Course Project

A custom APB UART IP

You are required to design a custom UART IP that is wrapped with the AMBA APB interface that can be a part of an SoC.

Objective

The goal of this project is to design and understand a **Universal Asynchronous Receiver Transmitter (UART) peripheral wrapped with an AMBA APB interface**. Students will implement the APB slave logic that communicates with a UART core, providing register-based access for control, status, data, and baud rate configuration.

This project helps students learn:

- How to build a memory-mapped peripheral with APB.
- How to map registers to control UART hardware.
- How to handle write/read operations, status updates, and synchronization.
- How to integrate and test UART communication using a bus interface.

1. UART

UART stands for Universal Asynchronous Receiver/Transmitter.

It is a simple serial communication method where data is sent one bit at a time over a single wire in one direction, plus another wire for the reverse direction.

It's point-to-point, meaning one transmitter talks directly to one receiver. Many microcontrollers, FPGAs, and PCs have built-in UART hardware blocks because it's so widely used.

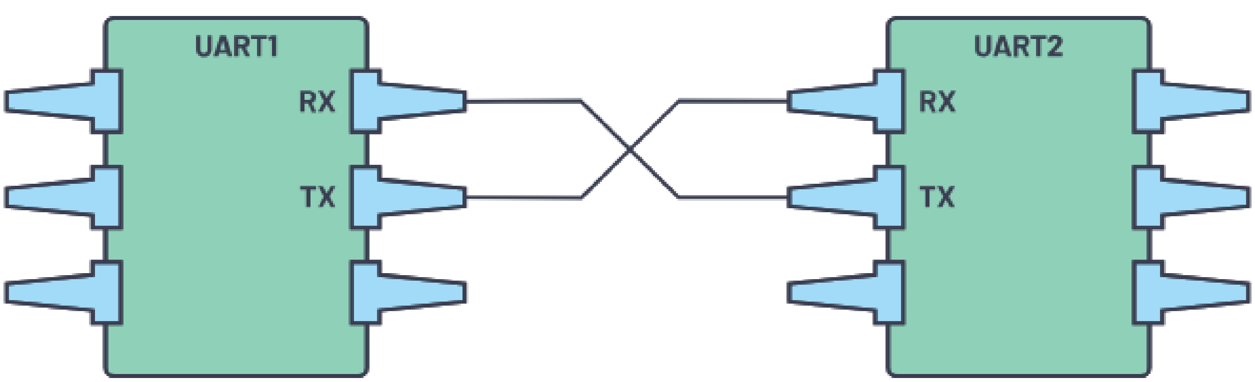

figure 1 UART chips communicating [source [ADI]]

UART uses 2 physical connections, one to transmit data & another to receive data.

Communication happens in **frames**. Each frame has a very specific structure, so the receiver knows where it starts, where it ends, and what data it contains.

The most common frame format is 8-N-1, which means:

8 data bits (sent least significant bit first)

N → No parity bit

1 stop bit

Before the data bits, there's a start bit (logic 0), and after them there's one or more stop bits (logic 1). The line stays idle (logic 1) between frames.

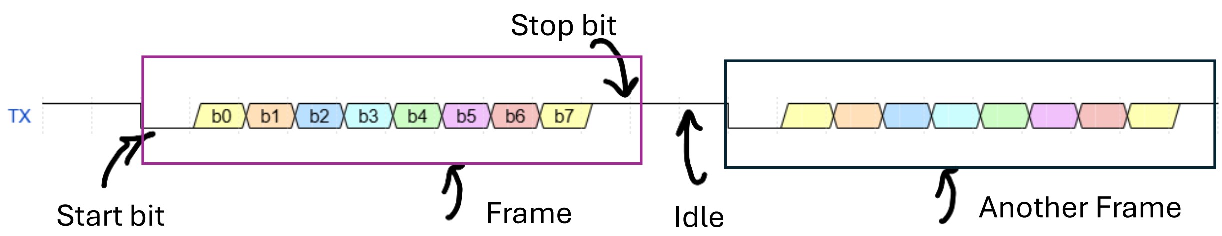

figure 2 UART frame

Before sending or receiving data, receiver and transmitter are configured to use the same **baud rate** (bits per second), **e.g. 9600, 115200, etc.**
At 9600 baud → 1 bit period = 1/9600 ≈ 104.167 μs.

UART architecture

The receiver:

Only receives one byte at a time

- Waits for a falling edge (**start bit**).
- Samples the data bits **in the middle of each bit period**.
- Validates stop bit(s)
- Asserts ***busy*** when a byte is being received
- Asserts ***done*** when a byte is received

This is done through this suggested architecture

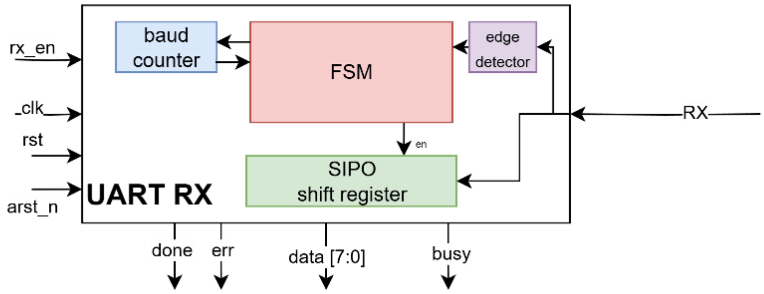

Figure 3 UART RX Architecture

Edge detector:

Catches the line transition from idle to low due to the start bit

Baud counter:

Baud counter is a down counting counter where the FSM controls its load value and gets a flag when it is zero

SIPO shift register

Enabled by the FSM, it samples the RX line every time it's enabled

RX FSM

- Loads the baud counter & waits until it reaches zero
- Control the SIPO shift register enable signal thus sampling bits in the middle
- Keeps track of how many bits have been sampled by the SIPO

Example State diagram

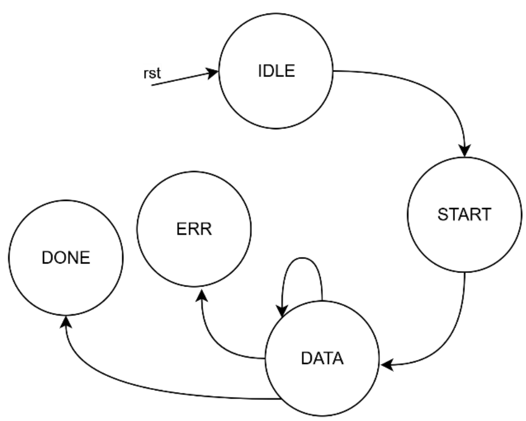

figure 4 RX FSM

- After starting condition, the FSM should wait for 1.5-bit time to enable the SIPO to sample the starting bit
- Then it should wait for 1 bit time
- The FSM should track the number of bits sampled & check for a correct stop bit
- It should assert **err** if framing error is detected

The Transmitter

- Sends only one byte at a time
- Appends start and stop bits
- Handles sending data with right bit times based on baud rate
- Asserts **done** after finishing sending the loaded byte
- Asserts **busy** while sending the loaded byte

Suggested Architecture

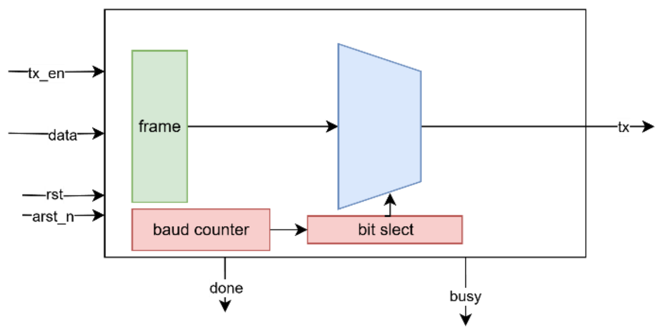

figure 5 UART TX

- **Baud counter** generates bit time based on required baud rate (9600) & system clock (100 MHz)
- **Bit select** keeps track of sent bits and holds the chosen bit on the line for the required bit time

UART register space

All registers are 32 bits

Address	Name	Description
0x0000	CTRL_REG	Contains bits for tx_en , rx_en , tx_rst & rx_rst
0x0001	STATS_REG	Contains bits for rx_busy , tx_busy , rx_done, tx_done & rx_error
0x0002	TX_DATA	UART Tx data
0x0003	RX_DATA	UART Rx data
0x0004	BAUDIV	Variable baud rate for UART [BONUS]

Operation

1. Before every transmission and after every reception of data assert the soft reset bit [tx_rst & rx_rst]
   For UART TX
2. For UART TX write to the TX_DATA register the data to be sent
3. Assert the tx_en bit in CTRL_REG
4. Read tx_done & tx_busy to check data sending
   For UART RX
5. Assert rx_en bit in CTRL_REG
6. Read rx_done , rx_busy , rx_error

2. AMBA APB interface

Why We Need AMBA Buses?

AMBA (**Advanced Microcontroller Bus Architecture**) was introduced by ARM to solve some key integration problems in SoCs:

1. **Standardization** – Before AMBA, each SoC project often had custom bus protocols, making IP reuse difficult. AMBA provides a common, well-defined interface for IP blocks from different vendors.
2. **Simplified IP Integration** – With a standard bus, you can plug CPU cores, peripherals, and memory controllers together without redesigning the communication protocol each time.
3. **Scalability** – Supports a wide range of devices: from high-speed CPU-to-memory transfers to low-speed control registers.
4. **Performance Optimization** – Different AMBA bus types are optimized for different needs (high-throughput, low-latency, low-power).
5. **Lower Development Time & Cost** – Encourages IP reuse, verification reuse, and faster SoC assembly.
6. **Compatibility with ARM Ecosystem** – Most ARM IP blocks are AMBA-compliant, making it the industry default.

Bus	Purpose	Typical Use
AHB (Advanced High-performance Bus)	High-speed, pipelined, single-clock bus for high-bandwidth components	CPU ↔ Memory, DMA ↔ Memory
APB (Advanced Peripheral Bus)	Simple, low-power, non-pipelined bus for low-speed peripherals	Timers, UART, GPIO
AXI (Advanced eXtensible Interface)	High-performance, high-throughput, multiple outstanding transactions, burst support	CPU ↔ High-speed memory, GPU, interconnect fabrics
ACE (AXI Coherency Extensions)	Adds cache coherency support to AXI	Multi-core CPUs with shared caches
CHI (Coherent Hub Interface)	Very high-performance, fully coherent interconnect for large SoCs	Modern multi-core ARM processors
ASB (Advanced System Bus) (legacy)	Older AMBA bus, predecessor to AHB	Now mostly obsolete

About the APB protocol

- The APB protocol is a low-cost interface, optimized for minimal power consumption and reduced interface complexity. The APB interface is not pipelined and is a simple, synchronous protocol. Every transfer takes at least two cycles to complete.
- The APB interface is designed for accessing the programmable control registers of peripheral devices. APB peripherals are typically connected to the main memory system using an APB bridge. For example, a bridge from AXI to APB could be used to connect a number of APB peripherals to an AXI memory system.
- APB transfers are initiated by an APB bridge. APB bridges can also be referred to as a Requester. A peripheral interface responds to requests. APB peripherals can also be referred to as a Completer. This specification will use Requester and Completer.

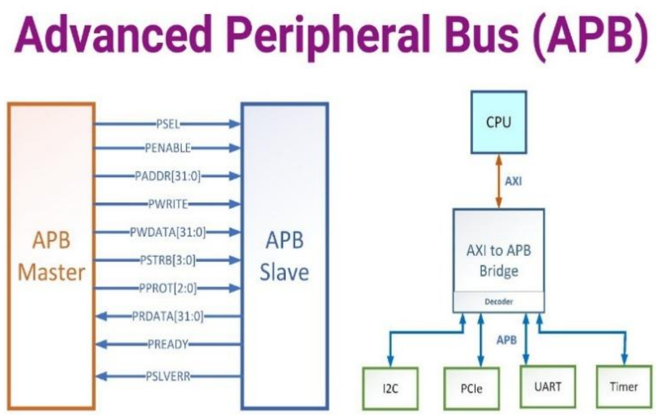

Figure 6: APB Prortocol A

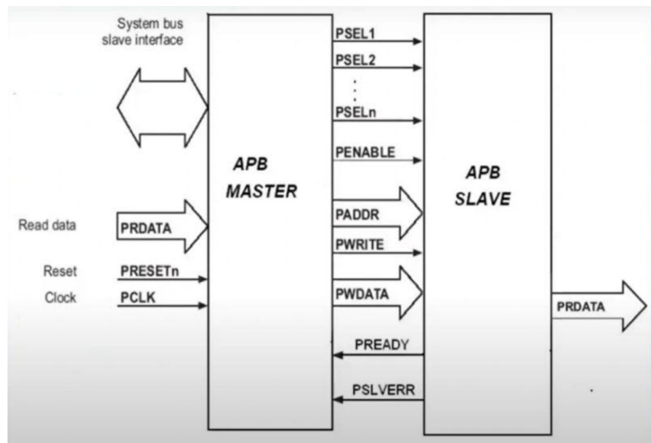

Figure 7: APB Prortocol B

Specifications:

Your design must comply with the AMBA APB (Advanced Peripheral Bus) protocol.
The signals are:

Signal	Direction	Description
PCLK	Input	APB clock
PRESETn	Input	Active-low reset
PADDR	Input [31:0]	Address bus
PSEL	Input	Peripheral select (1 = selected)
PENABLE	Input	Asserted in the ACCESS state from the master to enable the R/W operations
PWRITE	Input	1 = write cycle, 0 = read cycle
PWDATA	Input [31:0]	Write data bus
PRDATA	Output [31:0]	Read data bus
PREADY	Output	Ready signal (1 = transfer complete)

Protocol Requirements

- On **write** (PENABLE =1, PSEL=1, PWRITE =1):
  - Data from **PWDATA** must be written to the addressed register.
  - **PREADY** must go high for one cycle to acknowledge the transaction.
- On **read** (PENABLE =1, PSEL=1, PWRITE =0):
  - Data from the addressed register must appear on **PRDATA**.
  - **PREADY** must go high for one cycle to acknowledge the transaction.
- Between transactions, the FSM must return to **IDLE**.

Top level Architecture

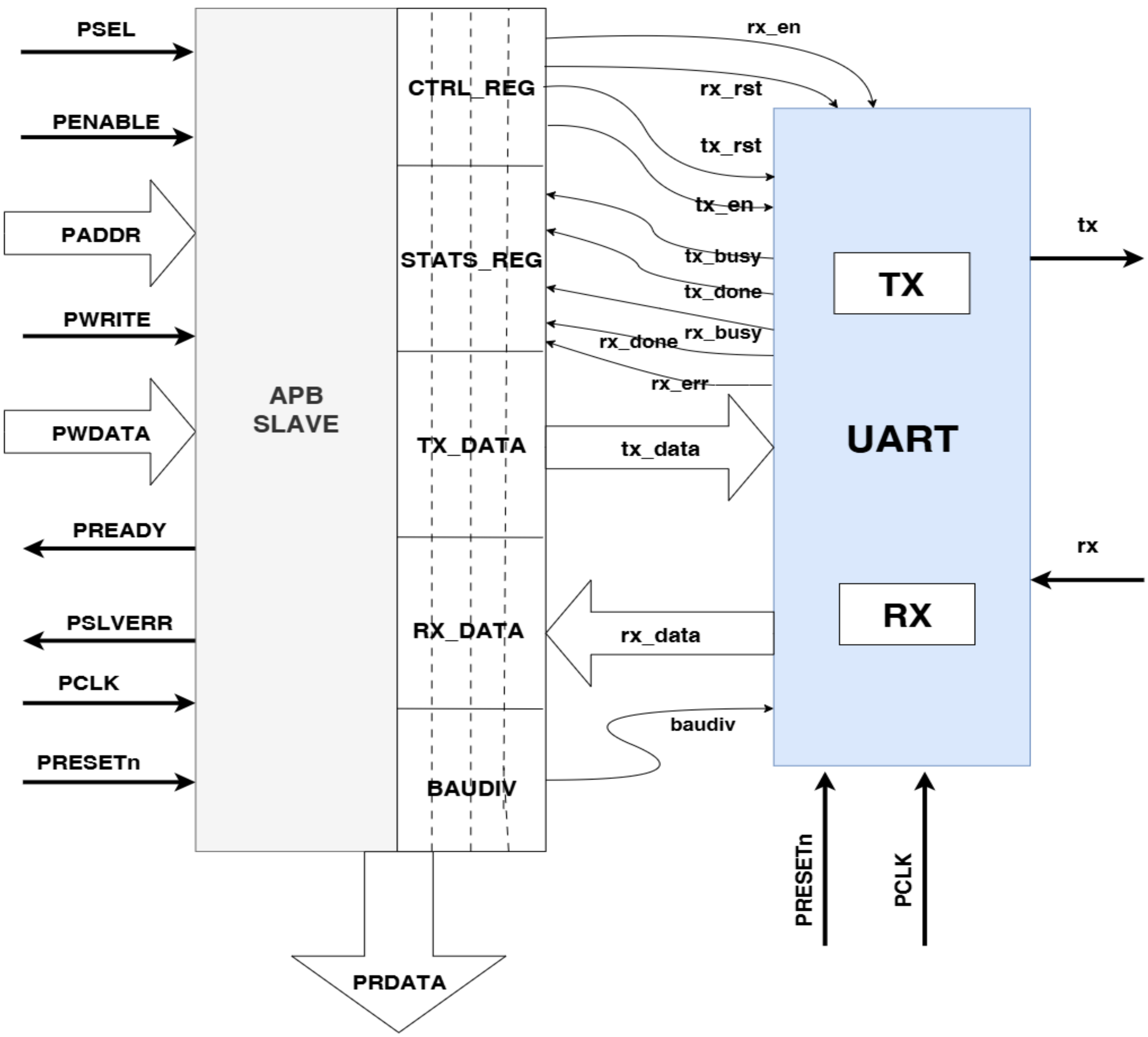

Figure 8: Custom APB UART IP

Deliverables

1. Documentation (PDF Report)

Prepare a **well-structured PDF document** that includes:

1. **Title Page**
2. **Introduction**
3. **Design Analysis**
4. **State Diagrams**
5. **Design Decisions**
6. **Verification Strategy**
7. **Simulation Results**
8. **Conclusion**

2. RTL Designs (Verilog)

Submit clean, **well-commented Verilog RTL code** for the following modules:

1. **UART Transmitter**
2. **UART Receiver**
3. **APB UART Wrapper**

3. Testbenches (Verilog)

For each module, submit a **self-checking test bench** where possible.

1. **UART TX Testbench**
2. **UART RX Testbench**
3. **APB UART Wrapper Testbench**

To be handed in as a GitHub repository with the following folders

./src : for your design files

./fpga : for your FPGA runs, utilization & timing reports

. /dv : for your testbench files and .do files

. /docs : for project documents