UART Packetizer with FSM and FIFO Integration

Introduction

The system is designed to accept 8-bit parallel data packets, buffer them using an **asynchronous First-In, First-Out (FIFO)** memory, and transmit them serially via a custom **UART** protocol. The design is modular, comprising a Packetizer Finite State Machine (FSM), an asynchronous FIFO, a UART-like serial output block, and a top-level interconnecting module.

Assumed Parameters:

- System Clock (clk): 50 MHz
- Default Baud Rate: 115200 bps
- FIFO Depth: 16 words
- FIFO Data Width: 8 bits
- Target FPGA : AMD-Xilinx Kintex-7 xc7k70tfbg676-1

1.a) Design Verilog RTL Code for the Modules

- **Asynchronous FIFO Buffer** (fifo_async.v): This module buffers incoming 8-bit data words. It is designed to handle potentially different clock domains for write and read operations, using **Gray code pointers** for robust clock domain crossing (CDC). It provides fifo_full, fifo_empty, and a data_out_valid signal. The depth is 16 words.

- **Packetizer FSM** (packetizer_fsm.v): This state machine manages the data flow. It monitors the FIFO status, waits for an external tx_ready handshake signal, initiates FIFO reads, and commands the UART block to start serial transmission.

- **UART Serial Output Block** (uart_tx.v): This module takes an 8-bit parallel data byte from the FSM (originating from the FIFO) and transmits it serially. The frame consists of **one start bit (low), eight data bits (LSB first), and one stop bit (high)**. A baud rate generator, derived from the system clock, controls the bit transmission speed.

- **Top Module** (uart_packetizer_top.v): This module instantiates and interconnects the FIFO, FSM, and UART transmitter, mapping system-level inputs and outputs to the appropriate sub-module ports. For the described system, both FIFO write and read clocks, as well as the FSM and UART clocks, are driven by the main 50 MHz system clk.

1.b) Simulation Testbench and Results

In this testbench for the uart_packetizer_top module, a structured sequence of operations is used to verify UART transmission functionality and control logic. Initially, the DUT is reset and the clock is generated. The testbench simulates writing multiple bytes (up

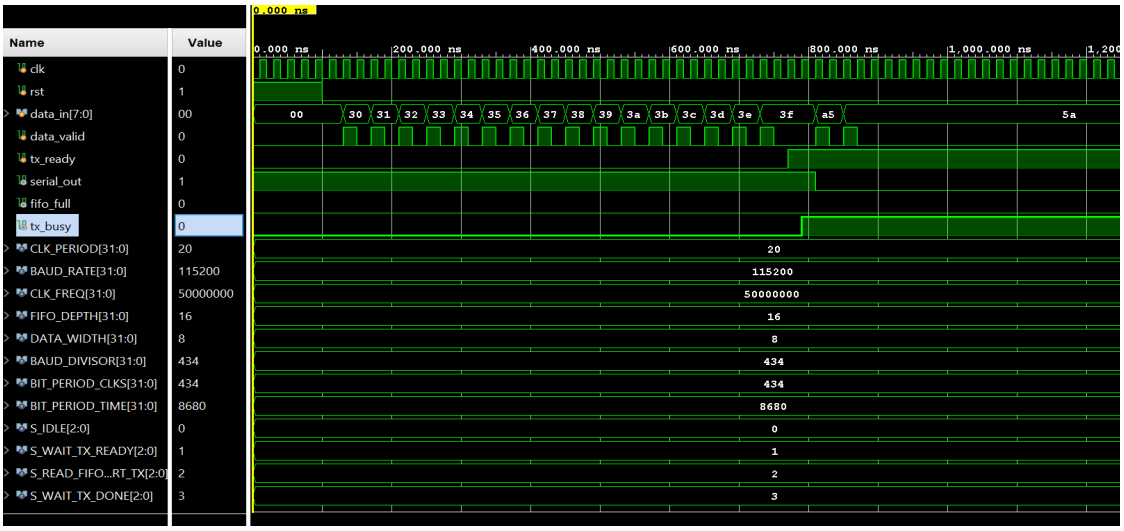
to FIFO depth) into the input FIFO while tx_ready is low, validating proper FIFO filling by checking the fifo_full signal. Next, transmission is enabled by asserting tx_ready, and the system waits for the UART to finish transmitting, confirming FIFO emptiness afterward.

To verify the serial output, specific bytes (0x5A, and 0xF0) are written and their complete UART frame structure (start bit, data bits, and stop bit) is checked against the actual serial_out. A check_packet task performs bit-by-bit verification over calculated baud-aligned intervals. Additionally, the FSM state is monitored after writing a byte while tx_ready is deasserted to ensure the design enters the expected S_WAIT_TX_READY state. The testbench concludes after all checks pass, demonstrating functional coverage of FIFO behavior, UART serialization, and FSM transitions.

1.FIFO Write and fifo_full Assertion

- Timeframe: At the beginning of simulation.
- Signals to watch:

  data_valid: should pulse high 16 times, one per byte (0x30 to 0x3F).
  data_in: carries **0x30 to 0x3F** sequentially.



2.UART Transmission of FIFO Data

- After tx_ready = 1 is asserted, UART transmission begins.
- Signals to watch:

  serial_out: This should toggle for each bit of each UART frame:
  Each byte will take 10 bits × baud period (≈ 86.8µs per byte at 115200 baud).
  tx_busy: Goes high during active transmission.
  fifo_full: Goes low as FIFO empties.

3. FIFO Empty and Final State

Once all FIFO data is sent, fifo_empty becomes high.
FSM returns to S_IDLE unless another byte arrives.

4. Packet Check for 0x5A

UART waveform output (serial_out) for **0x5A (8'b01011010)** will look like this:

Start bit: 0
Data bits (LSB first):0 1 0 1 1 0 1 0
Stop bit: 1

5. Extra Packet (0xF0) with tx_ready = 0

FSM pauses in S_WAIT_TX_READY state.
Transmission resumes only after you set tx_ready = 1.
Then serial output for 0xF0 begins (**frame = 0_00001111_1** in UART bits).

Complete Waveform timing:

Time	Signal	What Happens
0–320 ns	data_valid, data_in	16 writes to FIFO
After	fifo_full = 1	FIFO full
+ Delay	tx_ready = 1	Transmission starts
Next ~1.4ms	serial_out	UART bytes sent (0x30–0x3F)
tx_busy = 1	High during active transmission	
fifo_empty = 1	After all 16 bytes sent	
Later	0x5A sent	Check framing
Finally	0xF0 sent after pause	FSM waits, resumes on tx_ready

Learnings:

I’ve faced the unexpected glitches or spikes on the UART TX line, especially around bit transitions, even though the design logic appeared correct. In my case, it was caused by unregistered combinational changes to serial_out directly within the FSM.

How I resolved it:

Originally, my code was driving serial_out directly like this:

serial_out <= tx_shift_reg[0];

This meant that serial_out could change state immediately based on logic updates, even between clock edges — leading to glitches/spikes, especially in simulations or when synthesized without proper constraints.

What I changed:

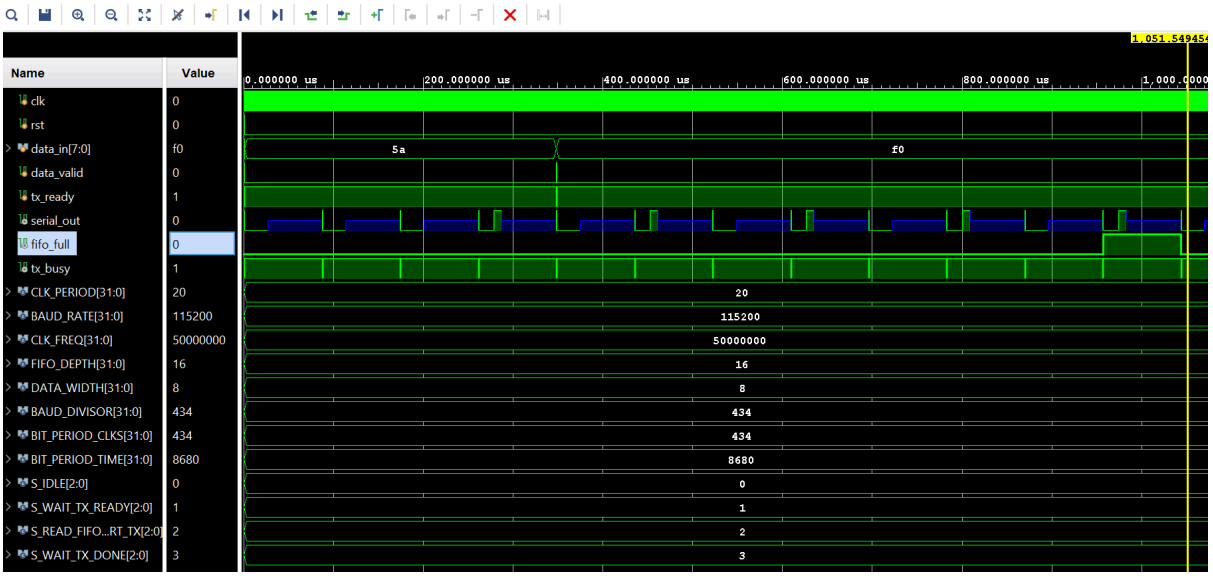
I introduced a new register, serial_out_next, and made sure all changes to the serial line were registered properly:

- First, compute serial_out_next based on logic.
- Then, update serial_out on the next clock edge.

This way, serial_out only transitions on clk, making it free from glitches and metastability.

Here’s the key change:

```
always @(posedge clk or posedge rst) begin
 if (rst) begin
        serial_out <= 1'b1;
 end else begin
         serial_out <= serial_out_next;
 end
 end
```

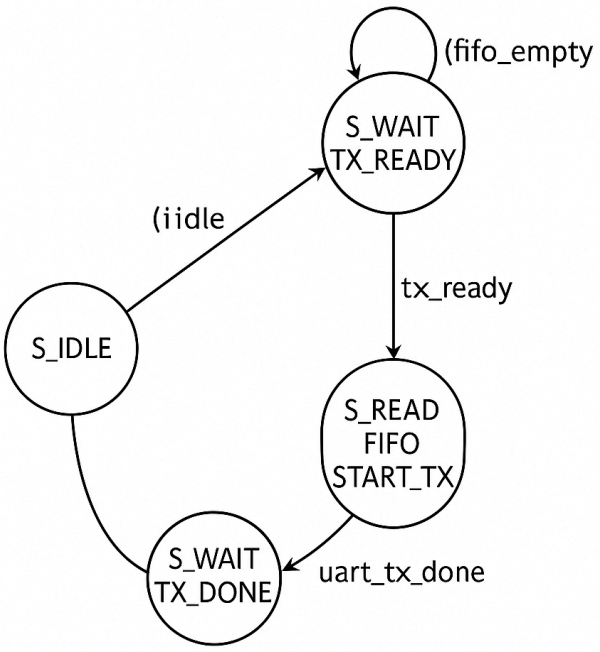


Now, **serial_out** is 100% clocked and deterministic, removing those sudden spikes and unknowns in waveform.After this fix, the UART TX waveform showed clean, edge-aligned transitions, and simulation matched the protocol spec perfectly.

The **Z state** seen between bytes is likely due to serial_out going **tri-state during idle** — this is often tool-specific in behavioral simulation.

Mistakes	Effect	Fix
FIFO read while empty	Unknown ('XX') data sent	Gate fifo_rd_en with !fifo_empty
Holding start_tx too long	Invalid UART triggering	Pulse start_tx for 1 cycle
Using FIFO output directly	Data glitches during transfer	Latched FIFO output to fifo_data_latched
serial_out glitches and 'Z'	Unstable UART output	Registered serial_out_next added
Idle line not explicitly held	Floating line	Set serial_out <= 1'b1 when idle

1. c) FSM state diagram along with state transitions.



S_IDLE (000)
S_READ_FIFO (001)
S_LATCH_AND_START_TX (010)
S_WAIT_TX_DONE (011)

Current State	Input Conditions	Next State	fifo_rd_en	uart_start_tx	uart_data_to_tx	tx_busy
S_IDLE (000)	!fifo_empty	S_WAIT_TX_READY	0	0	8'h00	0
S_IDLE (000)	fifo_empty	S_IDLE	0	0	8'h00	0
S_WAIT_TX_READY (001)	fifo_empty	S_IDLE	0	0	8'h00	0
S_WAIT_TX_READY (001)	!fifo_empty && tx_ready	S_READ_FIFO_START_TX	0	0	8'h00	0
S_WAIT_TX_READY (001)	!fifo_empty && !tx_ready	S_WAIT_TX_READY	0	0	8'h00	0
S_READ_FIFO_START_TX (010)	don’t care	S_WAIT_TX_DONE	1	1	fifo_data_out	1
S_WAIT_TX_DONE (011)	uart_tx_done	S_IDLE	0	0	8'h00	1
S_WAIT_TX_DONE (011)	!uart_tx_done	S_WAIT_TX_DONE	0	0	8'h00	1

FIFO Usage Explanation

The asynchronous FIFO (fifo_async.v) acts as an **elastic buffer,** decoupling the external data source (writing data in parallel) from the packetizing and serial transmission process (reading data).

- **Write Operation**: When the external source provides an 8-bit data word on data_in and pulses data_valid high, the data is written into the FIFO, provided fifo_full is not asserted. This operation is synchronized to the FIFO's write clock (wr_clk, which is the system clk in this top-level configuration). The internal write pointer increments.

- **Read Operation**: The Packetizer FSM monitors the FIFO's status via fifo_has_data_for_fsm (derived from fifo_empty). When data is available and the FSM is ready to process it (i.e., in S_READ_FIFO_START_TX state after tx_ready handshake), it asserts fifo_rd_en for one clock cycle. This causes the oldest data word in the FIFO to be presented on its rd_data output port, which is then captured by the UART module. This operation is synchronized to the FIFO's read clock (rd_clk, also system clk). The internal read pointer increments.

Rationale for Synchronous/Asynchronous FIFO:

An asynchronous FIFO was chosen for this design due to the requirement that the system accepts data from an "**external data source**". This phrasing strongly implies that the clock associated with the incoming **data_in and data_valid signals** (the write clock domain) may be **different** from, or asynchronous to, the system's main 50 MHz clk (the read clock domain, used by the FSM and UART transmitter).

Attempting to pass data between two asynchronous clock domains without proper synchronization mechanisms, such as those employed in an asynchronous FIFO, would lead to a **high probability of metastability** in the flip-flops sampling the cross-domain signals.

An asynchronous FIFO is specifically designed to handle data transfer across such clock domain boundaries safely. It achieves this through **Independent Write and Read** Pointers and Pointer Synchronization.

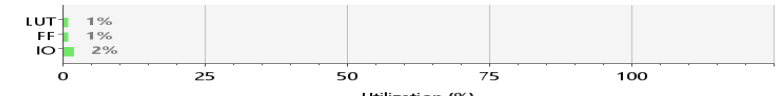
Question 2

c) Report the resource utilization and achieve maximum operating clock frequency at Synthesis level and post-place and route level?

Resource utilization at Synthesis level:

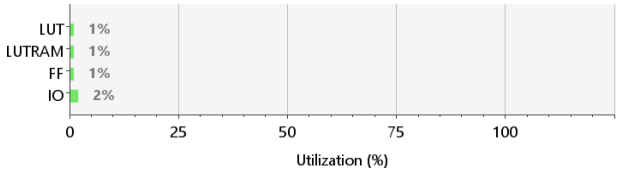
Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)	BUFGCTRL (32)
uart_packetizer_top	50	70	8	1
fifo_inst (async_fifo)	22	40	0	0
fsm_inst (packetizer_fsm)	2	2	0	0
uart_tx_inst (uart_tx)	26	28	0	0

Resource	Utilization	Available	Utilization %
LUT	50	134600	0.04
FF	70	269200	0.03
IO	8	400	2.00



Resource Utilisation at post-place and route level:

Resource	Utilization	Available	Utilization %
LUT	50	134600	0.04
LUTRAM	2	46200	0.00
FF	70	269200	0.03
IO	8	400	2.00



Timing Report:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.837 ns	Worst Hold Slack (WHS):	0.016 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	45783	Total Number of Endpoints:	45783	Total Number of Endpoints:	16113

All user specified timing constraints are met.

Category	Value	Interpretation
WNS	0.837 ns	Positive slack ⇒ met setup timing
THS	0.016 ns	Positive hold slack ⇒ hold is met
WPWS	3.750 ns	Pulse width slack is healthy
Total Failing Endpoints	0	No paths are violating constraints

Maximum operating clock frequency

WNS = +0.837 ns

So, the data path delay is: 20.000 ns - 0.837 ns = **19.163 ns**
Max operating clk frequency ≈ 1 / 19.163 ns ≈ **52.2 MHz**

d) Describe the encoding technique used in the FSM implementation. Provide a tool generated report indicating the type of encoding technique used for FSM implementation.

```
---------------------------------------------------------------------------
INFO: [Synth 8-802] inferred FSM for state register 'current_state_reg' in module 'packetizer_fsm'
---------------------------------------------------------------------------------------------------
                   State |                     New Encoding |               Previous Encoding
---------------------------------------------------------------------------------------------------
                  S_IDLE |                               00 |                              000
          S_WAIT_TX_READY |                               01 |                              001
    S_READ_FIFO_START_TX |                               10 |                              010
          S_WAIT_TX_DONE |                               11 |                              011
---------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'current_state_reg' using encoding 'sequential' in module 'packetizer_fsm
---------------------------------------------------------------------------
```

Encoding Technique : **Sequential**

e) Propose techniques to increase the maximum operating frequency of the RTL modules developed in question 1.

1. **Pipelining**: It involves identifying the **longest combinational logic path** in the design (the critical path, identifiable from Vivado timing reports and inserting registers (flip-flops) within this path. This breaks the long path into **two or more shorter paths**, each of which can operate at a higher frequency. While pipelining increases throughput (Fmax), it also introduces latency (additional clock cycles for data to pass through).

2. **Retiming**: Retiming is an optimization technique where registers are moved across combinational logic gates without altering the functional behavior or the number of

registers in a cycle (latency).The goal is to balance delays within clock cycles more effectively.

3. **Logic Optimization and Restructuring**:

Reduce Fanout: High-fanout nets (a signal driving many loads) can cause significant delays. If a critical path involves a high-fanout net, replicating the driver logic can reduce the load on each copy, potentially improving timing.

Simplify Complex Logic: Review complex combinational logic (e.g., nested if-else, wide comparators) in critical paths. Sometimes, restructuring this logic (e.g., using case statements instead of long if-else-if chains if appropriate, or breaking down wide operations) can lead to a more efficient implementation by the synthesis tool.

Optimize Arithmetic Operations: If arithmetic operations are part of the critical path, ensure they are implemented efficiently. For instance, check if dedicated DSP blocks are being used appropriately if applicable, though this design is unlikely to require them.

4. **State Encoding for FSM (If FSM is Critical**): While "auto" FSM encoding is usually effective, if the FSM is identified as the bottleneck, explicitly setting the FSM encoding (e.g., to "one-hot" or "binary") via RTL attributes can be tried to see if it improves timing for that specific FSM structure.