VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL -ELECTRONICS
DEPARTMENT OF ELECTRONICS

---------------o0o---------------

GRADUATION THESIS

IMPLEMENT AN AMBA AHB – APB UART INTERGRATED WITH HIGH BANDWIDTH ON-CHIP RAM ON RISC-V 32 BITS PROCESSOR

INSTRUCTOR: **Dr. Tran Hoang Linh**

Student's Name: **Vo Viet Hung**

Student's ID: **2051076**

Hồ Chí Minh city, Oct 2nd, 2024

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH   CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

TRƯỜNG ĐẠI HỌC BÁCH KHOA   Độc lập – Tự do – Hạnh phúc.

-----☆-----   -----☆-----

Số: ______ /BKĐT

Khoa: **Điện – Điện tử**

Bộ Môn: **Điện Tử**

NHIỆM VỤ LUẬN VĂN TỐT NGHIỆP

1. HỌ VÀ TÊN: **VÕ VIẾT HƯNG**   MSSV: **2051076**
2. NGÀNH: **THIẾT KẾ VI MẠCH – PHẦN CỨNG**   LỚP: **TT20HSA1**
3. Đề tài: IMPLEMENT AN AMBA AHB – APB UART INTERGRATED WITH HIGH BANDWIDTH MEMORY ON RISC-V 32 BITS PROCESSOR
4. Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):

   Thực hiện thiết kế và mô phỏng cách thức hoạt động của hệ thống AMBA AHB – APB UART bằng kiến trúc máy tính RISC-V 32 bits kết hợp với bộ nhớ với tốc độ cao trên kit tiêu chuẩn FPGA DE10.
5. Ngày giao nhiệm vụ luận văn: ngày 8 tháng 10 năm 2024
6. Ngày hoàn thành nhiệm vụ: ngày 09-12 tháng 12 năm 2024
7. Họ và tên người hướng dẫn: Phần hướng dẫn

   ....TS. Trần Hoàng Linh....................   ....

   ....

Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

Tp.HCM, ngày 02 tháng 11 năm 2024

CHỦ NHIỆM BỘ MÔN   **NGƯỜI HƯỚNG DẪN CHÍNH**

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):.......................
Đơn vị:..
Ngày bảo vệ : ....
Điểm tổng kết: ....
Nơi lưu trữ luận văn: ....

ACKNOWLEDGEMENTS

Dear Dr. Trần Hoàng Linh,

I hope this letter finds you well. I am writing to express my deepest gratitude for yours and Mr. Cao Xuân Hải (Teaching Assistant) for invaluable guidance and support throughout the course of my Major project. Your dedication, insight, and encouragement have been instrumental in helping me successfully complete this significant milestone.

Throughout the project, there were and are many challenges that I faced, both in terms of research and technical execution. However, your expertise and thoughtful feedback not only provided me with the direction I needed but also inspired me to dig deeper and approach problems with a more analytical and creative mindset. The constructive criticism you offered helped me refine my work, and the knowledge you shared has broadened my understanding far beyond the scope of this project.

I am truly grateful for the time and effort you invested in reviewing my progress and offering timely advice, even when your schedule must have been demanding. Your patience in answering my questions and your willingness to share your experiences made a tremendous difference in my learning journey. This project has taught me valuable lessons not just academically, but also in how to approach complex tasks and persevere through difficulties—skills that I am confident will benefit me greatly in the future.

Once again, thank you very much for everything. I greatly appreciate all that you have done, and I look forward to applying the lessons I have learned in my future endeavors. I wish you continued success in your teaching and research and hope that you continue to inspire and guide many more students in the years to come.

Sincerely,

Võ Viết Hưng

Hồ Chí Minh city, 2nd October, 2024

Student's Signature

[illegible]

Võ Viết Hưng

THESIS ABSTRACT

This project presents the implementation of AHB-APB and APB-UART bridges, aiming to establish efficient communication between Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB), with a specific focus on UART (Universal Asynchronous Receiver-Transmitter) interfacing. The primary objective is to design and verify a system that bridges the high-speed AHB to the slower APB, enabling seamless data transfer to UART devices.

The project includes the design of the AHB-APB bridge to translate high-speed transactions from the AHB master to the APB peripherals. Additionally, the APB-UART bridge is implemented to facilitate serial communication by managing data transmission and reception between the processor and external UART devices.

Comprehensive simulation and validation were carried out using such tools/verification methods: "SystemVerilog HDL via Quartus II as a compiler and examinate basic testbenches on ModelSim to verify its operation", ensuring the accuracy of data transfer and correct operation of the UART protocol. The results demonstrate that the system meets the required performance standards, including proper handling of data flow, synchronization, and signal integrity.

This project contributes to the field of embedded systems by providing a reliable solution for bridging high-speed and low-speed buses, with potential applications in various communication systems requiring UART interfacing.

TABLE OF CONTENT

1. INTRODUCTION

1.1. Topic's selection reason

In this Graduation Thesis Project, I chose to design an AMBA (Advanced Microcontroller Bus Architecture) system featuring AHB (Advanced High-performance Bus) to APB (Advanced Peripheral Bus) UART (Universal Asynchronous Receiver-Transmitter) integrated with High Bandwidth On-Chip RAM stems from the growing need for efficient, reliable and scalable communication in embedded systems.

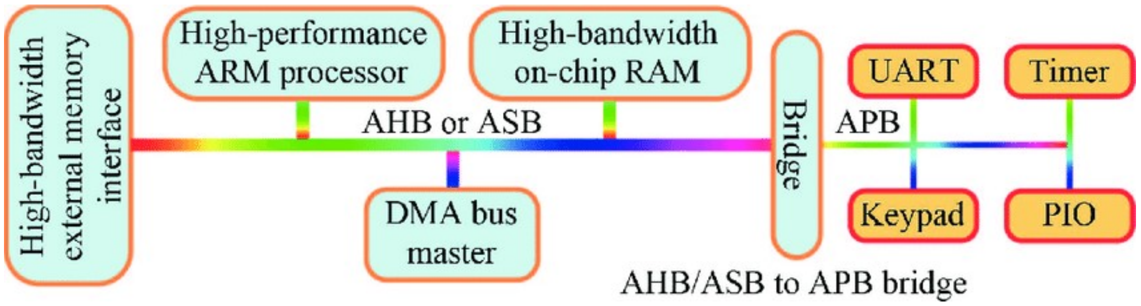

High-bandwidth external memory interface
High-performance ARM processor
High-bandwidth on-chip RAM
AHB or ASB
DMA bus master
Bridge
APB
UART
Timer
Keypad
PIO
AHB/ASB to APB bridge

Figure 1.1: A typical AMBA based SoC design

To more specific, AHB provides high-speed communication suitable for processors and memory interfaces, while APB offers a simpler, lower-power connection to peripherals. By integrating UART, a widely used communication protocol, into this setup, the project aims to facilitate robust and reliable serial communication between the system and external devices. This design ensures correctness data on both transfer and receiver across different customizing of data frame, enhancing the overall performance and versatility of the SoC (System on Chip). In a result, this project will not only focus on creating a cohesive and efficient bridge between these components, but also implementing part of operating process of realistic AMBA which making it a critical study for modern embedded system applications.

1.2. Topic's aim

- ***Implement a pipeline RISV-V 32 bits Processor Architecture***: By using the RV32I instruction set. The processor will feature a five-stage pipeline—Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write-Back—to improve execution speed by processing multiple instructions simultaneously. Key design elements include a control unit for managing control signals, an ALU for arithmetic operations, and mechanisms to handle data and control hazards. Each stage will be designed as a separate module in SystemVerilog, with testing and validation focusing on accurate instruction handling and performance efficiency.

- ***Design and Implementation of the AHB-APB Bridge***: The primary goal is to develop an efficient bridge that enables communication between the high-speed Advanced High-performance Bus (AHB) and the lower-speed Advanced Peripheral Bus (APB). This bridge should ensure seamless data transfer between the two buses while translating the signals and protocols between them.

- ***Development of the APB-UART***: The project aims to implement an APB-UART to facilitate serial communication between the system and external devices. This communication protocol will manage the data flow from the APB to the UART peripheral, ensuring accurate data transmission and reception in accordance with the UART protocol.

- ***Design an High Bandwidth On-Chip RAM***: Implement a multi-bank, dual-port architecture that enables parallel access, enhancing throughput. A high-frequency clock and pipelined access further reduce latency, while burst-mode access optimizes data transfer efficiency. Employing a row buffer minimizes repeated access delays, and error correction (ECC) improves reliability. Using a high-speed bus interface (AHB) ensures efficient memory transactions, while power management techniques such as clock gating and voltage scaling help control power and thermal impacts, making the RAM both fast and reliable for high-performance applications.

- ***Verification and Validation of the System***: Comprehensive testing and simulation will be conducted to verify the correctness and reliability of the design. The objective is to use simulation tools such as SystemVerilog and testbenches to rigorously validate the bridge's performance under various operational conditions, ensuring that it meets the system's requirements.

- ***Practical Application in Embedded Systems***: The final objective is to ensure that the implemented bridges can be practically applied to real-world embedded systems, particularly those requiring UART communication. The design should be flexible and robust enough to be integrated into systems that need efficient communication between high-speed processors and low-speed peripheral devices via UART.

2. THEORICTICAL OF PROJECT

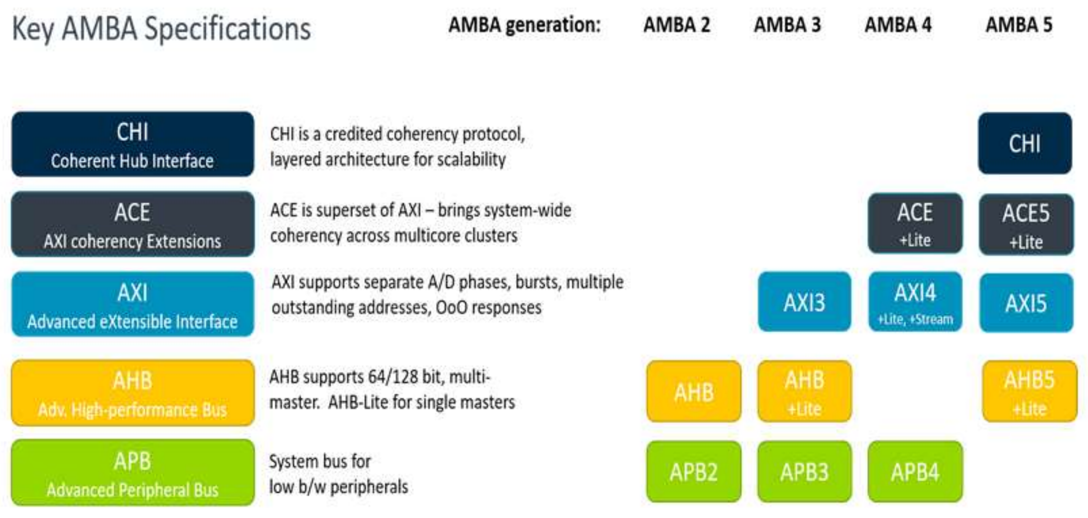

Key AMBA Specifications
AMBA generation:
AMBA 2
AMBA 3
AMBA 4
AMBA 5
CHI
Coherent Hub Interface
CHI is a credited coherency protocol, layered architecture for scalability
ACE
AXI coherency Extensions
ACE is superset of AXI – brings system-wide coherency across multicore clusters
AXI
Advanced eXtensible Interface
AXI supports separate A/D phases, bursts, multiple outstanding addresses, OoO responses
AHB
Adv. High-performance Bus
AHB supports 64/128 bit, multi-master. AHB-Lite for single masters
APB
Advanced Peripheral Bus
System bus for low b/w peripherals
CHI
ACE +Lite
ACE5 +Lite
AXI3
AXI4 +Lite, +Stream
AXI5
AHB
AHB +Lite
AHB5 +Lite
APB2
APB3
APB4

Figure 2: What are AMBA protocols

2.1. What is AMBA AHB & APB and its specification AHB – APB Bridge?

2.1.1. AMBA AHB (Advanced High-performance Bus)

AHB is designed to handle high-performance, high-speed communication between system components like the CPU, memory, and high-speed peripherals (e.g., DMA, bridges).

- ***Architecture***: AHB is a single-channel bus that uses a **centralized arbiter** to manage multiple bus masters (components that can initiate data transfers). Only one master can control the bus at a time, ensuring organized communication.

- ***Key Features:***

+ **Pipelined Operation**: AHB supports pipelining, allowing multiple operations to overlap for higher data throughput.

+ **Burst Transfers**: It supports burst mode to transfer multiple data items in a single transaction, reducing the need for repeated bus arbitration.

+ **Single Clock Edge**: It operates on a single rising clock edge to ensure simpler timing.

+ **Multiple Masters**: It allows multiple bus masters, each capable of initiating a transfer (e.g., CPU, DMA controller).

- *Use Case*: AHB is ideal for connecting high-speed components, such as the CPU, system memory, and other high-bandwidth devices, where fast data transfers and efficient bus management are critical.

2.1.2. AMBA APB (Advanced Peripheral Bus)

APB is designed for communication with **low-power, low-speed peripheral devices** such as UARTs, I2C, SPI, timers, or GPIOs that don't require the high-performance features of AHB.

- **Architecture**: APB uses a **simple and low-power interface**. It is usually connected to AHB through a **bridge** that converts high-speed AHB transactions to lower-speed APB transactions.

- **Key Features**:

+ **Low Power and Low Complexity**: APB doesn't support burst transfers or pipelined operations. This makes it simpler and more power-efficient.

+ **Single Master and Slave**: APB follows a simple master-slave architecture, where the APB bridge acts as the master, and peripherals are the slaves.

+ **Address and Data Phases**: The APB uses distinct address and data phases, meaning the address is supplied in one cycle and data is transferred in subsequent cycles, reducing speed but simplifying design.

+ **No Arbitration**: Since APB is a simple interface with typically a single bus master (the APB bridge), there's no need for bus arbitration or complex timing requirements.

- Use Case: APB is typically used to connect low-bandwidth peripherals that don't require high-speed transfers, making it ideal for power-sensitive areas of the SoC.

2.1.3. AHB – APB Bridge

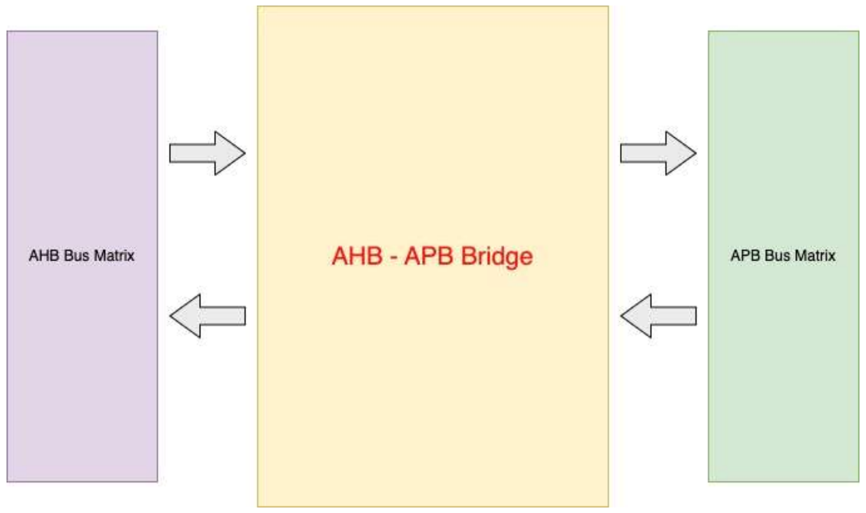


Figure 2.1.3: AHB – APB Bridge reference block (in general view)

The **AHB-APB bridge** is essential for translating communication between the high-speed AHB bus and the lower-speed APB bus. It acts as an intermediary, allowing devices connected to the AHB bus (like the CPU or DMA controller) to communicate with APB peripherals (like UART or GPIOs).

For more precisely, here is the general table for further detail of AHB – APB Bridge:

Feature	**AHB (Advanced High-performance Bus)**	**APB (Advanced Peripheral Bus)**
Purpose	High-speed communication for processor and memory	Low-power, low-speed peripheral communication
Bandwidth	High bandwidth	Low bandwidth
Latency	Low latency	Higher latency compared to AHB
Complexity	More complex	Simpler design

Feature	AHB (Advanced High-performance Bus)	APB (Advanced Peripheral Bus)
Power Consumption	Higher	Lower
Usage	Main bus for CPU, memory, and high-speed devices	Connecting peripheral devices like sensors
Clocking	Synchronous with the system clock	Can be asynchronous or synchronous
Data Transfer	Supports burst transfers and pipelined operations	Simple, single data transfers
Control Signals	More control signals (e.g., HTRANS, HBURST, HPROT)	Fewer control signals
Protocol Complexity	More complex protocol	Simpler protocol
Implementation Cost	Higher	Lower
Example Devices	CPUs, DMA controllers, high-speed peripherals	UART, GPIO, timers, lower-speed peripherals

Table 2.1.3: A typical AMBA AHB and APB characteristic

- Key Functions of the AHB-APB Bridge:

+ **Clock Domain Crossing**: AHB and APB may operate at different clock speeds. The bridge helps synchronize transactions between these two domains, ensuring data is transferred accurately.

+ **Protocol Translation**: The AHB and APB protocols are different in terms of how data is addressed and transferred. The bridge translates high-performance **pipelined** AHB transactions into the simpler **non-pipelined** APB format.

AHB uses a single-phase clock cycle with overlapping address and data phases (pipelined).

APB uses distinct address and data phases in separate cycles (non-pipelined).

+ **Address Decoding**: When a master (like the CPU) on the AHB bus wants to access a peripheral on the APB bus, the bridge decodes the address to determine which APB peripheral should receive the request.

+ **Latency Management**: Because the APB operates slower than AHB, the bridge manages the handshaking between buses, ensuring that the AHB master waits for the APB peripheral to complete its operation before continuing. This is important to maintain system stability and prevent data corruption.

- Advantages of Using AHB and APB Together:

+ **Efficiency**: High-performance components like CPUs can work at full speed on the AHB bus without being slowed down by lower-speed peripherals. The APB bus allows for simpler, lower-power communication with peripherals like UART, reducing the overall system power consumption.

+ **Scalability**: By separating the high-speed AHB bus from the low-speed APB bus, more peripherals can be added without significantly impacting the performance of the core system. The AHB-APB bridge can handle the necessary translation between the two buses.

+ **Simplified Design**: APB peripherals are typically simpler in design, reducing the complexity and cost of adding peripherals to the SoC. The bridge ensures that the complexity of translating signals and data is managed centrally.

2.2. What is UART (Universal Asynchronous Receiver-Transmitter)?

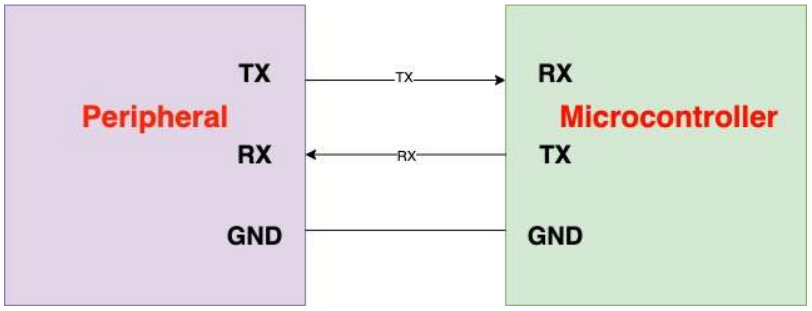


Figure 2.2: UART Receiving and Transmitting reference block (in general view)

UART is a hardware communication protocol used for serial communication between devices. It is widely used in embedded systems and computers for sending and receiving data over a serial interface.

2.2.1. General Characteristic of UART

- Key Concepts of UART:

+ **Serial Communication**: UART communicates data one bit at a time, sequentially, over a single data line. This contrasts with parallel communication, where multiple bits are transmitted simultaneously over multiple lines.

+ **Asynchronous Communication**: UART does not use a clock signal to synchronize the sender and receiver. Instead, it relies on start and stop bits to delimit the data frame, making it simpler and more flexible but potentially less accurate over long distances.

- Components of UART:

+ **Transmitter (TX)**: This component sends data out of the UART device. It takes parallel data from a parallel-to-serial converter and sends it bit by bit over the serial line.

+ **Receiver (RX)**: This component receives data into the UART device. It takes serial data from the serial line and converts it back into parallel data for processing by the device.

+ **Baud Rate**: This is the rate at which data is transmitted over the UART interface, typically measured in bits per second (bps). Both the transmitting and receiving UARTs must be configured with the same baud rate to ensure proper communication.

+ **Start Bit**: A single bit sent at the beginning of each data frame, indicating the start of the transmission. It is typically a low-level signal.

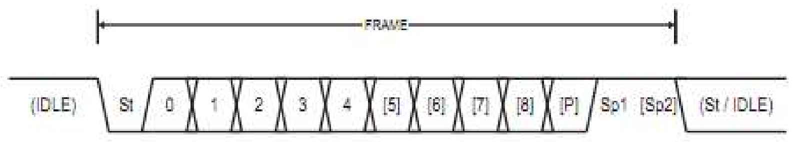
+ **Data Bits**: The actual data being transmitted. UARTs commonly use 5, 6, 7 or 8 data bits per frame.

+ **Parity Bit**: An optional error-checking bit that can be included in the data frame to detect transmission errors. It can be even, odd, or none, depending on the configuration.

+ **Stop Bits**: One or more bits sent at the end of each data frame to signal the end of the transmission. They are typically high-level signals.

2.2.2. How UART Works

Figure 19-4. Frame Formats



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

Figure 2.2.2: UART Receiving and Transmitting operation

- Data Framing: When transmitting data, the UART converts parallel data into a serial stream. It adds a start bit (normally LOW), the data bits, an optional parity bit, and one or more stop bits (normally HIGH) to form a complete data frame.

- **Transmission**: The data frame is sent bit by bit over the serial line. The receiving UART detects the start bit, then reads the data bits, checks the parity bit (if used), and looks for the stop bits to determine the end of the data frame.

- **Reception**: The receiving part of UART will check whether to receive a serial of data, which is detected by start bit (HIGH -> LOW), then it will generally accept and convert the serial of data in to parallel and ended by a stop bit (HIGH), however, if there is any requirement for parity bit checking in order to verification, the data will be stored as a temporary and wait until the checking bit process done, then if satisfied, the data will be collected, otherwise, flushing it away and await for the next frame in.

2.2.3. Applications of UART

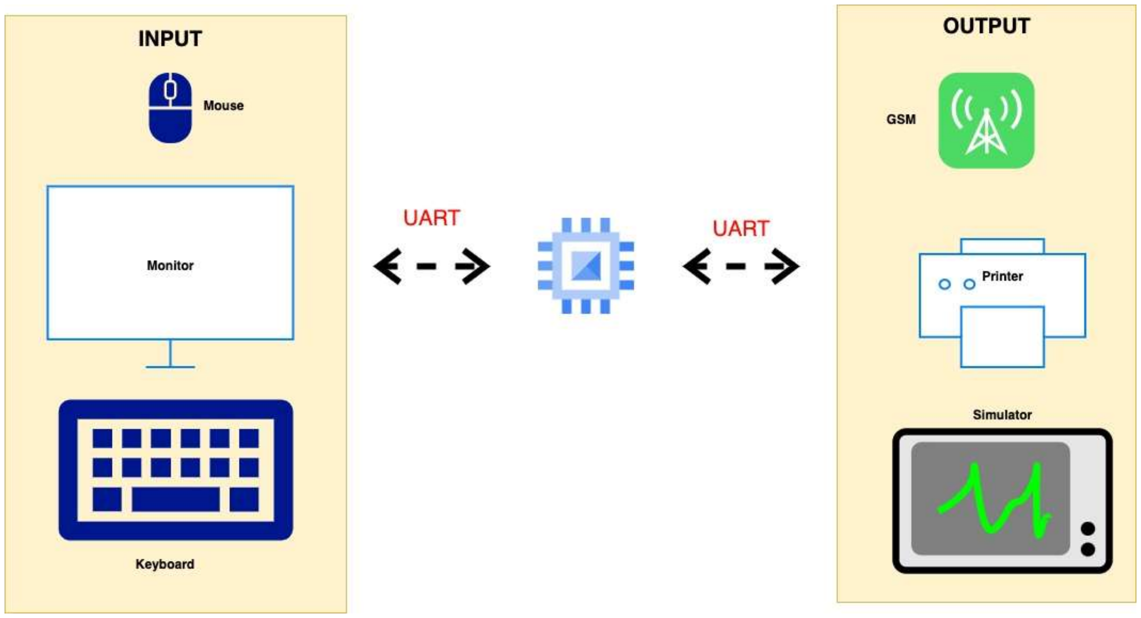


Figure 2.2.3: UART Protocol Suggesting Application

- **Embedded Systems**: UART is commonly used for communication between microcontrollers and peripherals like sensors, GPS modules, and communication modules (e.g., GSM, Bluetooth).

- **Computers**: Historically, UARTs were used for serial ports on computers for connecting modems, mice, and other peripherals. Today, UART communication is often used for debugging and low-speed data exchange.

- **Networking**: UART is used in various communication interfaces within networking equipment for configuration and data transfer.

2.2.4. Advantages of UART

- **Simplicity**: UART is relatively simple to implement compared to synchronous serial protocols, as it doesn't require a clock signal.

- **Flexibility**: It can handle variable data rates and different data formats, making it versatile for various applications.

- **Cost-Effective**: UART communication typically requires fewer components and less circuitry compared to parallel communication methods.

2.2.5. Disadvantages of UART

- **Distance Limitation**: UART is generally suitable for short-distance communication due to its reliance on asynchronous timing, which can lead to synchronization issues over long distances.

- **Data Rate Limitations**: The maximum data rate is limited by the baud rate and the quality of the communication line.

- **Error Detection**: While UART can use parity bits for error detection, it does not provide robust error correction mechanisms compared to more advanced protocols.

3. REGISTER TRANSFER LEVEL DESIGN AND IMPLEMENTATION

3.1. AHB - APB Bridge

3.1.1. AHB - APB Finite State Machine

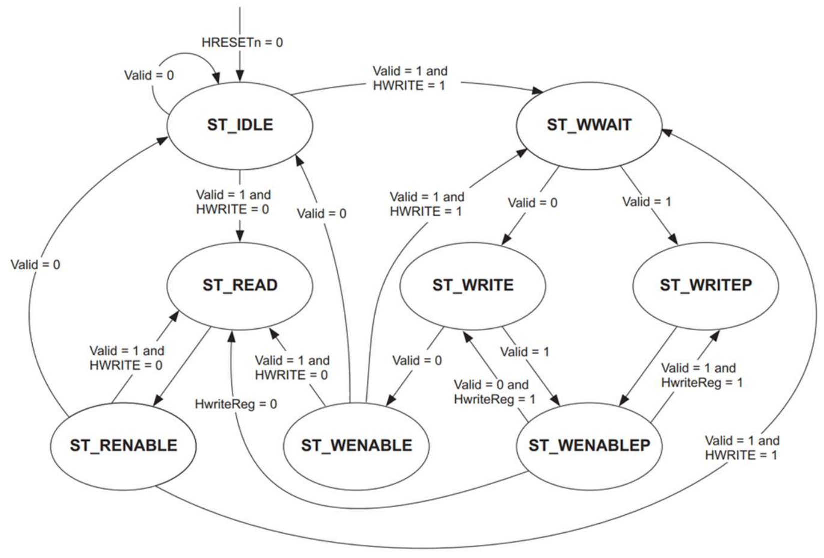


Figure 3.1.1: AHB – APB Bridge Finite State Machine

3.1.1.1. Explanation:

- ***ST_IDLE***: The FSM starts in this idle state, waiting for a valid request from the AHB side.

Transition:

+ If "**Valid** = 1" and "**HWRITE** = 0", it transitions to the **ST_READ** state (indicating a read operation).

+ If “**Valid** = 1” and “**HWRITE** = 1”, it transitions to the **ST_WWAIT** state (for a write operation).

+ The FSM remains in **ST_IDLE** if “**Valid** = 0”.

- ***ST_WWAIT***: A temporary waiting state before writing.

Transition:

+ If “**Valid** = 1” and “**HWRITE** = 1”, it moves to **ST_WRITE**.

+ If “**Valid** = 0”, it remains in **ST_WWAIT**.

- ***ST_WRITE***: Manages the actual write operation from AHB to APB.

Transition:

+ If “**Valid** = 0”, it returns to **ST_IDLE.**

+ If “**Valid** = 1”, it proceeds to **ST_WRITEP** for potential write enable processes.

- **ST_WRITEP**: Prepares or completes the write process.

Transition:

+ If “**Valid** = 1” and “**HWRITEReg** = 1”, it transitions to **ST_WENABLE** or **ST_WENABLEP** for enabling the write.

- **ST_WENABLE** and **ST_WENABLEP**: Handles enabling the write process for the APB side, ensuring data is ready for the peripheral.

Transition:

+ After enabling, it may go to **ST_RENABLE** if another read needs to happen, or return to **ST_IDLE** if the process is done.

- ***ST_READ***: Performs the read operation from APB to AHB.

Transition:

+ If “**Valid** = 1”, it goes to **ST_RENABLE** to handle enabling the read operation.

+ If “**Valid** = 0”, it returns to **ST_IDLE**.

- ***ST_RENABLE***: Enables the read process for the APB side, allowing the system to fetch data from the peripheral.

Transition:

+ If the read is successful, it transitions back to **ST_IDLE**.

3.1.1.2. General Overview:

- The FSM handles both read and write operations from the AHB bus to the APB bus.

- The states transition based on the validity of the signal ("**Valid** = 1") and whether it's a write ("**HWRITE** = 1") or read ("**HWRITE** = 0") operation.

- Write operations involve waiting and enabling states to ensure data integrity during the transaction.

- Read operations transition more directly but also involve enabling the read access for APB peripherals.

- This FSM ensures proper protocol conversion between the AHB and APB buses, managing timing and control flow to ensure valid data transfers.

3.1.2. AHB – APB Address Decoder

- Address Mapping:

+ The address decoder maps a specific range of AHB addresses to APB peripherals.

+ Each APB peripheral is assigned a unique address range, and the decoder matches the incoming address from the AHB side to the corresponding APB peripheral.

- Peripheral Selection:

+ Once the address is decoded, the decoder generates a **select signal** (**PSELx**) that enables the appropriate APB peripheral.

+ Only the selected peripheral responds to the APB read or write operation, while other peripherals remain inactive.

- Control Signal Generation:

+ In addition to generating the peripheral select signal (**PSELx**), the decoder also produces other control signals such as **PENABLE** (to enable the peripheral) and **PREADY** (to indicate the peripheral is ready for data transfer).

+ These control signals ensure proper handshaking between the AHB bus and the selected APB peripheral.

- Address Validity:

+ The decoder checks whether the incoming AHB address falls within the valid range of APB peripherals. If not, it will ignore the request or generate an error signal to indicate an invalid access.

3.1.3. D – Flip Flop block for Signals' synchronizing

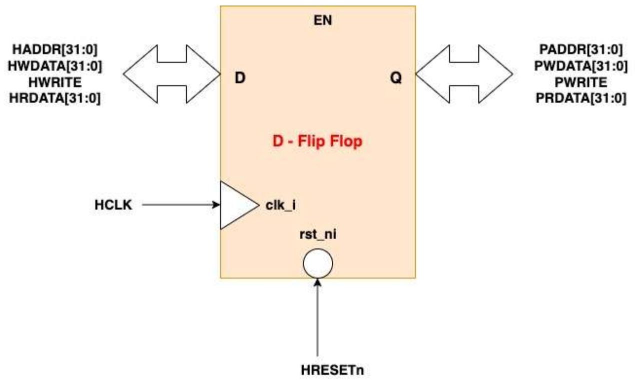


Figure 3.1.3: D – Flip Flop for Signal not generated from AHB – APB Bridge FSM

In an AHB-APB Bridge, a D Flip-Flop (D - FF) can be used to transfer or synchronize signals from the high-speed AHB (Advanced High-performance Bus) to the lower speed APB (Advanced Peripheral Bus). This is necessary because the AHB and APB operate at different clock domains or data rates. D flip-flops ensure that signals are captured, stored, and transferred cleanly between the two buses, preventing timing issues such as metastability.

3.1.3.1. Why Use D Flip-Flops in the AHB-APB Bridge?

- *Clock Domain Crossing*: The AHB typically operates at a higher clock frequency than the APB. When transferring signals between buses operating at different speeds, it's important to synchronize the signals properly to avoid timing problems. D flip-flops help in crossing the clock domains safely.

- Data Latching: The DFF can latch the data from the AHB bus, hold it stable, and then transfer it to the APB side. This prevents signal glitches or data corruption during transfer from AHB to APB.

- **Synchronous Transfer**: D flip-flops enable synchronous data transfer between the AHB and APB by ensuring that data is only passed when the receiving clock is ready to sample it.

3.1.3.2. D Flip-Flop Usage in AHB-APB Bridge

- *AHB Data Signals*: Data signals from the AHB bus (like HADDR and HWDATA), which are high-speed, are typically passed through D flip-flops before being sent to the APB bus as PADDR (APB Address) and PWDATA (APB Write Data).

- *Control Signals*: Critical control signals such as HWRITE (indicating AHB read/write operation), HTRANS (transfer type), and HREADY (ready signal) from the AHB side are captured using D flip-flops to ensure these signals are synchronized before they are used on the APB side.

- State Holding: The D flip-flops also store the current state of the transaction (whether it's a read or write operation), ensuring the correct behavior during the bridging process.

- Synchronizing Signals: Important handshake signals between the AHB and APB, like **HREADY** and **PREADY**, are passed through D flip-flops to ensure reliable handshaking between the two buses.

3.2. APB UART

3.2.1. APB Interface

3.2.1.1. APB Interface Finite State Machine

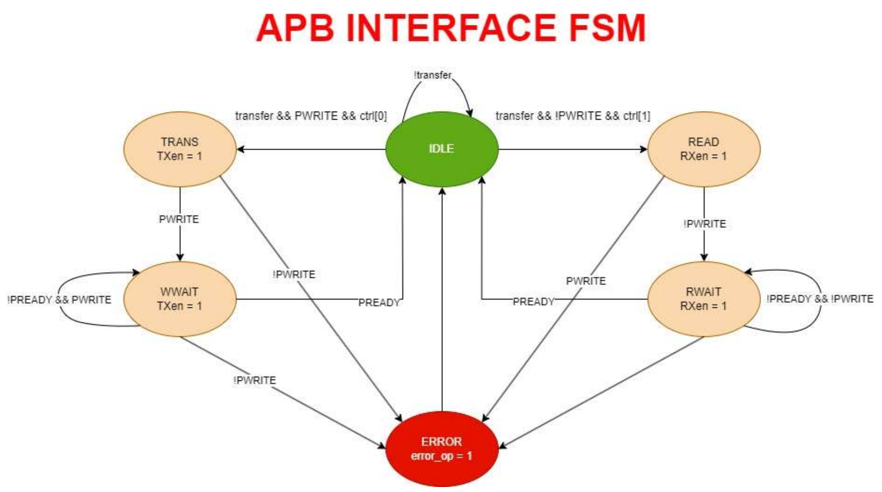


Figure 3.1.1: AHB – APB Bridge Finite State Machine

States and Transitions in the APB Interface FSM:

- **IDLE**: This is the starting or resting state, where the system waits for a transfer request.

Transition:

+ If "**transfer**" is active and "**PWRITE** = 1" (indicating a write operation), the FSM moves to the **TRANS** state.

+ If "**transfer**" is active and "**PWRITE** = 0" (indicating a read operation), the FSM moves to the **READ** state.

- **TRANS** (Transaction): This state handles the write operation with "**TXen** = 1" (transmit enable is active).

Transition:

+ If "**PWRITE** = 1", it proceeds to the **WWAIT** state (Write Wait) to wait for the peripheral to be ready.

+ If there is an invalid or incomplete write operation, it moves to the **ERROR** state.

- **WWAIT** (Write Wait): This state waits for the APB peripheral to be ready to accept the write data.

Transition:

+ If "**PREADY**" and "**PWRITE**" signals are both valid, it returns to the IDLE state, meaning the write operation was completed successfully.

- **READ**: This state handles the read operation with "**RXen** = 1" (receive enable is active).

Transition:

+ If the peripheral is not ready to respond ("**PREADY** = 0"), it moves to the **RWAIT** state (Read Wait).

+ If the read is invalid or incomplete, it moves to the **ERROR** state.

- **RWAIT** (Read Wait): This state waits for the APB peripheral to be ready for the read operation.

Transition:

+ When "**PREADY** = 1", meaning the peripheral is ready, the FSM returns to **IDLE** to signal the read operation is complete.

+ If the read or wait operation is invalid, it moves to the **ERROR** state.

- **ERROR**: The FSM enters this state when there is an error in the read or write operation.

Transition:

+ Once the error is addressed, the FSM will send a signal error as a flag and transition back to the **IDLE** state to reset the operation and be ready for the next request.

Key Signals:

- **PWRITE**: Indicates whether the operation is a write ("**PWRITE** = 1") or read ("**PWRITE** = 0").

- **TXen**: Transmit enable, used in write operations.

- **RXen**: Receive enable, used in read operations.

- **PREADY**: Ready signal from the peripheral, indicating whether it is ready to accept (read) or provide (write) next data.

- **ctrl[1:0]:** Control signals that further define the transfer and error conditions.

Summary:

- The FSM starts in the **IDLE** state and transitions to **TRANS** for write operations or **READ** for read operations.

- After a transfer request, it waits in either **WWAIT** or **RWAIT** for the peripheral to be ready.

- If the operation completes successfully, it returns to **IDLE**. If there's a failure, the FSM enters the **ERROR** state to handle the error and return to **IDLE** for recovery.

This FSM efficiently manages read and write operations between the AHB and APB interfaces, ensuring that proper handshaking and error handling are in place.

3.2.1.2. APB Interface Address Register

The APB Interface Address Register for the UART consists of a series of registers that enable control over the APB UART's operations. These registers allow for reading and writing data and configuring the UART by addressing specific memory locations. Below is a description of each register:

Address	Register	Description	PWRITE = 1 (Write)	PWRITE = 0 (Read)
0x00	Data Register	Holds the data to be transmitted or received	Write data to be transmitted	Read received data
0x04	Control Register	Controls the UART's operation (enable, config, etc.)	Write control signals	Read current configuration
0x08	State Register	Holds the state of the UART (ready, busy, error, etc.)	Write to enable certain states or conditions	Read the current state of the UART
0x10	Counter Divisor	Sets the divisor for the Baud rate generator	Write divisor value for Baud rate	Read current Baud rate divisor value

Table 3.2.1.2: APB Interface Address Register detail explanation

3.2.2. UART Baud Rate Generator

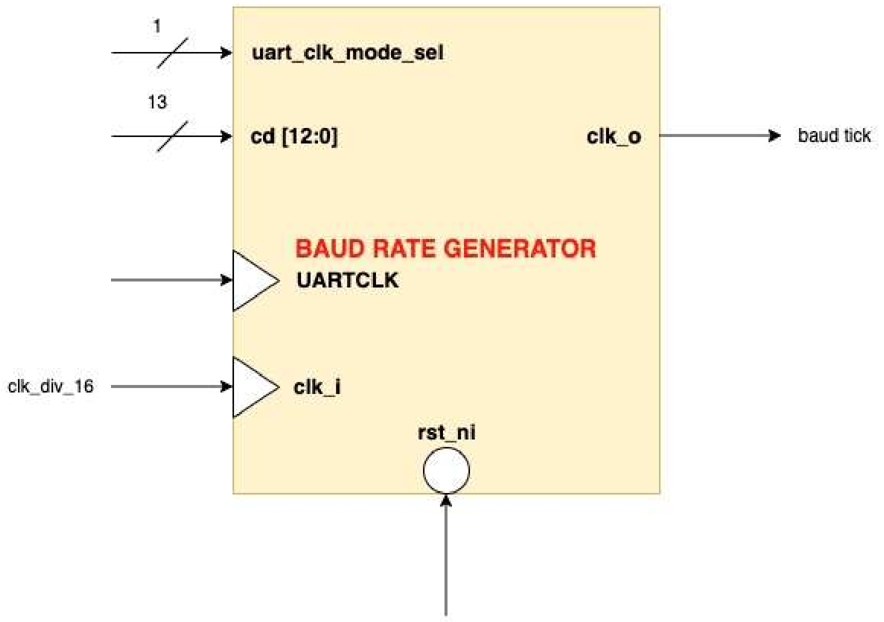


Figure 3.2.2: APB UART Baud Rate Generator block

The UART Baud Rate Generator is a crucial component that controls the speed of communication in UART (Universal Asynchronous Receiver/Transmitter) by generating the clock signal used to send and receive data. The baud rate defines the number of bits transmitted or received per second, and the generator ensures that this rate is consistent across the communication link.

3.2.2.1. How the UART Baud Rate Generator Works:

- ***Clock Input***: The baud rate generator receives a system clock (e.g., 50 MHz) as input. The system clock is typically much faster than the required baud rate for UART communication.

- ***Divisor***:

+ To achieve the correct baud rate, the system clock is divided by a specific divisor value. This divisor is set in a special register, typically the counter divisor register (like the one at address “0x10” in the APB Interface).

+ The baud rate can be calculated using the formula:

$$\text{Baud Rate} = \frac{\text{System Clock}}{\text{Divisor Value} \times 16}$$

Here, the factor of 16 accounts for the oversampling rate commonly used in UART communication.

- ***Configurable Divisor***:

+ The divisor value can be written to the Baud Rate Divisor Register (as seen at “0x10” in design) to configure the baud rate.

+ By changing the divisor, you can adjust the baud rate to match the required speed for UART communication.

- **Oversampling**:

+ UART communication typically involves oversampling of the received signal to ensure accurate data capture. The most common oversampling rate is 16x. This means the baud rate clock is derived from dividing the system clock by 16 and the divisor.

+ Some UART systems allow oversampling to be configured for even more accuracy (e.g., 8x or 32x oversampling), but 16x is standard.

3.2.2.2. Example Calculation:

If the system clock is 50 MHz and you want to achieve a baud rate of 115200 bps (bits per second), the divisor would be calculated as:

$$\text{Divisor} = \frac{50\,\text{MHz}}{115200 \times 16} \approx 27.1267$$

+ In this case, the divisor can be rounded to 27.

+ Writing this value to the Baud Rate Divisor Register configures the UART to communicate at approximately 115200 bps.

No.	Desired Baudrate	Baudrate divisor
1.	600	5208
2.	4800	651
3.	9600	325
4.	19200	162
5.	28800	108
6.	38400	81
7.	57600	54
8.	115200	27
9.	230400	13
10.	460800	6
11.	921600	3

Table 3.2.1: Baudrate counter divisor (cd) references

Summary:

- The UART Baud Rate Generator ensures synchronized communication between the UART and peripheral devices by generating the clock signal at the required baud rate.

- It uses a divisor to adjust the system clock down to the appropriate frequency.

- Configurable baud rates allow flexibility in UART communication to meet the requirements of different systems and devices.

3.2.3. TX – RX FIFO (Transmit and Receive First In, First Out Buffers)

In UART communication, data is transmitted and received in a sequential manner, one bit at a time. The TX (Transmit) FIFO and RX (Receive) FIFO buffers are used to temporarily store data during this process, ensuring smooth data transfer between the system and the UART without loss of data due to timing mismatches.

3.2.3.1. TX - RX FIFO (Transmit FIFO)

The Transmit - Receive FIFO stores data that needs to be transmitted or received via the UART. It operates on the principle of First In, First Out (FIFO), meaning the first data byte written into the FIFO is the first one to be transmitted, and so on for receiving side. This helps in decoupling the CPU from the transmission process, as the CPU can write multiple bytes to the FIFO, and the UART will handle the transmission and receiving asynchronously.

- ***Key Features of RX - TX FIFO***:

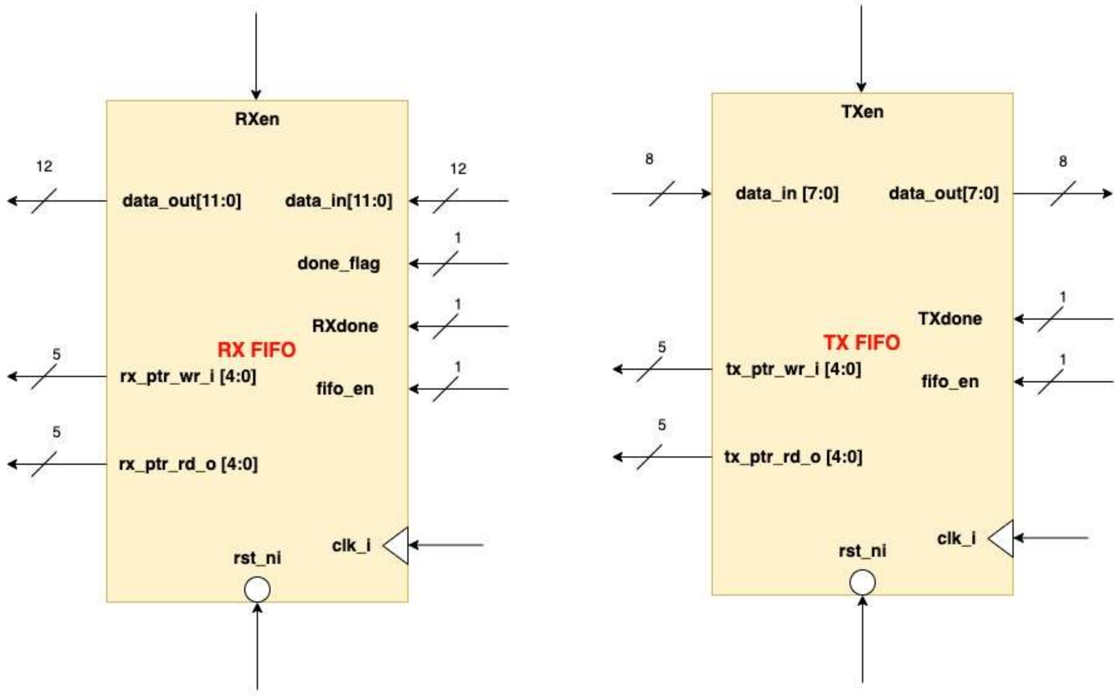


Figure 3.2.3: APB UART RX - TX FIFO

+ **Buffering**: The CPU can write/read data via the TX - RX FIFO buffer at a faster rate, while the UART transmits the data at the configured baud rate. This prevents the CPU from having to wait for each byte to be sent before writing the next byte.

+ **Size**: The size of the FIFO is typically configurable (32 bytes), and the more space available, the less often the CPU has to handle the transmit buffer.

+ **Empty/Full Flags**:

RX/TX Empty Flag: Indicates when the FIFO is completely empty and ready for new data.

RX/TX Full Flag: Indicates when the FIFO is full and cannot accept more data until some data has been transmitted.

3.2.3.2. TX - RX FIFO Operation:

- Write: The CPU or a higher-level controller writes data to the TX FIFO.

- **Transmission**: The UART reads the data from the TX FIFO and transmits it serially, one byte at a time, at the specified baud rate.

- **Receive**: The UART receives incoming data and stores it in the RX FIFO buffer.

- **Read**: The CPU or higher-level controller reads data from the RX FIFO at a convenient time.

- **Buffer Monitoring**: The UART will keep transmitting from the FIFO until it is empty. On the other hand, The UART continues to store incoming data into the FIFO until it is full, after which it either generates an error (overflow) or waits for the CPU to read the buffer.

3.2.3.3. Benefits of TX-RX FIFOs:

- **Efficiency**: FIFO buffers decouple the CPU from the UART, allowing asynchronous data transmission and reception. This improves system performance as the CPU doesn't need to handle every byte of data individually in real-time.

- **Prevention of Data Loss**: With FIFOs, even if the CPU is busy with other tasks, incoming and outgoing data is safely stored in the buffer, preventing data loss.

3.2.4. RX Finite State Machine

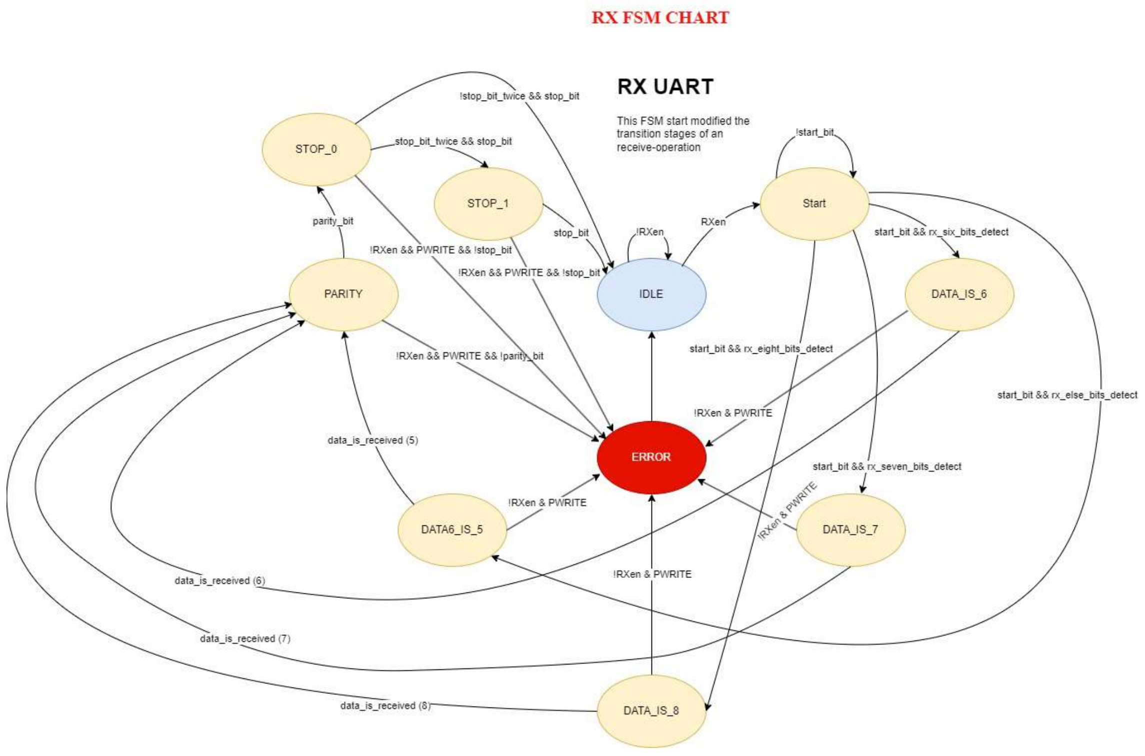

RX FSM CHART
RX UART
This FSM start modified the transition stages of an receive-operation
STOP_0
STOP_1
PARITY
IDLE
Start
DATA_IS_6
ERROR
DATA6_IS_5
DATA_IS_7
DATA_IS_8
!stop_bit_twice && stop_bit
stop_bit_twice && stop_bit
!start_bit
parity_bit
stop_bit
!RXen
RXen
start_bit && rx_six_bits_detect
!RXen && PWRITE && !stop_bit
!RXen && PWRITE && !stop_bit
start_bit && rx_eight_bits_detect
!RXen && PWRITE && !parity_bit
start_bit && rx_else_bits_detect
!RXen & PWRITE
data_is_received (5)
start_bit && rx_seven_bits_detect
!RXen & PWRITE
!RXen & PWRITE
!RXen & PWRITE
data_is_received (6)
data_is_received (7)
data_is_received (8)

Figure 3.2.4: APB UART RX Finite State Machine

Key States of the RX FSM:

- **IDLE**: The FSM starts in the **IDLE** state, where it waits for the start of a transmission.

Transition:

+ If the RX side of UART is enable "**RXen**", the FSM moves to the **START** state.

- **START**: This state checks for the start bit, which signals the beginning of data transmission.

Transition:

+ If the start bit is valid and RX data ("**rx_eight_bits_detect**", "**rx_seven_bits_detect**", "**rx_six_bits_detect**" or "**rx_else_bits_detect**") is detected, it moves to the **DATA_IS_6**, **DATA_IS_7**, **DATA_IS_8** or **DATA_IS_5** states, depending on the number of bits desired to be received.

- **DATA_IS_5, DATA_IS_6, DATA_IS_7, DATA_IS_8**: These states correspond to the reception of data bits. The FSM processes and receives the data bits from the UART input.

Transition:

+ Once the data is fully received ("**data_is_received**"), the FSM transitions based on the configuration:

+ If parity is enabled, it moves to the **PARITY** state.

+ If parity is disabled, it moves to the **STOP_0** state for checking the stop bits.

- **PARITY**: This state handles the checking of the parity bit (if enabled).

Transition:

+ After checking parity, the FSM transitions to the **STOP_0** state to verify the stop bits.

- **STOP_0** and **STOP_1**: These states are used to check the presence of one or two stop bits at the end of the data transmission.

Transition:

+ The FSM checks the stop bit and without customer characterize ("**stop_bit**" and "**!stop_bit_twice**"). If it's valid, the FSM returns to the IDLE state. Else if stop bit and customer characteristic ("**stop_bit**" and "**stop_bit_twice**"), the FSM turns to **STOP_1** and continue previous process.

+ If the stop bit is invalid, it transitions to the **ERROR** state.

- **ERROR**: If any error is detected during the reception process (invalid start bit, parity error, or stop bit error), the FSM transitions to the **ERROR** state.

Transition:

+ In the **ERROR** state, the operation is stopped, and the FSM must be reset or return to **IDLE** after the error is handled.

Summary:

- The FSM starts in IDLE and transitions based on the detection of a start bit.

- The data is received in the DATA_IS_X states, with different states based on the number of bits.

- Parity and stop bits are checked in their respective states.

- If any issue arises, the FSM moves to the ERROR state.

This FSM effectively manages the reception of data via UART, handling start bits, data bits, parity bits, and stop bits while ensuring proper error detection.

3.2.5. TX Finite State Machine

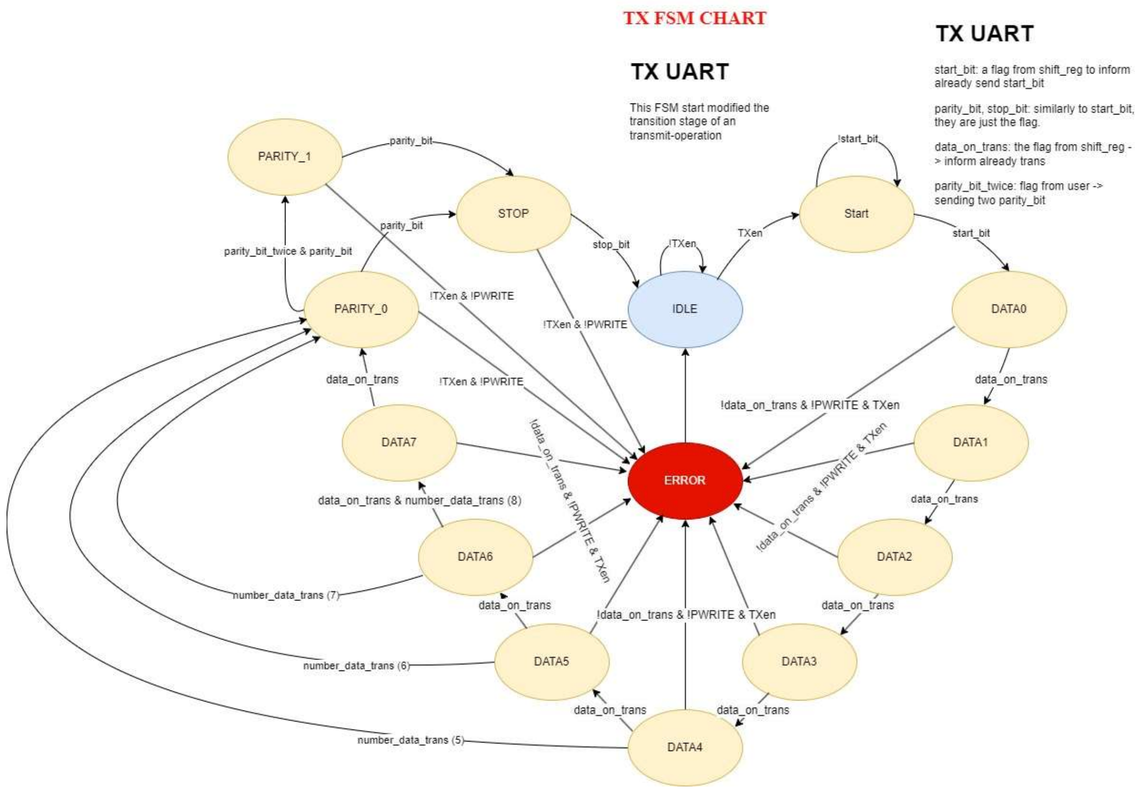

TX FSM CHART
TX UART
This FSM start modified the transition stage of an transmit-operation
TX UART
start_bit: a flag from shift_reg to inform already send start_bit
parity_bit, stop_bit: similarly to start_bit, they are just the flag.
data_on_trans: the flag from shift_reg -> inform already trans
parity_bit_twice: flag from user -> sending two parity_bit
PARITY_1
STOP
IDLE
Start
DATA0
PARITY_0
DATA7
ERROR
DATA1
DATA6
DATA2
DATA5
DATA3
DATA4
parity_bit
parity_bit
parity_bit_twice & parity_bit
stop_bit
!TXen
TXen
!start_bit
start_bit
!TXen & !PWRITE
!TXen & !PWRITE
!TXen & !PWRITE
data_on_trans
data_on_trans
!data_on_trans & !PWRITE & TXen
!data_on_trans & !PWRITE & TXen
!data_on_trans & !PWRITE & TXen
!data_on_trans & !PWRITE & TXen
data_on_trans & number_data_trans (8)
data_on_trans
number_data_trans (7)
data_on_trans
data_on_trans
number_data_trans (6)
data_on_trans
data_on_trans
number_data_trans (5)

Figure 3.2.5: APB UART TX Finite State Machine

The TX FSM (Finite State Machine) for UART transmission operates similarly to the RX FSM, but it manages the transmission of data rather than the reception. Let's go through the steps for a typical TX FSM based on what is usually expected in UART transmission:

Key States of the TX FSM:

- **IDLE**: The FSM remains in this state when there is no data to transmit.

Transition:

+ When data is ready to be sent ("**TXen**" is asserted), the FSM transitions to the **START** state to initiate transmission.

- **START**: The FSM begins transmission by sending a start bit (which is typically a low signal) to indicate the start of data transmission.

Transition:

+ After the start bit is transmitted, the FSM moves to DATA0 state to start transmitting data.

- **DATAx**: These states handle the transmission of the actual data bits, typically one bit per clock cycle, starting with the least significant bit (LSB).

Transition:

+ After each data bit is transmitted, the FSM continues sending until all data bits are transmitted. It then transitions to the **PARITY** state if parity is enabled or directly to the **STOP_0** state if parity is disabled.

- **PARITY** (if parity is enabled): The FSM sends the parity bit to ensure data integrity by checking for odd or even parity.

Transition:

+ After transmitting the parity bit, the FSM transitions to the **STOP_0** state to send the stop bits.

- **STOP_0** and **STOP_1**: These states transmit the stop bit(s) (typically one or two high-level bits) to signal the end of data transmission.

Transition:

+ Once the stop bit(s) are transmitted, the FSM returns to the **IDLE** state, ready for the next transmission.

- **ERROR** (Optional): This state is entered if there is a failure in the transmission process, such as incorrect signaling or data error during transmission.

Transition:

+ The FSM can return to the **IDLE** state after the error is handled or reset.

Summary:

The TX FSM operates by starting with the **IDLE** state, initiating transmission upon data readiness, and following the sequence: **START** → **DATAx** → (optional) **PARITY** → **STOP_0 (1)** → **IDLE**. If any error occurs during the process, the FSM transitions to the ERROR state for error handling.

This FSM ensures smooth UART data transmission by managing the correct sequencing of bits and signaling in accordance with the UART protocol.

3.2.6. TX Shift Register

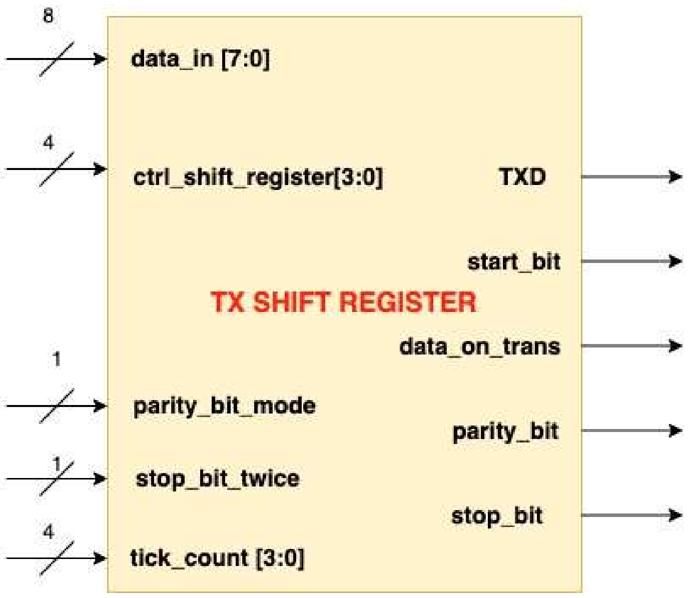


Figure 3.2.6: APB UART TX Shift Register Block

3.2.6.1. TX Shift Register Block Explanation

The TX Shift Register responsibles for serializing parallel data and sending it out over a serial interface (TXD). In this design, the block has 3 inputs and 5 outputs, performing the key task of converting the 8-bit parallel data into a serial data stream with appropriate control signals.

Inputs:

- **data_in[7:0]** (8 bits length): This is the parallel data input to the TX Shift Register. It contains the 8-bit data that needs to be serialized and transmitted over the TXD line. The data is typically loaded into the shift register at the beginning of a transmission cycle.

- **ctrl_shift_register** (4 bits): This control input manages the behavior of the shift register. Each bit of the control signal can have a specific function, such as enabling the shift operation, controlling the transmission of start/stop bits, or managing parity. The control signal ensures that the data is shifted out in the correct sequence and format.

- **tick_count** (4 bits): The tick count acts as a timing signal, controlling when the data is shifted out from the register. For example, if "**tickcount** = 1", the shift register outputs the corresponding bit of the data at position 1 (“**data_out**[1]”). It helps in synchronizing the bit-wise transmission of the data.

- **parity_bit_mode** and **stop_bit_twice**: These input signals are used as a specializing **ENABLE** to control the TX Shift Register block to send out data for **parity_bit** & second **stop_bit** flag or not.

Outputs:

- **start_bit**: This flag indicates the beginning of the transmission. After sending the data bits, a start_bit flag is sent to the TX FSM of APB UART inform that start bit is being transmitted. The start_bit is typically at ‘0’ (LOW).

- **data_on_trans** (Data on Transmission): This output flag is the alarm data from the 8-bit input, sent out one bit at a time based on the tick count. Each bit of the data is sequentially transmitted during the transmission cycle, starting from the least significant bit (LSB).

- **parity_bit**: The parity_bit flag is generated based on the control settings (if parity is enabled). It helps in error detection by ensuring that the number of 1’s in the transmitted data (including the parity bit) is either even or odd, depending on the parity type.

- **stop_bit**: This flag indicates the end of the transmission. After all data bits and the parity bit (if enabled) are transmitted, a stop bit (typically set to 1) is sent to indicate the completion of the data transmission. The stop bit flag informs the system that the transmission is complete.

- **TXD** (Transmitted Data): This is the serial output of the shift register. The TXD signal is the actual data line through which the serialized data (start bit, data bits, parity bit, stop bit) is sent. It functions as the primary serial port, converting the parallel data into a single line serial output, which can be received by another device.

3.2.6.2. Operation:

- When a transmission starts (from the TX FSM), the start bit is transmitted first to notify the receiver and send out the start_bit flag for letting the TX FSM knows whether to continue.

- Then, the tick_count controls when each data bit is shifted out through the data_on_trans signal. The 8-bit data is sent one bit at a time, starting from the LSB.

- After the data bits, if parity is enabled, the parity_bit is generated and transmitted.

- Finally, the stop bit is sent, and the stop_bit Flag is raised to signal the completion of transmission.

- The TXD signal serves as the serial output, carrying all the bits (start, data, parity, stop) in sequence for transmission.

This block efficiently serializes parallel data, managing the transmission format and ensuring that all necessary bits (start, data, parity, stop) are correctly transmitted in order.

3.2.7. RX Shift Register

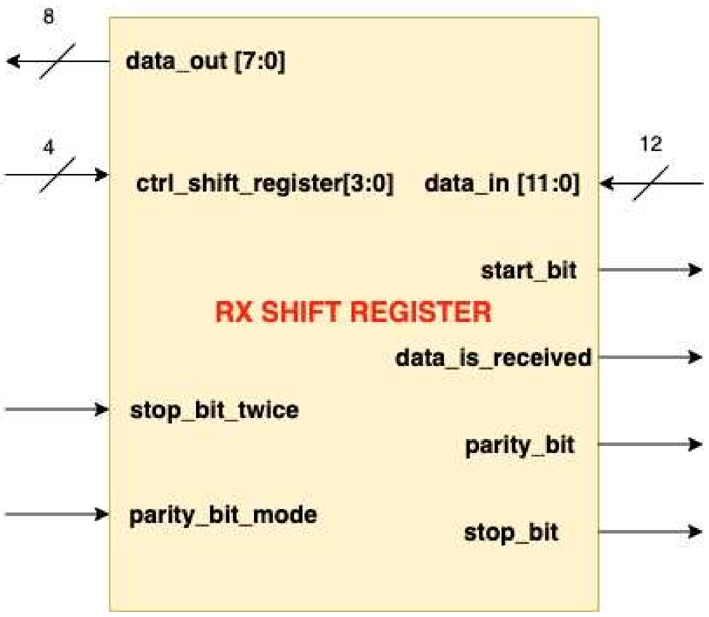


Figure 3.2.7: APB UART RX Shift Register Block

3.2.7.1. RX Shift Register Block Explanation

The RX Shift Register is responsible for receiving and deserializing serial data from the UART, converting it into parallel data, and validating the integrity of the received data. In this design, the block has 4 inputs and 5 outputs, playing a crucial role in managing the receipt of serial data, checking for the correct start, stop, and parity bits, and outputting the received data in parallel form.

Inputs:

- **data_in[11:0]** (12 bits): This input provides the received parallel data, which is already packaged by UART Start bit Detect block, which includes not just the 8-bit data being received but also the start bit, stop bits, and parity bit (if has enabled). The 12-bit width allows the block to handle these additional bits as part of the overall received data packet.

- **ctrl_shift_register** (4 bits): The control input manages the operational mode of the shift register and controls the output flags, including the start_bit, data_is_received, stop_bit, parity_bit. Each bit of the control signal could enable or disable specific features, ensuring proper processing of the received data.

- **stop_bit_twice** (User Setting): This input determines whether the RX Shift Register expects one or two stop bits at the end of the transmission. In some UART configurations, two stop bits are sent instead of one to ensure reliable communication. This input configures the RX shift register to check for the appropriate number of stop bits.

- **parity_bit_mode** (User Setting): This input selects whether parity checking is enabled or disabled, and if enabled, the type of parity (odd or even) to be used. It configures the shift register to handle the parity bit accordingly, either checking the received parity or ignoring it based on the mode.

Outputs:

- **start_bit**: Same as TX Shift Register, this flag indicates the beginning of the receiving. After receiving the data bits (LOW), a **start_bit** flag is sent to the RX FSM of APB UART inform that start bit is being received. The start_bit is typically at '0' (LOW).

- **data_is_received**: This output flag signals that the full 8-bit data has been successfully received and is now ready to be processed or passed on to subsequent blocks. It indicates that the transmission is complete and that the data has been correctly captured by the RX shift register.

- **parity_bit**: The parity bit is received along with the data and is checked based on the mode setting (enabled/disabled). This output indicates whether the received parity bit matches the expected parity (odd or even). It is used for error detection to ensure that no data corruption occurred during transmission.

- **stop_bit**: This flag indicates that the stop bit (or stop bits) has been successfully received, signaling the end of the transmission. Depending on the stop_bit_twice setting, it will check for one or two stop bits. The flag is raised to confirm that the transmission has been properly terminated.

- **data_out** (8 bits): This is the parallel output of the received 8-bit data. After the complete data packet has been received (including the start, data, parity, and stop bits), the 8-bit data is made available on this output for further processing by other blocks or for storage in memory. The data_out signal represents the core data content of the received transmission.

3.2.7.2. Operation:

- When the RX shift register detects a start bit, it knows that a new byte of data is being received and begins shifting in the received data.

- The data_in input is processed as 12 bits parallel receiver, meaning that it will received all 12 bits from RX FIFO and store inside for next using.

- The ctrl_shift_register manages the checking of start, stop, and parity bits.

- The stop_bit_twice input determines how many stop bits are expected at the end of the transmission.

- The parity_bit_mode configures the block to either check the parity or ignore it based on the transmission protocol.

- Once all 8 data bits are received, the data_is_received flag is raised, indicating that the full byte has been captured.

- If all checks (start bit, stop bit(s), parity bit) are successful, the data_out output will present the parallel form of the 8-bit received data for further use.

This block efficiently deserializes UART data, ensuring that the transmission protocol (start, stop, parity) is respected and that errors are flagged where necessary, providing the 8-bit data in parallel form for easy integration into the system.

3.2.8. UART start bit detector

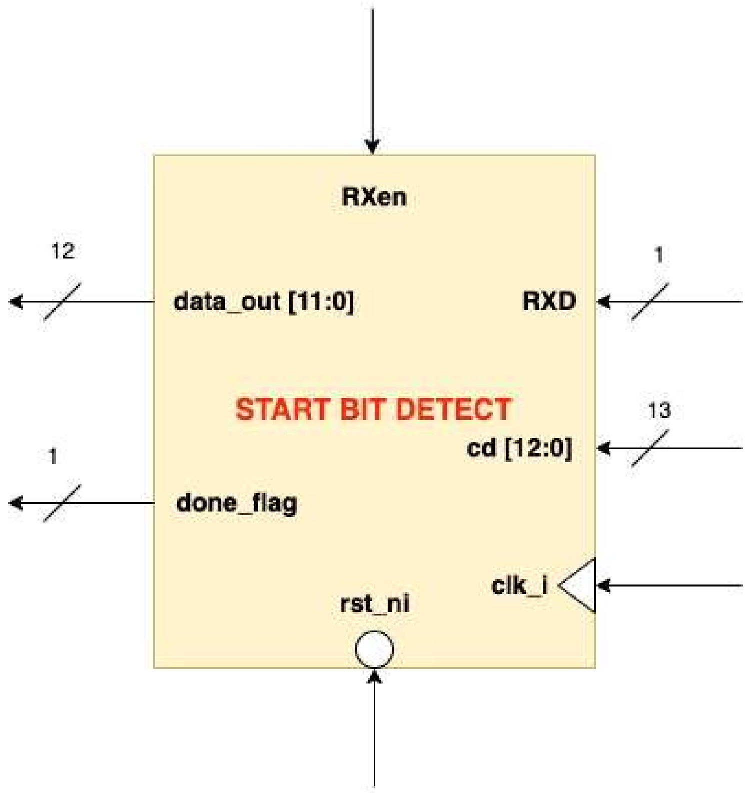


Figure 3.2.8: APB UART Start Bit Detect Block

3.2.8.1. Start Bit Detect Block Explanation

The Start Bit Detect block is responsible for detecting the beginning of a UART transmission by identifying the start bit in the incoming serial data stream. The start bit is a crucial part of UART communication, marking the transition from an idle line (typically high) to an active transmission. This block processes the input serial data, enables the reception process, and outputs the received 12-bit data along with a flag indicating that the data is ready to be passed to the RX FIFO for storage.

Inputs:

- **RXD** (Serial Input): This input receives the serial data from the UART line. It is monitored continuously to detect the falling edge of the start bit, which indicates the beginning of a new data transmission. RXD carries the serial bits including the start bit, data bits, parity bit (if applicable), and stop bits.

- **cd** (Counter Divisor, 13 bits): The counter divisor is used to set the timing for detecting transitions on the RXD line. Since UART operates asynchronously, the system needs to sample the RXD line at a specific rate to correctly detect when the start bit begins. The cd input provides a 13-bit value that controls the sampling rate and helps detect the falling edge of the RXD signal, indicating the start of the transmission.

- **RXen** (Receive Enable): This input acts as an enable signal for the reception process. When RXen is asserted, the block is allowed to detect the start bit and process the incoming data. If RXen is low, the block will not engage in data reception, effectively disabling the operation until enabled.

Outputs:

- **done_flag**: The done_flag signals to the RX FIFO that the full 12-bit data (which includes start, data, parity, and stop bits) has been received and is ready for transmission to the RX FIFO. Once the block detects the start bit and completes the reception of all 12 bits, the done_flag is asserted. This output is used to synchronize the RX FIFO with the received data, ensuring proper data handling and storage.

- **data_out[11:0]**: This 12-bit output carries the entire received data packet, which includes:

+ Start bit

+ 8 data bits (the core transmitted information)

+ Parity bit (if enabled)

+ Stop bit(s)

The **data_out[11:0]** provides the deserialized data in parallel format, making it ready for further processing or storage in the RX FIFO. It contains all the information needed to validate and reconstruct the original transmitted data.

3.2.8.2. Operation:

- **Start Bit Detection**: The block continuously monitors the RXD input for a falling edge, which represents the start bit. UART lines typically remain idle at a high logic level (1), and the start bit is a transition from high to low (falling edge), signaling the beginning of a new transmission.

- **Sampling Timing**: The cd (counter divisor) input determines the sampling rate for monitoring the RXD line. To accurately detect the start bit, the system must sample at a frequency appropriate for the baud rate of the UART communication. The cd value helps synchronize the sampling to the timing of the incoming data.

- **Data Reception**: Once the start bit is detected, the block continues to receive the remaining bits (8 data bits, parity bit, stop bit). It shifts in each bit from the serial RXD input and stores the full 12-bit data in the data_out register.

- **Completion Flag**: After the full 12 bits have been received, the done_flag is asserted, indicating that the data is ready to be passed to the RX FIFO. This flag ensures that the received data is properly transferred without overlap or loss.

- **Data Output**: The parallelized 12-bit data is made available on the data_out[11:0] output, providing the complete data packet received from the serial UART transmission. This output can then be passed to the RX FIFO for storage and further use.

The Start Bit Detect block plays a critical role in synchronizing the reception process with the incoming UART data stream, accurately detecting the start of a transmission, and ensuring that the received data is correctly captured and flagged for further processing.

4. RTL IMPLEMENTATION RESULT AND TESTING METHOD

4.1. AHB – APB Bridge

4.1.1. RTL Implementation result via Quartus II 64-Bit

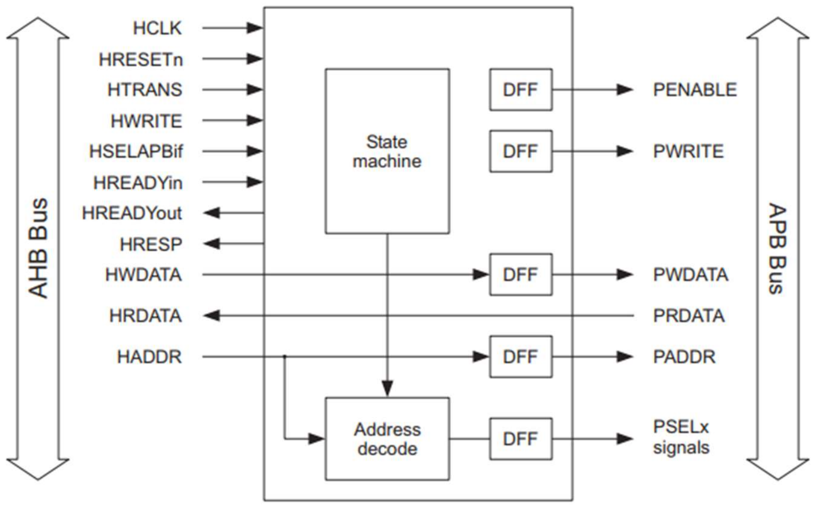


Figure 4.1.1.1: AHB – APB Bridge Block Diagram for reference

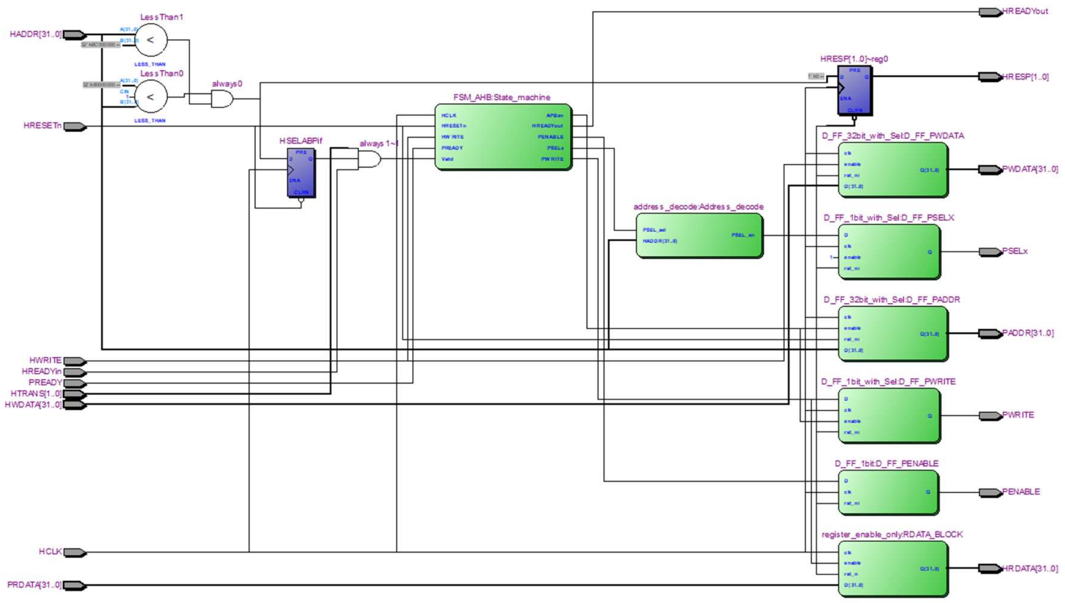


Figure 4.1.1.2: AHB – APB Bridge Netlist Viewer Implementation in general

4.1.2. Testing Methodology

The waveform displayed in the image shows the behavior of various signals during the execution of your testbench for the AHB to APB bridge. I'll break down the waveform step by step and explain how it matches the given testbench code.

- Clock Signal (HCLK):

+ The clock signal oscillates continuously with a 50% duty cycle, as defined by the "initial" block in the testbench (HCLK = 0; forever #10 HCLK = ~HCLK;). This creates a clock period of 20 ns.

- Reset Signal (HRESETn):

+ The reset is active low (HRESETn = 0), as shown by the waveform in the initial phase, and then de-asserts to high (HRESETn = 1) after 40 ns. This ensures the module is reset at the start and then begins normal operation.

- AHB Transaction Phases:

+ The AHB master initiates transactions with the signals **HADDR**, **HTRANS**, **HWRITE**, **HWDATA**, and **HREADYin**.

+ **HTRANS**: Controls the type of transfer.

+) At the start, it is 2b00 (**IDLE**), then switches to 2b10 (**NONSEQ**) to indicate the start of a write transaction, followed by 2b11 (**SEQUENTIAL**), and back to 2b00 (**IDLE**).

+ **HADDR**: Holds the address for the transaction.

+) Initially, it's 0x00000000 and then changes to 0x80000000 for the write transaction, followed by 0x80000004 for the read transaction.

+ **HWRITE**: Determines whether the transaction is a read or write.

+) Initially, it's 0 (read), then changes to 1 (write) when the **HTRANS** signal indicates a valid transaction (**NONSEQ**).

+ **HREADYin**: Indicates when the bus is ready for the next transaction. It's 1 when the transaction is ready to proceed, and 0 when the bus is not ready.

- Write Transaction to Address 0x80000000:

+ The first valid transaction starts after 100 ns.

+ **HADDR** is set to 0x80000000, indicating the write operation to that address.

+ **HWRITE** is 1, meaning it's a write transaction.

+ **HTRANS** is set to 10 (**NONSEQ**), indicating the start of a new transaction.

+ **HWDATA** is set to 0x12345678, the data to be written.

+ **HREADYin** is 1, indicating that the master is ready to initiate the transaction.

+ After this, the transaction is completed, and the bus moves back to the **IDLE** state.

- **Read Transaction from Address 0x80000004**:

+ After 100 ns, a read transaction starts.

+ **HADDR** is set to 0x80000004, indicating a read operation from that address.

+ **HWRITE** is 0, meaning it's a read transaction.

+ **HTRANS** is 11 (**SEQUENTIAL**), indicating the continuation of the transaction sequence.

+ **PRDATA** is set to 0x87654321 in the testbench to provide data for the read.

+ **HRDATA** will hold this read data after the transaction is completed.

+ The bus eventually goes back to **IDLE** after this read.

- **APB Interface Signals**:

+ **PADDR**: This corresponds to the **HADDR** value when a valid transaction occurs, and it changes accordingly for both the write and read operations.

+ **PWDATA**: This signal takes the value of **HWDATA** during the write transaction (0x12345678).

+ **PWRITE**: Indicates whether the operation is a write (1) or read (0). It's aligned with the HWRITE signal.

+ **PENABLE**: This signal controls the timing of an APB transfer. It is enabled after the address and data are set.

+ **PSELx**: This signal is asserted to select the APB slave during the transaction.

Summary:

- **Initial State**: The system is reset, and all inputs are idle.

- **Write Transaction**: After 100 ns, a write transaction is initiated to 0x80000000 with data 0x12345678.

- **Read Transaction**: After another 100 ns, a read transaction is initiated from 0x80000004, and the read data (0x87654321) is available on the bus.

- **APB signals**: The APB signals (**PADDR**, **PWDATA**, **PWRITE**, **PENABLE**, and **PSELx**) follow the AHB signals and are active during valid transactions.

This testbench and simulation validate that the AHB to APB bridge correctly handles AHB read and write transactions, translates them into APB transfers, and returns appropriate data for read operations.

4.2. APB UART RX - TX

4.2.1. RTL Implementation result via Quartus II 64-Bit

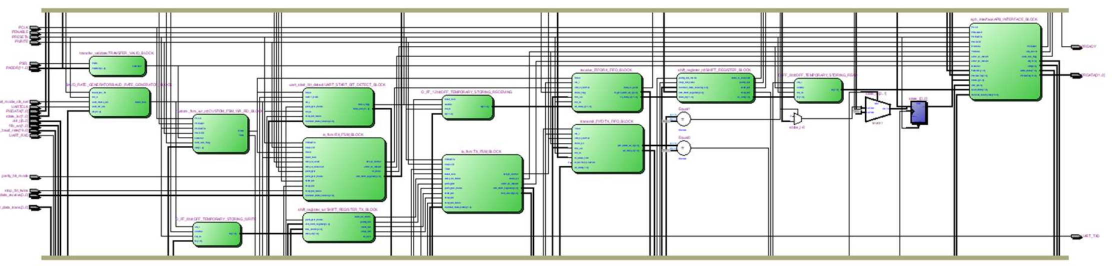


Figure 4.2.1: APB UART Netlist Viewer Implementation in general

4.2.2. Testing Methodology

In this tb_APB_UART testbench, the simulation is designed to verify the functionality of the APB_UART module by interacting with APB bus signals and UART communication. Below is a detailed breakdown of how the test works:

- **Input and Output Signal Declaration**:

+ Input and output signals are declared, corresponding to the required signals of the APB_UART module.

+ Input signals include control signals (**PCLK**, **UARTCLK**, **PRESETn**), UART configuration settings (baud rate, parity, stop bit), and APB bus signals (**PSEL**, **PENABLE**, **PWRITE**, **PADDR**, **PWDATA**).

+ Output signals include **PREADY** and **PRDATA** from the APB bus, and **UART_TXD** for UART transmission.

- **Instantiate the APB_UART Module**:

+ The testbench instantiates an instance of the APB_UART module, with all the input and output signals connected.

- **Clock Generation:**

+ Two clock generation blocks are used for **PCLK** (100 MHz) and **UARTCLK** (50 MHz). These clocks provide the timing signals required for synchronous elements in the module:

+ **PCLK**: The main clock for the APB bus, with a period of 10 ns (100 MHz).

+ **UARTCLK**: The UART clock, with a period of 20 ns (50 MHz).

- **Initial Block for Initialization and Simulation**:

+ Default values are set to initialize the control signals of the module. For example:

+) **PRESETn** is initially set to 0 (reset active).

+) UART configurations like desired_baud_rate is set to 9600, parity_bit_mode, and **stop_bit_twice** are set accordingly.

+) **UART_RXD** is set to 1 (in UART idle state, the **RXD** line is typically high).

+ After 20 ns, the **PRESETn** signal is brought high (1), allowing the module to begin operating.

- **APB Transaction (APB Write Transaction):**

+ Once the module is initialized, the testbench starts an APB write transaction:

+) **PSEL** is set to 1 to select the peripheral (the UART).

+) **PWRITE** is set to 1, indicating a write transaction.

+) **PADDR** is set to 0x400 and **PWDATA** is set to 0xA5 to send data to the module.

+) After 40 ns, **PENABLE** and **PSEL** are de-asserted to end the write transaction.

+ The testbench then waits for the **PREADY** signal to confirm that the module has completed the transaction.

- **APB Read Transaction:**

+ The testbench performs an APB read transaction:

+) **PSEL** is set to 1 to select the peripheral again.

+) **PWRITE** is set to 0, indicating a read transaction.

+) The address **PADDR** remains 0x400 to read data from the same address used for the write.

+) Once the read transaction is complete and **PREADY** is asserted (set to 1), the result from **PRDATA** is displayed on the console.

- **End of simulation**:

+ After printing the read data, the simulation finishes with a call to $finish, 100 ns after the last operation.

Summary:

- This testbench verifies communication via the APB bus with the APB_UART module:

+ First, a write transaction is performed (writing the data 0xA5 into the module).

+ Then, a read transaction is performed to retrieve the same data from the module.

+ The result of the transaction is printed to the console to check the correctness of the operation.

Purpose of the Test:

- To ensure that the APB_UART module can correctly handle read/write operations over the APB bus.

- To check the interaction between control and data signals, ensuring that the module can receive data from PWDATA and send it back via PRDATA.

5. RESULT OF IMPLEMENTATION

5.1. Result of AHB – APB Bridge testbench

Note: This module normally run base on rising edge of CLK input, low active reset and HRESP will always be 2`b01 to indicate the interfacing process is.

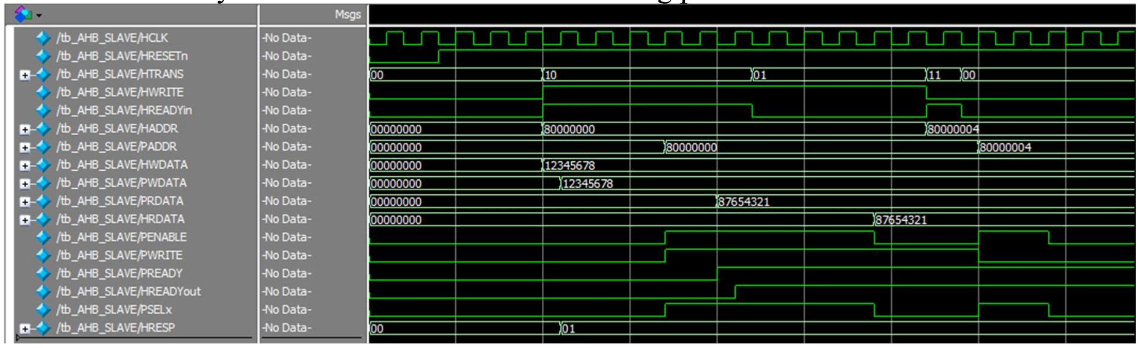


Figure 5.1: AHB – APB Bridge testbench simulation

Explanation:

- Before sending or reading signal come into, AHB – APB Bridge would check the validity of the PSEL base on HTRANS & HADDR first (in the range of 0x80000000 < HADDR < 0x8C0000000), in order to detect the correctness data transfer and whether the UART destination is point to (in the future, if I have changed to continue with this design, there would not be only 1 UART communication bus). Then due to the signal input:

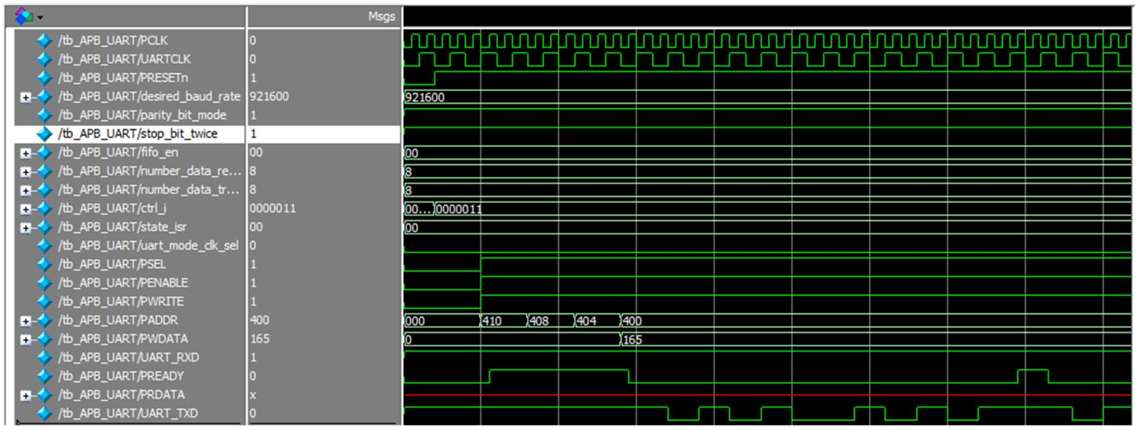
+ When PWRITE = 1, The AHB – APB Bridge will consider this is a write transaction, which lead to enable the PSELx (point to peripheral), PENABLE, PREADYout are HIGH (1). Then based on HWDATA, the PWDATA to APB Peripheral is transferred into.

+ In contrast, When PWRITE = 1, The AHB – APB Bridge will consider this is a read transaction. Then based on PRDATA, the HRDATA would be received the data and ready for transmit out to the AHB Master. On the other hand, the other signal such as PSELx (point to peripheral), PENABLE, PREADYout are HIGH (1) as similar to write transaction.

5.2. Result of APB UART testbench

In this testbench for simulation, I have just used the basic function for receiving and transmitting data (fifo_en = 2`b00, uart_mode_sel_clk = 0, ctrl = 2`b11). Which lead to the clock signal currently running is at 100MHz, no FIFO is using for storage, available for both transmit and receiver (ctrl = 2`b11).

5.2.1. Write Transaction



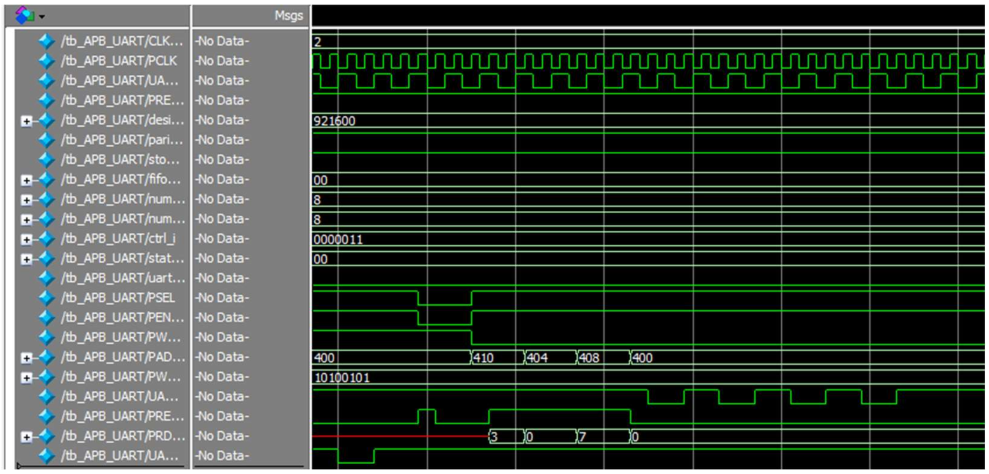
	Msgs
/tb_APB_UART/PCLK	0
/tb_APB_UART/UARTCLK	0
/tb_APB_UART/PRESETn	1
/tb_APB_UART/desired_baud_rate	921600
/tb_APB_UART/parity_bit_mode	1
/tb_APB_UART/stop_bit_twice	1
/tb_APB_UART/fifo_en	00
/tb_APB_UART/number_data_re...	8
/tb_APB_UART/number_data_tr...	8
/tb_APB_UART/ctrl_i	0000011
/tb_APB_UART/state_isr	00
/tb_APB_UART/uart_mode_clk_sel	0
/tb_APB_UART/PSEL	1
/tb_APB_UART/PENABLE	1
/tb_APB_UART/PWRITE	1
/tb_APB_UART/PADDR	400
/tb_APB_UART/PWDATA	165
/tb_APB_UART/UART_RXD	1
/tb_APB_UART/PREADY	0
/tb_APB_UART/PRDATA	x
/tb_APB_UART/UART_TXD	0

Figure 5.2.1: APB TX UART testbench simulation

Explanation:

- In this part of testbench simulation, firstly, I transmitted data of user custom such as cd (counter divisor) = 3 related to Baud rate input (921600) references is in the table 3.2.1; ctrl(2`b11) to enable for both transmit and receive; state interrupt via APB interface by PADDR at 0x410, 0x408, 0x404 respectively. Then by letting the PADDR to 0x400, this will enable the APB UART process to start to transmit data via serial signal TXD. So, as I transferred PWDATA (165 in decimal) its Binary's version will be 10100101 and LSB is transmitted first. Then, after finished the transfer process, PREADY is set to HIGH and transfer to AHB – APB Bridge, will alarm that APB UART is ready for the next data transaction.

5.2.2. Read Transaction



	Msgs
/tb_APB_UART/CLK...	-No Data-
/tb_APB_UART/PCLK	-No Data-
/tb_APB_UART/UA...	-No Data-
/tb_APB_UART/PRE...	-No Data-
/tb_APB_UART/desi...	-No Data-
/tb_APB_UART/pari...	-No Data-
/tb_APB_UART/sto...	-No Data-
/tb_APB_UART/fifo...	-No Data-
/tb_APB_UART/num...	-No Data-
/tb_APB_UART/num...	-No Data-
/tb_APB_UART/ctrl_i	-No Data-
/tb_APB_UART/stat...	-No Data-
/tb_APB_UART/uart...	-No Data-
/tb_APB_UART/PSEL	-No Data-
/tb_APB_UART/PEN...	-No Data-
/tb_APB_UART/PW...	-No Data-
/tb_APB_UART/PAD...	-No Data-
/tb_APB_UART/PW...	-No Data-
/tb_APB_UART/UA...	-No Data-
/tb_APB_UART/PRE...	-No Data-
/tb_APB_UART/PRD...	-No Data-
/tb_APB_UART/UA...	-No Data-

Figure 5.2.2: APB UART RX testbench simulation

Explanation:

- Similar as write transaction, in this part of testbench simulation, firstly, I tried to receive data of as I have configured before so the result of PRDATA would be cd (counter divisor) = 3 related to Baud rate input (921600) references is in the table 3.2.1; ctrl = 7 (7`b111) which the ctrl[2] is the flag for error during transmit (not finished transfer cycle); state = 0 because I did not using interrupt via APB interface and so on with PADDR at 0x410, 0x408, 0x404 respectively. Then by letting the PADDR to 0x400, this will enable the APB UART process to start to receive data via serial signal RXD. So, as I have simulated that RXD is reading the same data as I have transferred 165 in decimal. However, I have not finished with this part because of arising problem in testbench. As its Binary's version supposed to be `0 for start 8`10100101 for data, 1 for parity and 2`b11 for stop bit and LSB is received first. Then if the RX process done, we will see the HRDATA would be 165 as I have said above.

6. CONCLUSION AND DEVELOPMENT PATH

6.1. Conclusion

The implementation of the AHB-APB bridge and APB-UART interface in this project successfully demonstrates the capability to manage communication between high-speed and low-speed components within a System on Chip (SoC). The AHB-APB bridge efficiently connects the high-performance AHB bus to the peripheral-focused APB bus, while the APB-UART interface ensures seamless UART communication, a critical feature in many embedded systems.

By leveraging Finite State Machines (FSM) for both the AHB-APB bridge and UART communication, the project has ensured proper control of data flow, managing state transitions in both read and write operations. The integration of key components such as the baud rate generator and FIFO buffers for UART transmission and reception further strengthens the reliability of the system, providing flexibility in handling data with different rates.

Overall, this project has not validated via kit DE-10 standard yet due to the custom IP process design, the core design and communication between buses but also laid a foundation for more advanced developments in SoC designs that require seamless integration of high-performance and peripheral components.

6.2. Development Path

Moving forward, several development paths can be explored to enhance and expand the functionality of this project:

- **Enhanced Error Handling**: Implement more sophisticated error detection and correction mechanisms for both the AHB-APB bridge and UART interfaces. This includes handling scenarios like bus contention, data corruption, and protocol violations more robustly.

- **Multi-Channel UART Support**: Extending the APB-UART interface to support multiple UART channels simultaneously would increase the design's applicability in systems with numerous communication requirements.

- **Higher Baud Rate Support**: Optimize the baud rate generator and UART interface to support higher communication speeds without sacrificing reliability, particularly for applications requiring high-speed serial communication.

- **Interrupt and DMA Support**: Integrate interrupt and DMA (Direct Memory Access) support in the UART module to offload the CPU and enable more efficient data handling, especially in high-performance systems where multiple processes occur simultaneously.

- **Test and Verification**: Expand the testbench and verification environment to include corner cases, stress testing, and formal verification techniques to ensure the design is robust and reliable under various conditions.

- **Integration into Complex SoCs**: Integrating this bridge and UART interface design into larger and more complex SoC designs, potentially with multiple AHB and APB bridges, would prove its scalability and adaptability in real-world applications.

These development paths would not only improve the current implementation but also extend its applicability to a broader range of applications, making the system more reliable, efficient, and versatile for various embedded system designs.

7. REFERENCES

1. ***AMBA AHB Protocol Specification***
https://developer.arm.com/documentation/ihi0033/latest/

2. ***AHB Example AMBA System Technical Reference Manual***
https://developer.arm.com/documentation/ddi0170/a/I967114

3. ***DirectCore Advanced Microcontroller Bus Architecture - Bus Functional Model***
https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/UserGuides/CoreAMBA_BFM_UG.pdf

4. ***Effective Design and Implementation of AMBA AHB Bus Protocol using Verilog***
https://www.researchgate.net/publication/337510558_Effective_Design_and_Implementation_of_AMBA_AHB_Bus_Protocol_using_Verilog

5. ***AMBA AHB Protocol*** https://fr.scribd.com/doc/41197279/AMBA-AHB-Protocol-Presentation

6. ***PrimeCell UART (PL011) Technical Reference Manual***
https://developer.arm.com/documentation/ddi0183/g

7. ***Design and FPGA Implementation of UART Using Microprogrammed Controller***
https://www.researchgate.net/publication/282030059_Design_and_FPGA_Implementation_of_UART_Using_Microprogrammed_Controller