Project Title: UART Packetizer with FSM and FIFO Integration
Candidate: Abhinav Kumar /123.shandilyaabhinav@gmail.com
Date: 01/06/2025
Company: Digantara

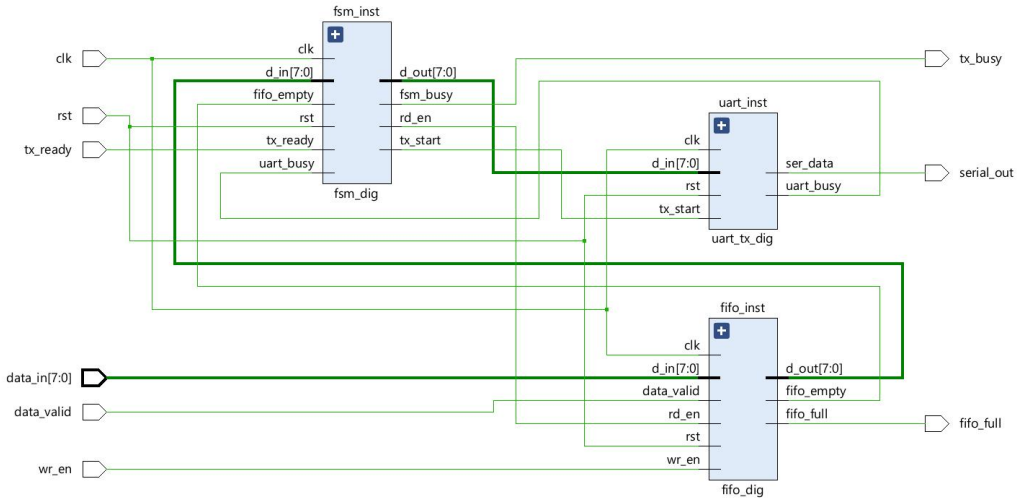
System Overview

Description
The UART Packetizer system accepts 8-bit parallel data, buffers it using a FIFO, and serializes it via a UART-like interface. The system uses a finite state machine (FSM) to control data flow from the FIFO to the UART transmitter.

Assumptions

Parameter	Value
System Clock	50 MHz
Baud Rate	9600
FIFO Depth	16
Data Width	8 bits
UART Protocol	1 Start, 8 Data, 1 Stop

Given below is the schematic of the designed system.



The flow of the data can be generalized as

Parallel data ----->FIFO----->FSM----->UART packetizer
Given below is the complete description of all the modules.

FIFO Module Description

This Verilog module implements a **synchronous FIFO (First-In-First-Out) buffer** with configurable data_width and depth parameters. It allows concurrent read and write operations, synchronized with the clock signal.

Internally, the FIFO uses a memory array (fifo) to store data, and two pointers: rd_ptr for reading and wr_ptr for writing. A 5-bit counter (count) tracks the number of valid entries in the FIFO.

Write Operation:

On every rising edge of the clock, if reset is not active and the FIFO is not full, data is written into the FIFO when both **wr_en (write enable)** and **data_valid** signals are high. The data is stored at the location pointed to by **wr_ptr**, and the pointer is incremented afterward.

Read Operation:

Data is read from the FIFO when rd_en (read enable) is high and the FIFO is not empty. The value at the location pointed to by rd_ptr is output through d_out, and the pointer is incremented.

Status Flags:

The module uses the count register to manage fifo_full and fifo_empty flags. When a valid write occurs and the FIFO is not full, the counter is incremented. When a valid read occurs and the FIFO is not empty, the counter is decremented

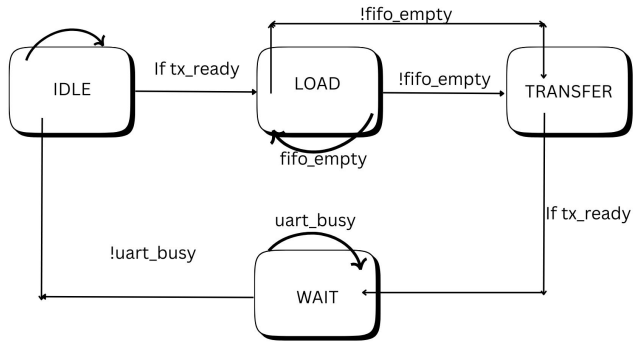
```
case ({wr_en && data_valid && !fifo_full, rd_en && !fifo_empty})
  2'b10: count <= count + 1;
  2'b01: count <= count - 1;
  default: count <= count;
endcase
fifo_full <= (count == depth - 1);
fifo_empty <= (count == 0);
```

Why Synchronous FIFO ?

I implemented a synchronous FIFO because all modules in my design—FSM, FIFO, and UART—operate under the same clock domain (`clk`). Since there was no need to transfer data across different clock domains, a synchronous FIFO was the most efficient and appropriate choice.

It simplifies the design, reduces resource usage, and avoids the complexity associated with clock domain crossing, such as metastability handling and pointer synchronization. For this use case, where data flows smoothly within a single clocked system, a synchronous FIFO ensures both reliability and clarity.

FSM Module Description



This Verilog module implements a **finite state machine (FSM)** that controls UART data transmission using signals from a FIFO buffer. It operates on a 2-bit state system with four states: IDLE, LOAD, TRANSFER, and WAIT.

The FSM takes the following inputs:

- clk and rst for synchronization and reset,
- d_in as data input from the FIFO,
- uart_busy indicating ongoing UART transmission,
- fifo_empty flag to check FIFO status,
- tx_ready indicating UART is ready for new data.

It produces:

- tx_start to initiate transmission,
- rd_en to trigger a FIFO read,
- fsm_busy indicating the FSM is active.

FSM Encoding Description

The FSM in the fsm_dig module uses **sequential encoding**, where each state is assigned a unique binary value in a natural counting sequence. The four states—IDLE, LOAD, TRANSFER, and WAIT—are encoded as 2-bit binary codes from 00 to 11. This approach minimizes the number of bits needed to represent states and is efficient for small FSMs. The synthesis tool identifies and preserves this encoding during synthesis, as confirmed by the reported state register encodings matching the assigned binary values.

UART Transmitter Module Description

This module implements a **UART transmitter** that serializes parallel data and sends it over a single line (ser_data) at a defined baud rate. It is parameterized to support configurable clock frequency, baud rate, and data width.

Parameters:

- clk_freq: System clock frequency (default: 50 MHz)
- baud_rate: UART baud rate (default: 9600)
- data_width: Width of the data to be transmitted (default: 8 bits)

Inputs:

- d_in: Parallel data input of width data_width
- tx_start: Signal to trigger data transmission
- clk: System clock
- rst: Active-high reset

Outputs:

- ser_data: Serial output line (UART TX)
- uart_busy: Indicates that a transmission is in progress

Functionality Overview:

On tx_start, and if uart_busy is low, the module loads d_in into a 10-bit data_frame structured as:

- 1 start bit (0)
- 8 data bits (d_in)
- 1 stop bit (1)

The module then serially transmits data_frame bit by bit via ser_data based on the baud rate timing. The baud rate is derived using the formula:

clk_per_bit = clk_freq / baud_rate

Transmission timing is managed by a clk_count counter which divides the system clock to match the baud rate.

A bit_count counter tracks the number of bits transmitted. Once all 10 bits (start, data, stop) are sent, uart_busy is de-asserted and ser_data returns to idle high.

Key Control Logic:

- Transmission begins when tx_start is high and uart_busy is low.
- During transmission, uart_busy remains high to block new transfers.
- Each bit is held on ser_data for one full baud cycle.

This module ensures standard UART frame formatting and stable baud rate timing for reliable serial communication.

TestBench Details .

Testbench Description
The testbench is designed to validate the functionality of the uart_packetizer_dig module by simulating the transmission of two data packets. It initializes the system, applies reset, feeds two 8-bit data values sequentially into the DUT, and simulates tx_ready to allow transmission to begin. The goal is to observe correct data packet storage and serial output behavior.

Signal Application Timeline

0 ns:
clk = 0, rst = 1, data_valid = 0, tx_ready = 0, wr_en = 0
→ Initial system state, reset asserted.

40 ns:
rst = 0
→ Reset released; normal operation starts.
60 ns:
data = 0xAC, wr_en = 1, data_valid = 1
→ First packet written to FIFO.
80 ns:
data_valid = 0
→ First write complete.
120 ns:
data = 0xCA, data_valid = 1
→ Second packet written to FIFO.
140 ns:
data_valid = 0, wr_en = 0
→ Second write complete.
160 ns:
tx_ready = 1
→ UART ready to transmit buffered data.

Results