valid handshaking.

- **APB Interface:**
    - The paddr signal carries the read address.
    - penable and pselx signals are asserted to enable communication.
    - The prdata signal outputs the data read from the specified address.

3. **Burst Write Operation**

- **Waveform Observation:**
    - Multiple addresses and data values are transferred over consecutive clock cycles.
    - Hburst indicates a burst type (e.g., incrementing or wrapping).
    - Haddr changes incrementally (or according to the burst type), and corresponding data (Hwdata) is sent.
- **APB Interface:**
    - The paddr signal increments according to the burst pattern.
    - pwdata changes with each write cycle to match the burst data.
    - penable remains asserted for the duration of the burst transfer.
- **State Transitions:**
    - AHB state transitions (Htrans) show transaction validity.
    - APB signals (penable, pselx) ensure the transaction is completed successfully.


/AHB_to_APB_Bridge_tb/Hclk
/AHB_to_APB_Bridge_tb/Hresetn
/AHB_to_APB_Bridge_tb/Hrdata
/AHB_to_APB_Bridge_tb/Hreadyout
/AHB_to_APB_Bridge_tb/Hwrite
/AHB_to_APB_Bridge_tb/Hreadyin
/AHB_to_APB_Bridge_tb/Haddr
/AHB_to_APB_Bridge_tb/Hwdata
/AHB_to_APB_Bridge_tb/Htrans
/AHB_to_APB_Bridge_tb/clk
/AHB_to_APB_Bridge_tb/resetn
/AHB_to_APB_Bridge_tb/prdata
/AHB_to_APB_Bridge_tb/pready
/AHB_to_APB_Bridge_tb/pwrite
/AHB_to_APB_Bridge_tb/penable
/AHB_to_APB_Bridge_tb/paddr
/AHB_to_APB_Bridge_tb/pwdata
/AHB_to_APB_Bridge_tb/pselx
/AHB_to_APB_Bridge_tb/ahb_master/Hclk
/AHB_to_APB_Bridge_tb/ahb_master/Hresetn
/AHB_to_APB_Bridge_tb/ahb_master/Hrdata
/AHB_to_APB_Bridge_tb/ahb_master/Hreadyout
0
1
00000000000000...
x
St0
St1
10000000000000...
00000000000000...
00
000
StX
10
00000000000000000000000010000000


/AHB_to_APB_Bridge_tb/ahb_master/Haddr
/AHB_to_APB_Bridge_tb/ahb_master/Hwdata
/AHB_to_APB_Bridge_tb/ahb_master/Htrans
/AHB_to_APB_Bridge_tb/ahb_master/Hburst
/AHB_to_APB_Bridge_tb/ahb_master/Hsize
/AHB_to_APB_Bridge_tb/ahb_master/burst_write/i
/AHB_to_APB_Bridge_tb/apb_interface/clk
/AHB_to_APB_Bridge_tb/apb_interface/resetn
/AHB_to_APB_Bridge_tb/apb_interface/prdata
/AHB_to_APB_Bridge_tb/apb_interface/pready
/AHB_to_APB_Bridge_tb/apb_interface/pwrite
/AHB_to_APB_Bridge_tb/apb_interface/penable
/AHB_to_APB_Bridge_tb/apb_interface/paddr
/AHB_to_APB_Bridge_tb/apb_interface/pwdata
/AHB_to_APB_Bridge_tb/apb_interface/pselx
/AHB_to_APB_Bridge_tb/apb_interface/single_write/addr
/AHB_to_APB_Bridge_tb/apb_interface/single_write/data
/AHB_to_APB_Bridge_tb/apb_interface/single_read/addr
/AHB_to_APB_Bridge_tb/apb_interface/single_read/data
/AHB_to_APB_Bridge_tb/apb_interface/burst_write/base...
/AHB_to_APB_Bridge_tb/apb_interface/burst_write/burst...
/AHB_to_APB_Bridge_tb/apb_interface/burst_write/i

# CHAPTER 7: CONCLUSION

During my training at Maven Silicon, I learned how to design an AHB to APB Bridge to facilitate communication between the high-performance AHB bus and the slower APB bus. The bridge translates AHB signals (address, data, read/write) to APB protocol signals through a state machine. The design handles both read and write operations, ensuring efficient data transfer with low latency. The bridge optimizes performance while maintaining correct timing and synchronization between the two buses. This project provided me with hands-on experience in Verilog, VLSI design, and bus protocol conversions. I developed skills in designing state machines, handling signal timing, and verifying complex systems. The project reinforced my understanding of SoC design, equipping me with a solid foundation for future VLSI design and verification tasks.

# REFERENCES

Verilog HDL Book by Samir Palnitkar

Quartus Prime

https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

ModelSim

https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html