Project:
UART

CEGG 3155– DIGITAL SYSTEMS II

Fall 2025

School of Electrical Engineering and Computer Science

University of Ottawa

Course Coordinator: Dr. Rami Abielmona

Isaiah Kwapisz 300377196
David Rosocha 300366191

Experiment Date: 11/3/2025
Submission Date: 12/01/2025

Method and Materials

2.1 Materials:

• Quartus II Web Edition for VHDL coding, simulation, and synthesis.

• Altera DE2 Development and Education board for experimental verification.

Specifically, for this lab, the Cyclone IV EP4CE115F29C

• Multisim for Verification/Testbenches

2.2. Method

The design of the traffic light controller was realized using the Finite State Machine (FSM) procedure outlined in the manual, which provides a systematic approach to developing synchronous sequential circuits. The design flow consisted of the following steps:

1. **Interpreting the problem specifications**
   The functional requirements of the traffic light controller were examined, including input conditions, output behavior, timing constraints, and priority rules between the main and side streets.

2. **Creating a state diagram**
   A state diagram was constructed to represent all possible light configurations and their transitions based on sensor input and timer conditions.

3. **Deriving the state table**
   The state diagram was converted into a state table, expressing present states, next states, and output values for all input combinations.

4. **Performing state minimization and assignment**
   Equivalent states were reduced to simplify the implementation, and a binary state encoding was selected to represent each state using flip-flops.

5. **Generating the transition table and design equations**
   The minimized state table was expanded into a transition table and corresponding excitation/output equations were derived for implementation using D-type flip-flops.

6. **Implementing the circuit**
   The resulting logic was implemented in VHDL at the structural level and synthesized onto the FPGA, forming a synchronous FSM that drives the traffic lights and auxiliary components.

Introduction

The primary objective of this project was to design, implement, and verify a complete Universal Asynchronous Receiver-Transmitter (UART) system using VHDL and the Altera Quartus II software. While Laboratory #3 focused on designing a Finite State Machine (FSM) to control a traffic light system with internal states visible only through LED indicators, this project transforms that isolated controller into a debuggable real-time system by integrating a serial communication interface. The core purpose is to enable the FPGA to transmit human-readable ASCII debug messages, such as "Mg-Sr" (Main Green, Side Red), to an external computer terminal whenever a traffic light state transition occurs.

UART Fundamentals and Asynchronous Communication

A UART facilitates asynchronous serial data transmission between digital systems. Unlike parallel communication, which transmits multiple bits simultaneously across dedicated wires, serial communication transmits data sequentially, one bit at a time, over a single wire. The term "asynchronous" means UART operates without a shared clock signal, requiring only two data lines (TxD for transmission and RxD for reception) plus a common ground. Both the transmitter and receiver must operate at the same pre-agreed data rate, known as the baud rate, and the receiver must accurately detect when data transmission begins and ends without any timing reference from the transmitter.

The baud rate, measured in bits per second (bps), defines transmission speed. This project supports multiple selectable baud rates (300-38400 bps) through a programmable Baud Rate Generator controlled by a 3-bit selector signal (SEL[2:0]). The UART protocol organizes data into frames: 1 start bit (logic '0') + 8 data bits (LSB first, ASCII-encoded) + 0 parity bits + 1 stop bit (logic '1'), totaling 10 bits per character.

Oversampling and Reception Strategy

A critical design challenge is accurately sampling the incoming asynchronous serial data stream. Since the receiver's clock is not synchronized with the transmitter's clock, directly sampling the RxD line at the baud rate clock (BClk) could result in sampling transitions rather than stable bit values. To mitigate this, the receiver employs an oversampling technique, sampling the RxD line at eight times the baud rate (BClk×8).

When the receiver detects the falling edge of the start bit, it waits 4 BClk×8 clock cycles (half the bit period) to position itself at the center of the start bit. It then continues sampling every 8 BClk×8 cycles to capture each data bit at its center point, providing robustness against noise, clock drift, and metastability issues.

Structural Design Architecture

The project required strict adherence to structural modeling—all atomic modules such as D flip-flops, multiplexers, comparators, shift registers, and counters were realized using Register Transfer Level (RTL) logic without behavioral constructs or pre-made IP cores. Key components include:

- **Baud Rate Generator**: Cascaded frequency dividers producing selectable baud rates from the 50 MHz system clock
- **Transmitter Module**: 8-bit Transmit Data Register (TDR), 8-bit Transmit Shift Register (TSR), and control FSM

- **Receiver Module**: 8-bit Receive Shift Register (RSR), 8-bit Receive Data Register (RDR), and control FSM
- **UART FSM Controller**: Manages data flow and handshaking between the traffic light controller and UART

The UART interfaces with the system through memory-mapped registers: TDR, RDR, Serial Communications Control Register (SCCR), and Serial Communications Status Register (SCSR). The SCCR contains control bits (TIE, RIE, SEL[2:0]), while the SCSR contains status flags (TDRE, RDRF, OE, FE) providing real-time operational feedback.

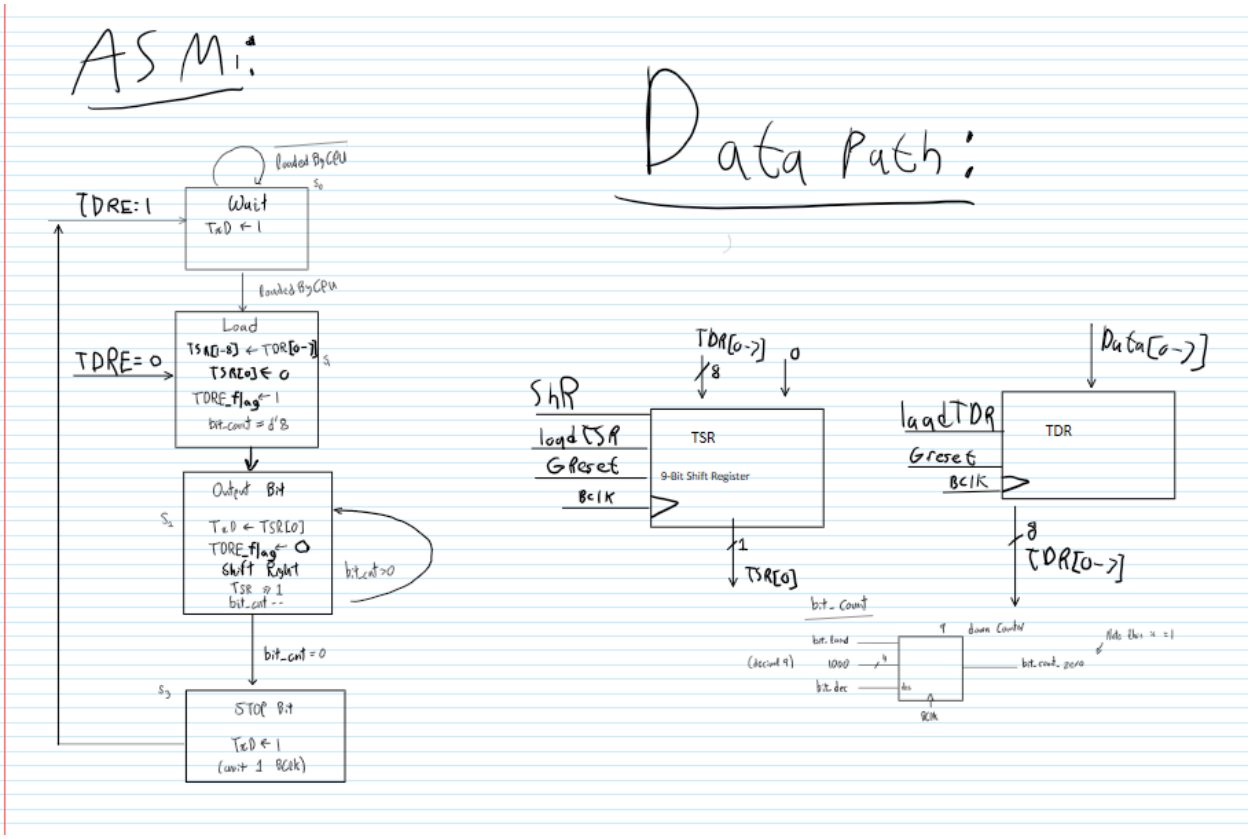
The traffic light FSM outputs a 3-bit state encoding signal representing the four main traffic light states. A dedicated UART FSM monitors this encoding and, upon detecting a state transition, initiates transmission of a corresponding 6-character ASCII debug message. The FSM monitors the TDRE status flag to ensure the previous character has been loaded before writing the next character, implementing a handshaking protocol that prevents data loss.

RS-232 Level Conversion and Project Significance

The electrical interface between the FPGA's CMOS logic levels (+3.3V/0V) and the RS-232 serial port standard (±12V) requires a MAX232 level-shifter integrated circuit. This device contains charge-pump circuitry that generates the ±12V voltages and converts between CMOS and RS-232 levels bidirectionally, preventing signal incompatibility and potential FPGA damage.

This project demonstrates modular system-on-chip design, clock domain crossing, finite state machine coordination, and real-time debugging methodologies. The skills developed—precise timing generation, protocol implementation, modular hierarchical design, and hardware-software interfacing—are directly applicable to embedded systems development and FPGA-based prototyping in industry.

Diagrams | Design Process via ASM Methodology

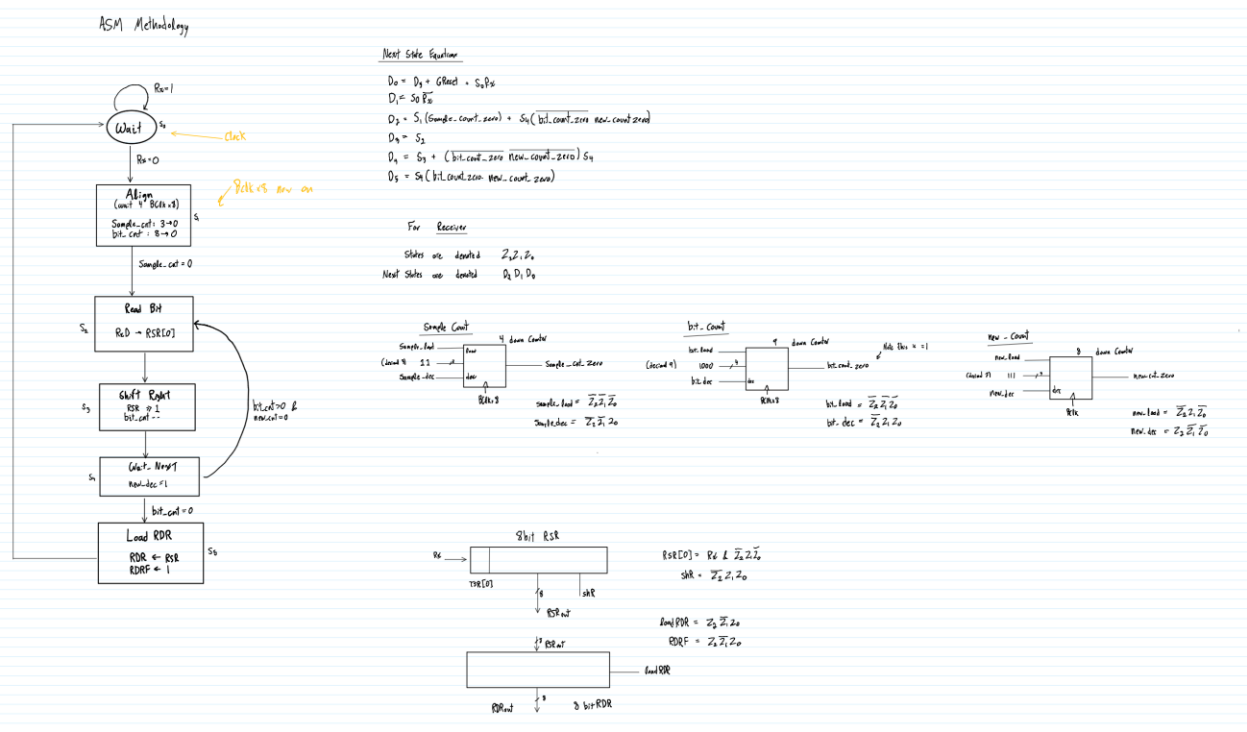


Transmitter ASM

Transmitter Explanation: TSR is a 9 bit register, which loads 0 into its LSB. Whenever it is shifted right, the MSB has a 1 loaded into it. Resetting the TSR loads all 1s into the register. Whenever bits are loaded into TSR, the transmitter sets a TDRE_Flag for a single clock pulse.

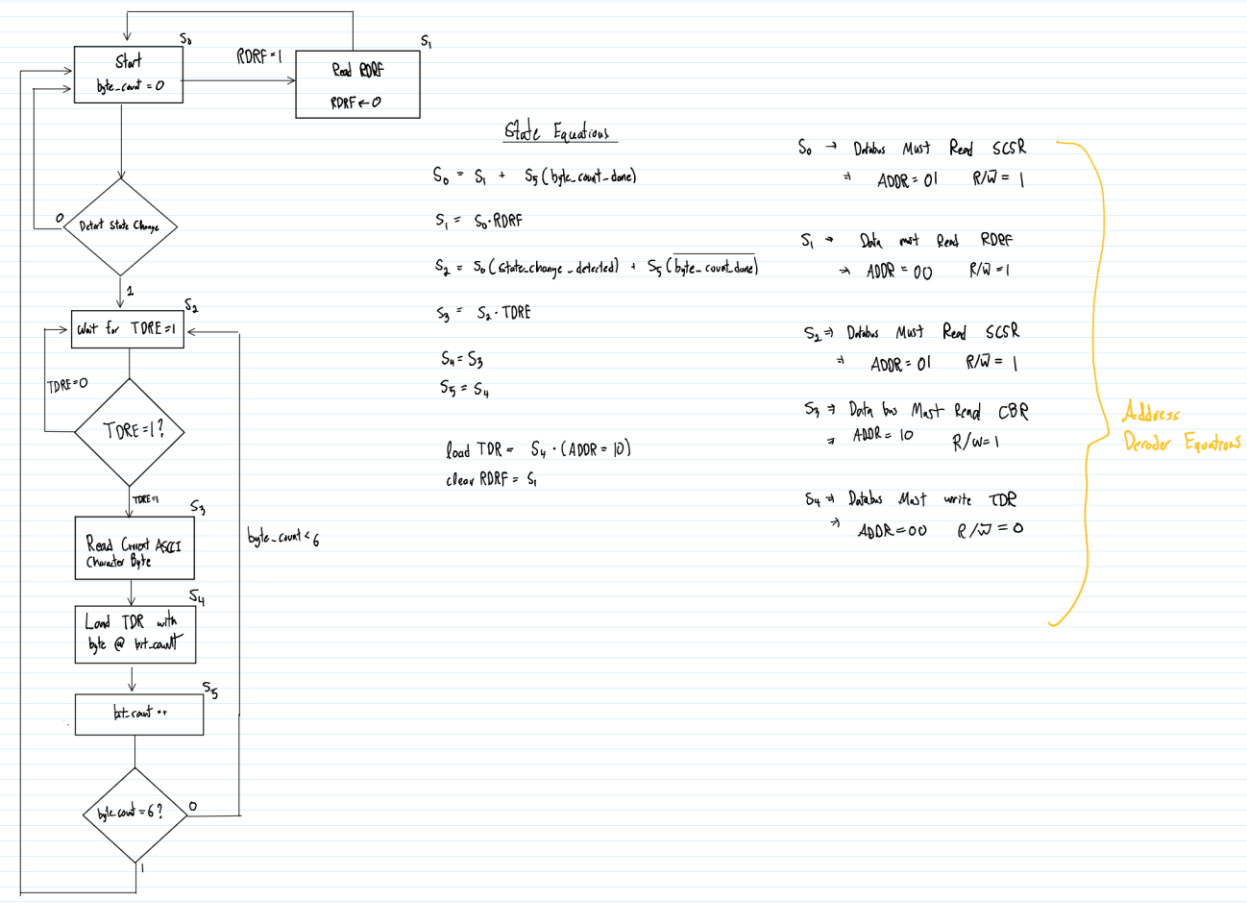
Design Equations:

- D0 = Q0T*DRE + Q3*TDRE
- D1 = Q0*TDRE’ + Q3*TDRE’
- D2 = Q1 + Q2*bit_count_zero’
- D3 = Q2*but_count_zero’
- LoadTSR = Q1
- ShR = Q2
- Bit_load = Q1
- Bit_dec = Q2
- TDRE_flag = Q1
- LoadTDR = load (input signal)

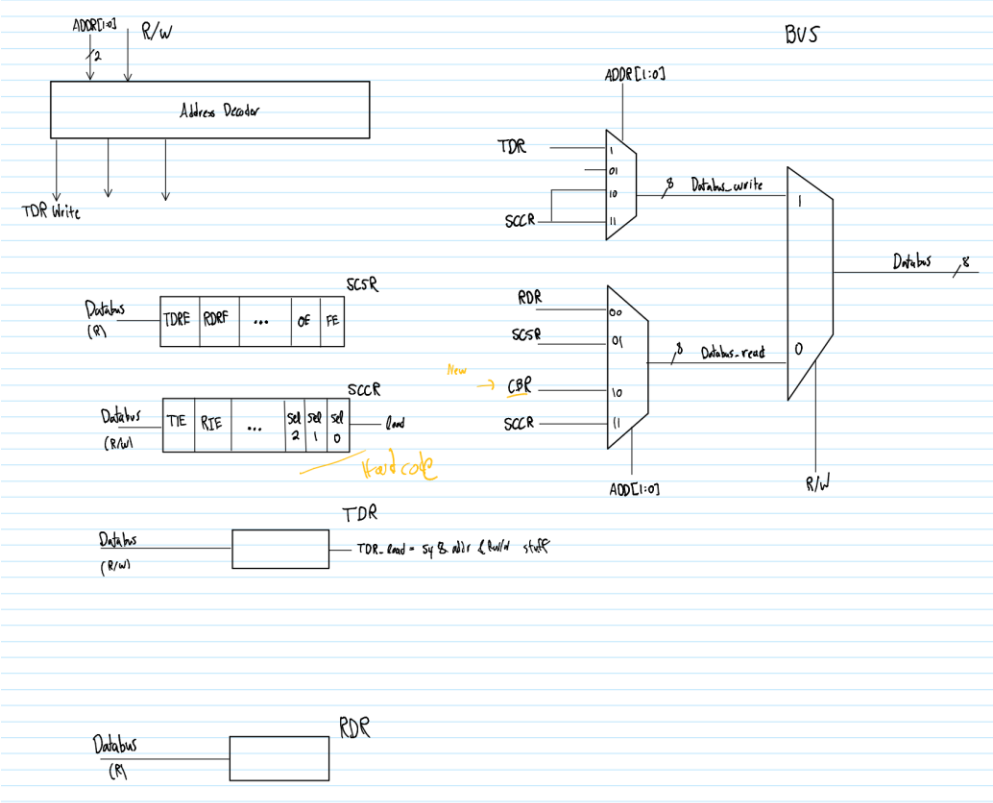


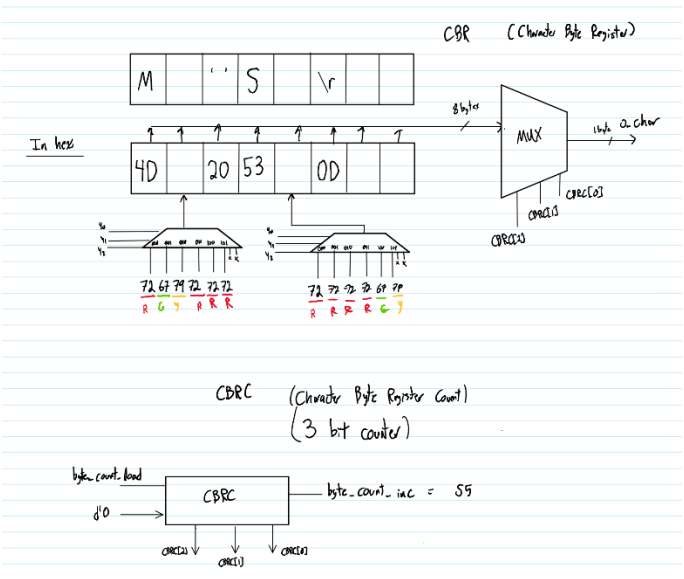
Note: we shift right 9 times to eliminate the start bit, hence why bit_count loads decimal 9.

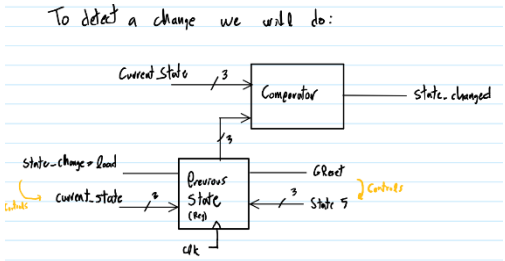
Receiver ASM



Microcontroller ASM







States	Encoding	Description
S_0	000	Mr Sr
S_1	001	Mg Sr
S_2	010	My Sr
S_3	011	Mr Sr
S_4	100	Mr Sg
S_5	101	Mr Sy

Microcontroller Datapath

Design Explanation

Discussion of Algorithmic Solution

The "Debug Loop" Algorithm

The algorithmic solution for this project introduces a secondary "observer" loop, the UART Debugger, that runs in parallel with the main traffic light logic. Unlike a standard sequential circuit that advances on every clock edge, this algorithm operates on an asynchronous "Request-Acknowledge" basis to ensure data integrity between the fast 50 MHz FPGA logic and the slow 9600 baud serial interface.

The operational algorithm is defined by the following sequential steps:

1. **State Monitoring:** The system continuously monitors the 3-bit state output (S2,S1,S0) from the Traffic Light Controller.
2. **Edge Detection:** To filter out stable states and react only to transitions, a comparison algorithm is used. The current state $S(t)$ is compared with the registered previous state S(t-1). If S(t) does not equal S(t-1), a single-cycle trigger pulse is generated.
3. **Interrupt Service Simulation:** This trigger acts as a pseudo-interrupt for the UART FSM. Upon receiving the trigger, the FSM resets the internal Byte Counter to zero and enters a "Fetch" state.
4. **Fetch-and-Transmit Loop:** The FSM uses the current traffic state and the byte count to address the Character Byte Register (CBR) lookup table. It retrieves the corresponding ASCII character (e.g., ‘M’) and asserts the Load signal to the transmitter.
5. **Handshaking:** Crucially, the algorithm prevents buffer overrun by polling the **TDRE** (Transmit Data Register Empty) status flag. The FSM holds the current byte and waits until TDRE returns to logic ‘1’ before incrementing the Byte Counter. This cycle repeats for exactly six bytes (the length of the debug message) before the FSM returns to the IDLE state[3].

UART Transmitter Algorithm

The transmitter algorithm is designed to manage the serialization of data while adhering to the timing constraints of the baud rate. As illustrated in the Transmitter ASM diagram, the logic relies on a handshake between the system clock and the baud clock.

1. **Idle/Wait State**: The controller remains in the IDLE state until a Load signal is received from the CPU (UART FSM), and Tx constantly displays (‘1’) indicating idling at output.
2. **Load State**: Upon receiving the load signal from the microcontroller, the 8-bit data is latched from the data bus into the Transmit Data Register (TDR). The algorithm then loads this data into the 9-bit Transmit Shift Register (TSR), effectively appending the Start Bit (logic '0') to the LSB position. Simultaneously, the Bit Counter is loaded with the frame length (10).
3. **Shift Loop**: The controller enters a transmission loop driven by the Baud Clock (BClk). In every cycle, the TSR performs a right shift, pushing the Start Bit, then the Data Bits, onto the TxD line.
4. **Completion**: The loop continues until the Bit Counter reaches zero (bit_cnt_z), indicating the Stop Bit has been sent. The controller then asserts the TDRE (Transmit Data Register Empty) flag, signaling to the CPU that it is ready to accept a new byte, and returns to the IDLE state.

3.2.2 UART Receiver Algorithm

The receiver algorithm utilizes an oversampling technique to filter noise and synchronize with the asynchronous incoming stream. As detailed in the Receiver ASM diagram, the control logic transitions through six distinct states (S_0 to S_5) using One-Hot encoding:

1. **Start Bit Detection ($S_0 \rightarrow S_1$)**: The algorithm waits in state S_0 for the RxD line to drop to logic '0'. Once detected, it transitions to S_1 and loads the Sample Counter.
2. **Alignment ($S_1 \rightarrow S_2$)**: In state S_1, the algorithm counts down 4 clock cycles (half a bit period at 8× oversampling) to align the sampling point with the center of the Start Bit.
3. **Sampling Loop ($S_2 \rightarrow S_3 \rightarrow S_4$)**: The system enters a loop where it reads the RxD value (S_2), shifts it into the Receive Shift Register (RSR) (S_3), and decrements the Bit Counter (S_4). This loop repeats 9 times to capture the ASCII character and remove the start bit.
4. **Load and Flag (S_5)**: Once the Bit Counter expires, the algorithm transitions to S_5. Here, it parallel-loads the data from the RSR into the Receive Data Register (RDR) and asserts the RDRF (Receive Data Register Full) flag, alerting the system that valid data is available.

3.2.3 Microcontroller (UART FSM) Algorithm

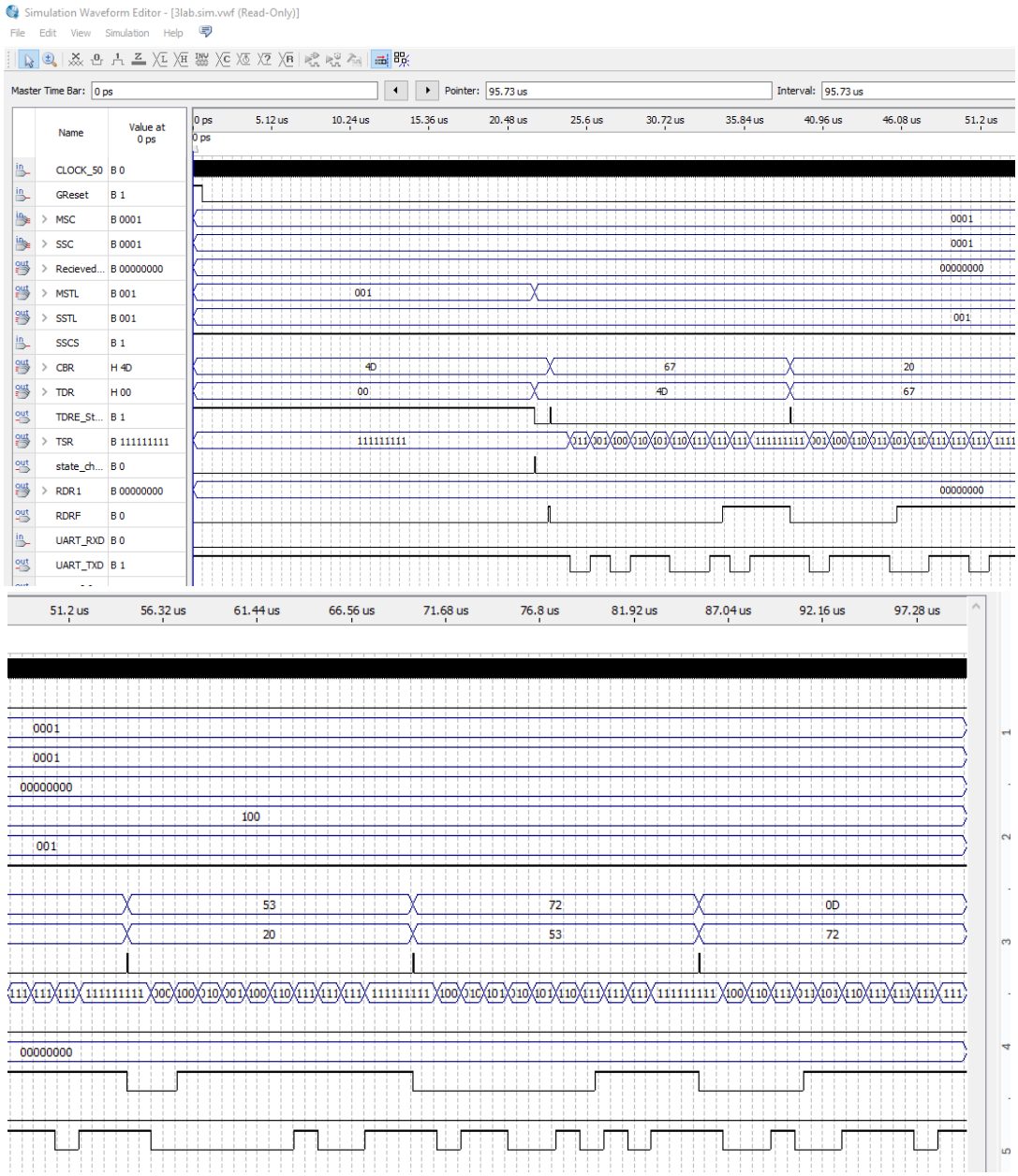
The UART FSM acts as the central processor, orchestrating the interaction between the traffic light states and the UART datapath. Its algorithm, shown in the Microcontroller ASM diagram, follows a "Fetch-and-Send" cycle:

1. **Event Detection (S_0)**: The FSM polls the State_Change_Detector. If a change in the traffic light state ($S_{new} \neq S_{old}$) is detected, it proceeds to S_1 and resets the Byte Counter to 0.
2. **Check RDRF ($S_0 \rightarrow S_1$)**: Before sending data, the FSM checks the RDRF status flag. This signals the microcontroller to load the contnets in the RDR to the databus, then set RDRF back to 0 to show we are ready to read the next byte.
3. **Handshake Verification ($S_0 \rightarrow S_2$)**: Before sending data, the FSM checks the TDRE status flag. It loops in state S_1 until TDRE is '1', ensuring the transmitter is idle.
4. **Data Fetch ($S_2 \rightarrow S_3$)**: The FSM uses the current traffic state and byte count to address the Character Byte Register (CBR). It retrieves the specific ASCII character for that position in the message string.
5. **Transmission Trigger (S_4)**: The FSM executes a write operation via the Address Decoder (Address "00", Write Enable '0'), latching the character into the Transmitter's TDR.
6. **Increment and Loop (S_5)**: The Byte Counter is incremented. The algorithm checks if the count has reached 6 (the length of the debug message). If not, it loops back to S_1 to send the next character; otherwise, it returns to S_0 to await the next traffic light transition.

Simulated Results

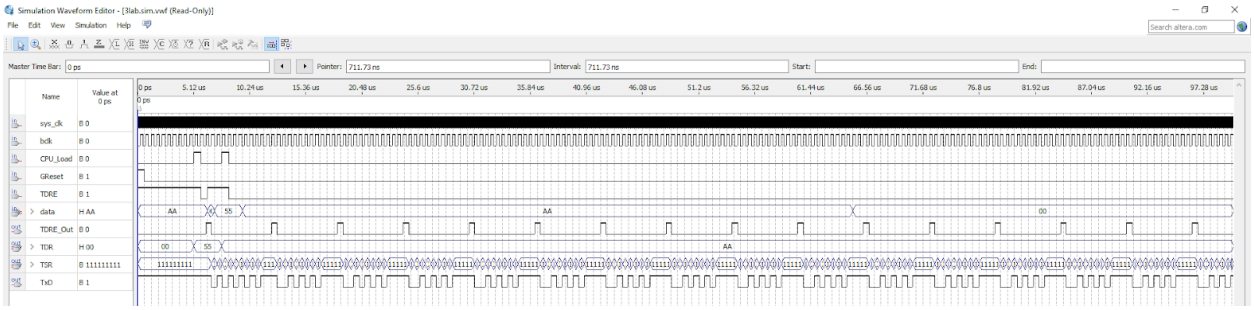
The results confirmed that the FSM interacted as expected, and they provided strong evidence that the design was correct before moving on to hardware testing.

Simulation 4 | Top Level

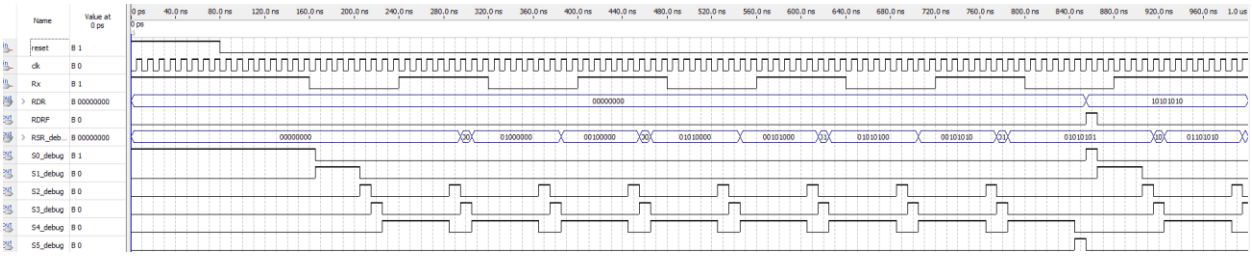


Simulation 6 | UART System

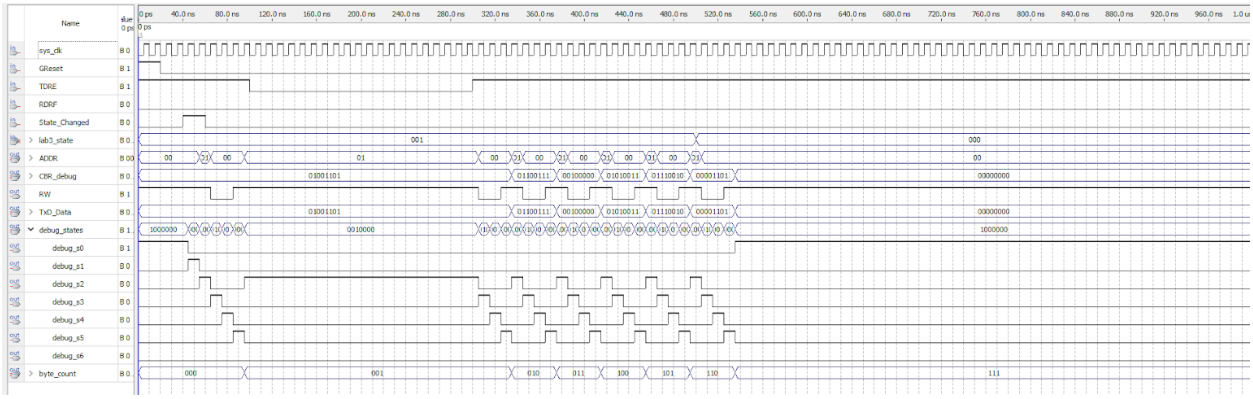
Simulation 6 | Transmitter



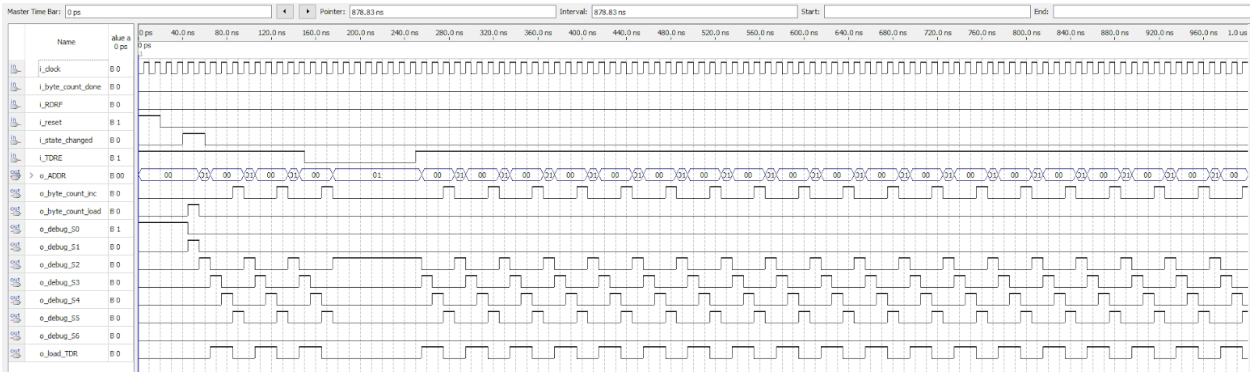
Simulation 5| Reciever



Simulation 6 | UART Interface FSM



Simulation 6 | UART FSM Controller

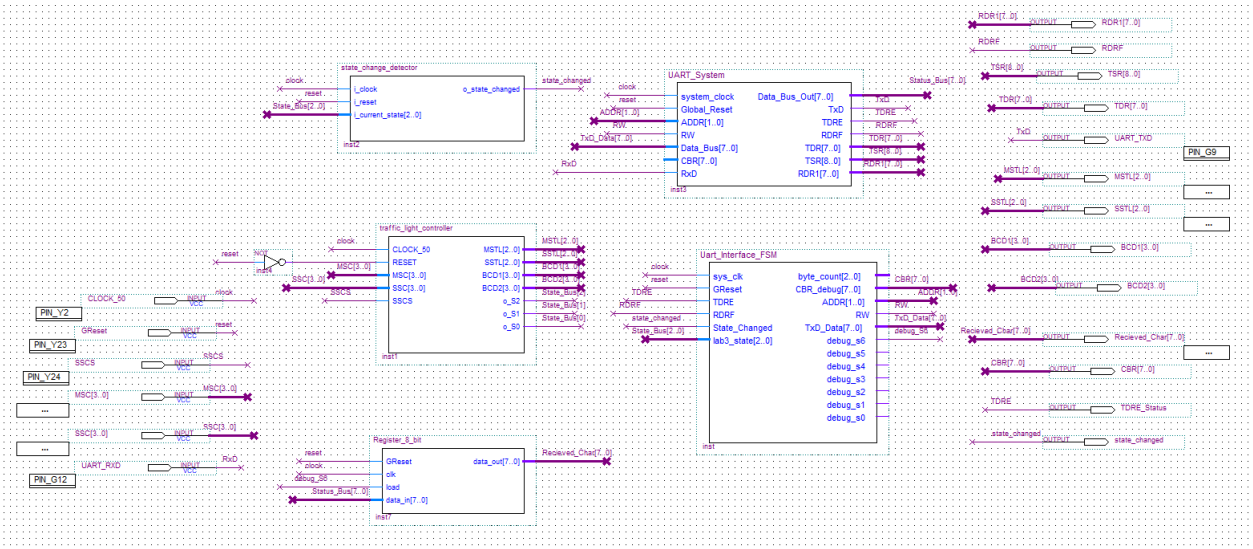


As shown by the top-level entity, our Controller is acting as expected. We transition between all states, as if it were a realistic traffic light, with proper reset functionality and sensor control logic.

As confirmed by the **experimental results**, (the live demo), our theoretical results are confirmed as we received 5/5 marks.

Part 4 | Code

Top Level bdf



For the purpose of this UART Project, our group will not showcase the Traffic Light Logic/VHDL files as it is covered in lab 3 and does not directly relate to the UART functionality. What matters from the Traffic Light Controller are the current states, encoded in the state variables S2,S1,S0.

State Change Detector

```vhdl
-- state_change_detector.vhd
-- Detects when the traffic light state changes
-- Compares current state with previous state (stored in register)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY state_change_detector IS
    PORT(
        i_clock         : IN  STD_LOGIC;
        i_reset         : IN  STD_LOGIC;
        i_current_state : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        o_state_changed : OUT STD_LOGIC
    );
END state_change_detector;

ARCHITECTURE structural OF state_change_detector IS

    COMPONENT register_3bit
        PORT(
            i_clock  : IN  STD_LOGIC;
            i_reset  : IN  STD_LOGIC;
            i_load   : IN  STD_LOGIC;
            i_data   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            o_data   : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT comparator_3bit
        PORT(
            i_A         : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            i_B         : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            o_not_equal : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL s_prev_state : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

    -- Store previous state (always enabled)
    prev_state_reg: register_3bit
        PORT MAP(
            i_clock  => i_clock,
            i_reset  => i_reset,
            i_load   => '1',  -- Always store current state
            i_data   => i_current_state,
            o_data   => s_prev_state
        );

    -- Compare current with previous
    state_comparator: comparator_3bit
        PORT MAP(
            i_A         => i_current_state,
            i_B         => s_prev_state,
            o_not_equal => o_state_changed
        );

END structural;
```

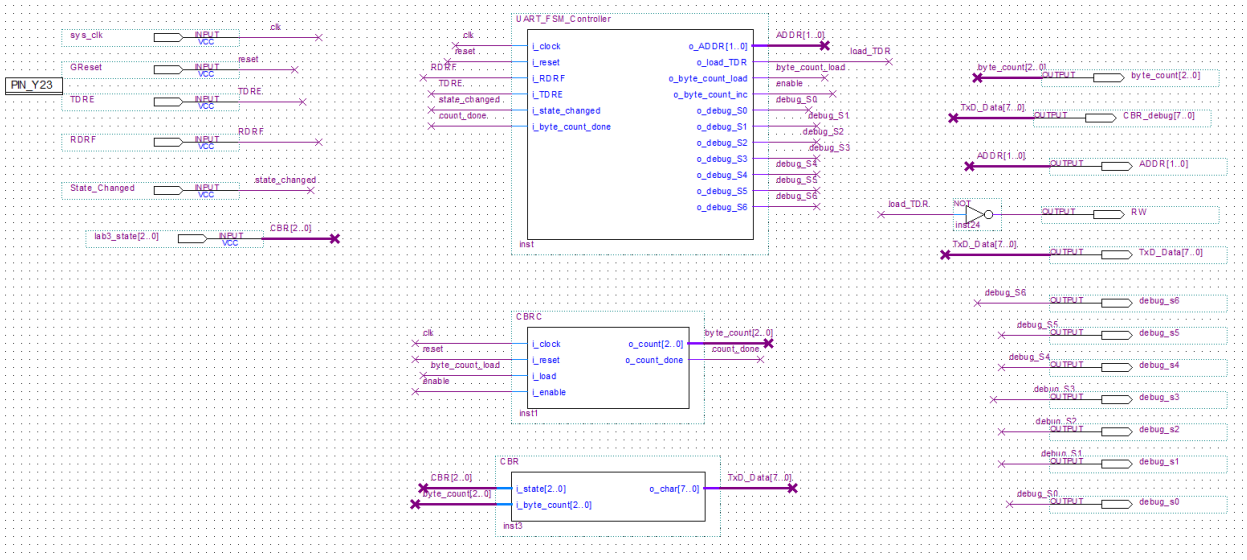
State Register

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY state_register IS
    PORT(
        i_clock      : IN  STD_LOGIC;
        i_reset      : IN  STD_LOGIC;
        i_S0_next    : IN  STD_LOGIC;
        i_S1_next    : IN  STD_LOGIC;
        i_S2_next    : IN  STD_LOGIC;
        i_S3_next    : IN  STD_LOGIC;
        i_S4_next    : IN  STD_LOGIC;
        i_S5_next    : IN  STD_LOGIC;
        i_S6_next    : IN  STD_LOGIC; -- NEW STATE 6 (Receive)
        o_S0         : OUT STD_LOGIC;
        o_S1         : OUT STD_LOGIC;
        o_S2         : OUT STD_LOGIC;
        o_S3         : OUT STD_LOGIC;
        o_S4         : OUT STD_LOGIC;
        o_S5         : OUT STD_LOGIC;
        o_S6         : OUT STD_LOGIC  -- NEW STATE 6
    );
END state_register;

ARCHITECTURE structural OF state_register IS
    COMPONENT dFF_set1
        PORT(D, clk, reset : IN STD_LOGIC; Q : OUT STD_LOGIC);
    END COMPONENT;

    COMPONENT dFF_reset0
        PORT(D, clk, reset : IN STD_LOGIC; Q : OUT STD_LOGIC);
    END COMPONENT;

BEGIN
    -- S0: Initial state (1 on reset)
    dff_s0: dFF_set1 PORT MAP(D => i_S0_next, clk => i_clock, reset => i_reset, Q => o_S0);

    -- S1-S6: Reset to 0
    dff_s1: dFF_reset0 PORT MAP(D => i_S1_next, clk => i_clock, reset => i_reset, Q => o_S1);
    dff_s2: dFF_reset0 PORT MAP(D => i_S2_next, clk => i_clock, reset => i_reset, Q => o_S2);
    dff_s3: dFF_reset0 PORT MAP(D => i_S3_next, clk => i_clock, reset => i_reset, Q => o_S3);
    dff_s4: dFF_reset0 PORT MAP(D => i_S4_next, clk => i_clock, reset => i_reset, Q => o_S4);
    dff_s5: dFF_reset0 PORT MAP(D => i_S5_next, clk => i_clock, reset => i_reset, Q => o_S5);
    dff_s6: dFF_reset0 PORT MAP(D => i_S6_next, clk => i_clock, reset => i_reset, Q => o_S6);
END structural;
```

8 Bit Register

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Register_8_bit is
    Port (
        GReset   : in  STD_LOGIC;                    -- Global reset (active high)
        clk      : in  STD_LOGIC;                    -- Clock signal
        load     : in  STD_LOGIC;                    -- Load enable signal
        data_in  : in  STD_LOGIC_VECTOR(7 downto 0); -- 8-bit input data
        data_out : out STD_LOGIC_VECTOR(7 downto 0)  -- 8-bit output data
    );
end Register_8_bit;

architecture structural of Register_8_bit is
    -- Component declaration for D flip-flop
    component DFF_reset0 is
        port(
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

    -- Internal signals
    signal reg       : STD_LOGIC_VECTOR(7 downto 0);
    signal reg_d     : STD_LOGIC_VECTOR(7 downto 0);
    signal load_n    : STD_LOGIC;

begin
    -- Output assignment
    data_out <= reg;

    -- Generate NOT gate for load signal
    load_n <= not load;

    -- Generate 8 flip-flops with 2-input mux logic
    gen_register: for i in 0 to 7 generate
        -- Mux logic: reg_d(i) = (load AND data_in(i)) OR (load_n AND reg(i))
        -- If load=1, select data_in(i); if load=0, select reg(i) (hold)
        reg_d(i) <= (load and data_in(i)) or (load_n and reg(i));

        -- Instantiate flip-flop
        ff_inst: DFF_reset0
            port map(
                D     => reg_d(i),
                clk   => clk,
                reset => GReset,
                Q     => reg(i)
            );
    end generate;

end structural;
```

UART Interface FSM BDF



UART_FSM_Controller

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY UART_FSM_Controller IS
    PORT(
        i_clock            : IN  STD_LOGIC;
        i_reset            : IN  STD_LOGIC;
        i_RDRF             : IN  STD_LOGIC;
        i_TDRE             : IN  STD_LOGIC;
        i_state_changed    : IN  STD_LOGIC;
        i_byte_count_done  : IN  STD_LOGIC;
        o_ADDR             : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        o_load_TDR         : OUT STD_LOGIC;
        o_byte_count_load  : OUT STD_LOGIC;
        o_byte_count_inc   : OUT STD_LOGIC;
        -- Debug
        o_debug_S0, o_debug_S1, o_debug_S2 : OUT STD_LOGIC;
        o_debug_S3, o_debug_S4, o_debug_S5, o_debug_S6 : OUT STD_LOGIC
    );
END UART_FSM_Controller;

ARCHITECTURE structural OF UART_FSM_Controller IS
    COMPONENT state_register
        PORT(
            i_clock, i_reset : IN STD_LOGIC;
            i_S0_next, i_S1_next, i_S2_next, i_S3_next, i_S4_next, i_S5_next, i_S6_next : IN
 STD_LOGIC;
            o_S0, o_S1, o_S2, o_S3, o_S4, o_S5, o_S6 : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT next_state_logic
        PORT(
            i_S0, i_S1, i_S2, i_S3, i_S4, i_S5, i_S6 : IN STD_LOGIC;
            i_RDRF, i_TDRE, i_state_changed, i_byte_count_done : IN STD_LOGIC;
            o_S0_next, o_S1_next, o_S2_next, o_S3_next, o_S4_next, o_S5_next, o_S6_next :
OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT output_logic
        PORT(
            i_S0, i_S1, i_S2, i_S3, i_S4, i_S5, i_S6 : IN STD_LOGIC;
            o_ADDR : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
            o_load_TDR, o_byte_count_load, o_byte_count_inc : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL s0, s1, s2, s3, s4, s5, s6 : STD_LOGIC;
    SIGNAL n0, n1, n2, n3, n4, n5, n6 : STD_LOGIC;

BEGIN
    U_Reg : state_register PORT MAP(
        i_clock => i_clock, i_reset => i_reset,
        i_S0_next => n0, i_S1_next => n1, i_S2_next => n2, i_S3_next => n3, i_S4_next => n4,
 i_S5_next => n5, i_S6_next => n6,
        o_S0 => s0, o_S1 => s1, o_S2 => s2, o_S3 => s3, o_S4 => s4, o_S5 => s5, o_S6 => s6
    );

    U_Next : next_state_logic PORT MAP(
        i_S0 => s0, i_S1 => s1, i_S2 => s2, i_S3 => s3, i_S4 => s4, i_S5 => s5, i_S6 => s6,
        i_RDRF => i_RDRF, i_TDRE => i_TDRE, i_state_changed => i_state_changed,

i_byte_count_done => i_byte_count_done,
        o_S0_next => n0, o_S1_next => n1, o_S2_next => n2, o_S3_next => n3, o_S4_next => n4,
 o_S5_next => n5, o_S6_next => n6
    );

    U_Out : output_logic PORT MAP(
        i_S0 => s0, i_S1 => s1, i_S2 => s2, i_S3 => s3, i_S4 => s4, i_S5 => s5, i_S6 => s6,
        o_ADDR => o_ADDR, o_load_TDR => o_load_TDR, o_byte_count_load => o_byte_count_load,
o_byte_count_inc => o_byte_count_inc
    );

    o_debug_S0 <= s0; o_debug_S1 <= s1; o_debug_S2 <= s2; o_debug_S3 <= s3;
    o_debug_S4 <= s4; o_debug_S5 <= s5; o_debug_S6 <= s6;

END structural;
```

SHR 9bit Register

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SHR_Register_9_Bit is
    port(
        ShR       : in  std_logic;
        load      : in  std_logic;
        GReset    : in  std_logic;
        clk       : in  std_logic;
        data      : in  std_logic_vector(7 downto 0);
        data_out  : out std_logic;
        TSR_debug: out std_logic_vector(8 downto 0)
    );
end entity;

architecture structural of SHR_Register_9_Bit is
    -- Component declaration for D flip-flop with active-high async reset to '0'
    component DFF_reset0 is
        port(
            clk   : in  std_logic;
            reset : in  std_logic;
            d     : in  std_logic;
            q     : out std_logic
        );
    end component;

    -- Internal signals
    signal reg_q       : std_logic_vector(8 downto 0);
    signal reg_d       : std_logic_vector(8 downto 0);
    signal load_n      : std_logic;
    signal ShR_n       : std_logic;

    signal load_select : std_logic_vector(8 downto 0);
    signal shift_select: std_logic_vector(8 downto 0);
    signal hold_select : std_logic_vector(8 downto 0);
    signal shift_in    : std_logic_vector(8 downto 0);

begin
    -- Output assignments
    data_out  <= reg_q(0);
    TSR_debug <= reg_q;

    -- Generate NOT gates for control signals
    load_n <= not load;
    ShR_n  <= not ShR;

    -- Shift data preparation (for bit i, shift_in is bit i+1, except MSB gets '1')
    shift_in(8) <= '1';
    shift_in(7 downto 0) <= reg_q(8 downto 1);

    -- Generate multiplexer logic for each bit
    gen_mux: for i in 0 to 8 generate
        -- 3-input priority mux: load > shift > hold
        -- load_select(i)  = load
        -- shift_select(i) = (not load) AND ShR
        -- hold_select(i)  = (not load) AND (not ShR)

        load_select(i)  <= load;
        shift_select(i) <= load_n and ShR;
        hold_select(i)  <= load_n and ShR_n;


        -- Mux output: (load_sel AND load_data) OR (shift_sel AND shift_data) OR (hold_sel AND hold_data)
        gen_bit_low: if i = 0 generate
            reg_d(i) <= (load_select(i) and '0') or
                        (shift_select(i) and shift_in(i)) or
                        (hold_select(i) and reg_q(i));
        end generate;

        gen_bit_high: if i >= 1 and i <= 7 generate
            reg_d(i) <= (load_select(i) and data(i-1)) or
                        (shift_select(i) and shift_in(i)) or
                        (hold_select(i) and reg_q(i));
        end generate;

        gen_bit_msb: if i = 8 generate
            reg_d(i) <= (load_select(i) and data(7)) or
                        (shift_select(i) and shift_in(i)) or
                        (hold_select(i) and reg_q(i));
        end generate;

        -- Instantiate flip-flop
        -- Note: DFF_reset0 resets to '0', but we need reset to '1'
        -- You may need to use DFF_reset1 or add inverter logic
        ff_inst: DFF_reset0
            port map(
                clk   => clk,
                reset => GReset,
                d     => reg_d(i),
                q     => reg_q(i)
            );
    end generate;

end architecture structural;
```

RSR 8bit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RSR_8bit is
    Port (
        clk      : in  STD_LOGIC;
        reset    : in  STD_LOGIC;
        RxD      : in  STD_LOGIC;
        RSR_load : in  STD_LOGIC;
        shift_R  : in  STD_LOGIC;        -- Back to shift_R
        RSR_out  : out STD_LOGIC_VECTOR(7 downto 0)
    );
end RSR_8bit;

architecture Structural of RSR_8bit is
    component dFF_reset0
        Port (
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

    signal Q : STD_LOGIC_VECTOR(7 downto 0);
    signal D : STD_LOGIC_VECTOR(7 downto 0);

begin
    -- Generate 8 D flip-flops
    gen_ffs: for i in 0 to 7 generate
        dff_i: dFF_reset0 port map (
            D     => D(i),
            clk   => clk,
            reset => reset,
            Q     => Q(i)
        );
    end generate;

    -- SHIFT RIGHT: Sample into bit 7, shift down toward bit 0

    -- Bit 7 (MSB): Load from RxD or shift in 0
    D(7) <= RxD when RSR_load = '1' else
            '0' when shift_R = '1' else
            Q(7);

    -- Bits 1-6: Shift in from next higher bit
    D(6) <= Q(7) when shift_R = '1' else Q(6);
    D(5) <= Q(6) when shift_R = '1' else Q(5);
    D(4) <= Q(5) when shift_R = '1' else Q(4);
    D(3) <= Q(4) when shift_R = '1' else Q(3);
    D(2) <= Q(3) when shift_R = '1' else Q(2);
    D(1) <= Q(2) when shift_R = '1' else Q(1);

    -- Bit 0 (LSB): Shift in from bit 1 or hold
    D(0) <= Q(1) when shift_R = '1' else Q(0);

    RSR_out <= Q;

end Structural;
```

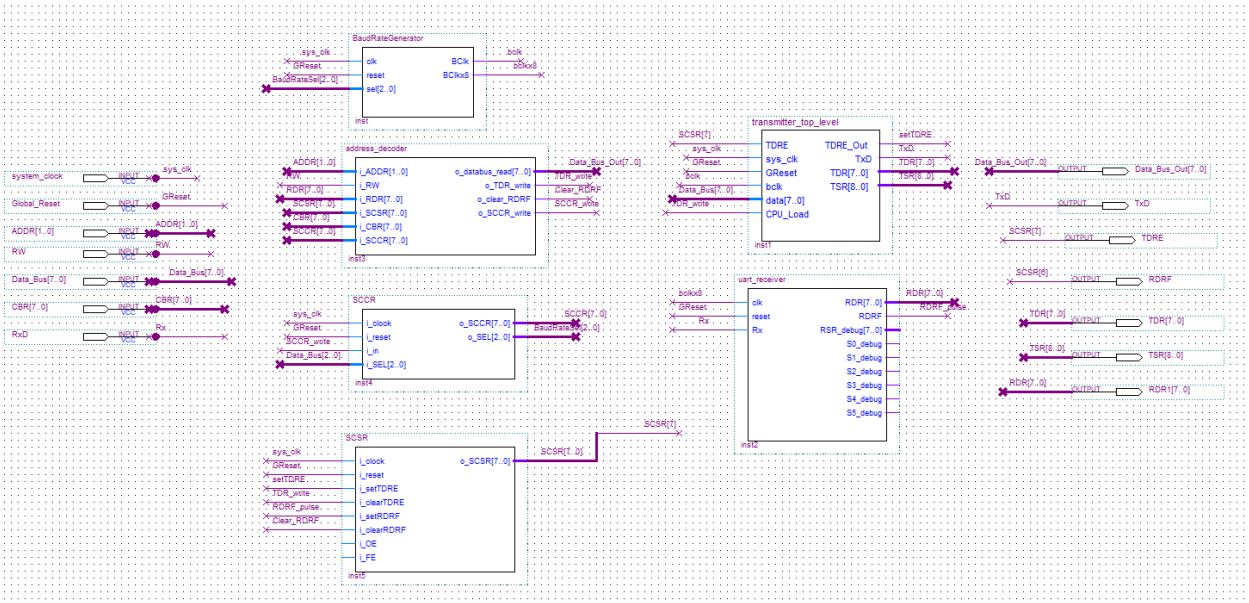
CBRC

```vhdl
-- CBRC.vhd (Character Byte Register Counter)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY CBRC IS
    PORT(
        i_clock         : IN  STD_LOGIC;
        i_reset         : IN  STD_LOGIC;
        i_load          : IN  STD_LOGIC;  -- Load initial value (0)
        i_enable        : IN  STD_LOGIC;  -- Enable increment
        o_count         : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);  -- Current count
        o_count_done    : OUT STD_LOGIC   -- '1' when count = 6 (done)
    );
END CBRC;

ARCHITECTURE structural OF CBRC IS
    -- Component declaration for up_counter
    COMPONENT up_counter
        GENERIC(
            n : INTEGER := 3
        );
        PORT(
            i_clock  : IN  STD_LOGIC;
            i_reset  : IN  STD_LOGIC;
            i_load   : IN  STD_LOGIC;
            i_enable : IN  STD_LOGIC;
            i_count  : IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
            o_count  : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL s_count : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN
    -- Instantiate the 3-bit up counter
    byte_counter: up_counter
        GENERIC MAP(
            n => 3  -- 3-bit counter (0-7 range, we use 0-5)
        )
        PORT MAP(
            i_clock  => i_clock,
            i_reset  => i_reset,
            i_load   => i_load,
            i_enable => i_enable,
            i_count  => "000",      -- Load value = 0 (start from beginning)
            o_count  => s_count
        );

    -- Output the count
    o_count <= s_count;

    -- Generate done flag
    -- When count = "110" (6), all 6 bytes have been sent (0,1,2,3,4,5)
    o_count_done <= '1' WHEN s_count = "110" ELSE '0';

END structural;
```

CBR

```
-- CBR.vhd (Character Byte Register)
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY CBR IS
    PORT(
        i_state      : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);  -- Lab 3 state (3 bits)
        i_byte_count : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);  -- Which byte to output (0-5)
        o_char       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)   -- ASCII character output
    );
END CBR;

ARCHITECTURE structural OF CBR IS
    -- Component declarations
    COMPONENT mux8to1_8bit
        PORT(
            i_sel    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            i_d0     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d1     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d2     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d3     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d4     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d5     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d6     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d7     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            o_q      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT mux6to1_8bit
        PORT(
            i_sel    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            i_d0     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d1     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d2     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d3     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d4     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d5     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d6     : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            o_q      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
    END COMPONENT;

    -- Fixed byte constants
    CONSTANT BYTE_0_M        : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"4D";  -- 'M'
    CONSTANT BYTE_2_SPACE    : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"20";  -- ' '
    CONSTANT BYTE_3_S        : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"53";  -- 'S'
    CONSTANT BYTE_5_CR       : STD_LOGIC_VECTOR(7 DOWNTO 0) := x"0D";  -- '\r'

    -- Variable byte signals (outputs from muxes)
    SIGNAL s_byte_1_variable : STD_LOGIC_VECTOR(7 DOWNTO 0);  -- g/y/r
    SIGNAL s_byte_4_variable : STD_LOGIC_VECTOR(7 DOWNTO 0);  -- r/r/g/y

    -- Mux select signals from state
    SIGNAL sel : STD_LOGIC_VECTOR(2 DOWNTO 0);  -- For mux


BEGIN
    -- Extract mux selects from state bits
    -- Based on your design: y1y0 and y2y0
    sel <= i_state(2 DOWNTO 0);  -- Use bits [2:0] for first mux


    -- First 4:1 MUX for Byte 1 (second character: g/y/r)
    -- y1y0: 00='R'(72), 01='G'(67), 10='Y'(79), 11='R'(72)
    mux_byte1: mux8to1_8bit
        PORT MAP(
            i_sel => sel,
            i_d0  => x"72",  -- 'r' for 000
            i_d1  => x"67",  -- 'g' for 001
            i_d2  => x"79",  -- 'y' for 010
            i_d3  => x"72",  -- 'r' for 011
            i_d4  => x"72",  -- 'r' for 100
            i_d5  => x"72",  -- 'r' for 100
            i_d6  => x"00",
            i_d7  => x"00",
            o_q   => s_byte_1_variable
        );

    -- Second 4:1 MUX for Byte 4 (fifth character: r/r/g/y)
    -- y2y0: 00='R'(72), 01='R'(72), 10='G'(67), 11='Y'(79)
    mux_byte4: mux8to1_8bit
        PORT MAP(
            i_sel => sel,
            i_d0  => x"72",  -- 'r' for 000
            i_d1  => x"72",  -- 'r' for 001
            i_d2  => x"72",  -- 'r' for 010
            i_d3  => x"72",  -- 'g' for 011
            i_d4  => x"67",  -- 'y' for 100
            i_d5  => x"79",
            i_d6  => x"00",
            i_d7  => x"00",
            o_q   => s_byte_4_variable
        );

    -- Final 6:1 MUX selects which byte based on byte_count
    mux_byte_select: mux6to1_8bit
        PORT MAP(
            i_sel => i_byte_count,
            i_d0  => BYTE_0_M,           -- Byte 0: 'M'
            i_d1  => BYTE_0_M,           -- Byte 0: 'M'
            i_d2  => s_byte_1_variable,  -- Byte 1: variable (g/y/r)
            i_d3  => BYTE_2_SPACE,       -- Byte 2: ' '
            i_d4  => BYTE_3_S,           -- Byte 3: 'S'
            i_d5  => s_byte_4_variable,  -- Byte 4: variable (r/g/y)
            i_d6  => BYTE_5_CR,          -- Byte 5: '\r'
            o_q   => o_char
        );

END structural;
```

Uart System


BaudRateGenerator
sys_clk
GReset
BaudRateSel[2..0]
clk
reset
sel[2..0]
BClk
BClkx8
bclk
bclkx8
inst
address_decoder
ADDR[1..0]
RW
RDR[7..0]
SCSR[1..0]
CBR[7..0]
SCCR[7..0]
i_ADDR[1..0]
i_RW
i_RDR[7..0]
i_SCSR[7..0]
i_CBR[7..0]
i_SCCR[7..0]
o_databus_read[7..0]
o_TDR_write
o_clear_RDRF
o_SCCR_write
Data_Bus_Out[7..0]
TDR_write
Clear_RDRF
SCCR_write
inst3
SCCR
sys_clk
GReset
SCCR_write
Data_Bus[2..0]
i_clock
i_reset
i_in
i_SEL[2..0]
o_SCCR[7..0]
o_SEL[2..0]
SCCR[7..0]
BaudRateSel[2..0]
inst4
SCSR
sys_clk
GReset
setTDRE
TDR_write
RDRF_pulse
Clear_RDRF
i_clock
i_reset
i_setTDRE
i_clearTDRE
i_setRDRF
i_clearRDRF
i_OE
i_FE
o_SCSR[7..0]
SCSR[7..0]
SCSR[7]
inst5
transmitter_top_level
SCSR[1]
sys_clk
GReset
bclk
Data_Bus[7..0]
TDR_write
TDRE
sys_clk
GReset
bclk
data[7..0]
CPU_Load
TDRE_Out
TXD
TDR[7..0]
TSR[8..0]
setTDRE
TxD
TDR[7..0]
TSR[8..0]
inst1
uart_receiver
bclkx8
GReset
Rx
clk
reset
Rx
RDR[7..0]
RDRF
RSR_debug[7..0]
S0_debug
S1_debug
S2_debug
S3_debug
S4_debug
S5_debug
RDR[7..0]
RDRF_pulse
inst2
Data_Bus_Out[7..0]
OUTPUT
Data_Bus_Out[7..0]
TxD
OUTPUT
TxD
SCSR[7]
OUTPUT
TDRE
SCSR[6]
OUTPUT
RDRF
TDR[7..0]
OUTPUT
TDR[7..0]
TSR[8..0]
OUTPUT
TSR[8..0]
RDR[7..0]
OUTPUT
RDR1[7..0]
system_clock
Global_Reset
ADDR[1..0]
RW
Data_Bus[7..0]
CBR[7..0]
RxD
INPUT
VCC
sys_clk
GReset
ADDR[1..0]
RW
Data_Bus[7..0]
CBR[7..0]
Rx

Baud Rate Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity BaudRateGenerator is
    port(
        clk    : in  std_logic; -- system clock
        reset  : in  std_logic;
        sel    : in  std_logic_vector(2 downto 0);
        BClk   : out std_logic;
        BClkx8 : out std_logic
    );
end BaudRateGenerator;

architecture Structural of BaudRateGenerator is

    signal clk_41_out  : std_logic;
    signal count_lines : std_logic_vector(7 downto 0);
    signal mux_out     : std_logic;

    -- Components
    component clock_div_41
        Port ( clk_in, reset : in STD_LOGIC; clk_out : out STD_LOGIC );
    end component;

    component counter_8_bit
        Port ( clk, reset : in STD_LOGIC; q : out STD_LOGIC_VECTOR(7 downto 0) );
    end component;

    component mux_8to1
        Port ( sel : in STD_LOGIC_VECTOR(2 downto 0); data : in STD_LOGIC_VECTOR(7 downto 0
); y : out STD_LOGIC );
    end component;

    component clock_div_8
        Port ( clk_in, reset : in STD_LOGIC; clk_out : out STD_LOGIC );
    end component;

begin
    -- 1. Divide by 41
    U_Div41 : clock_div_41
        port map(
            clk_in  => clk,
            reset   => reset,
            clk_out => clk_41_out
        );

    -- 2. The 8-bit Counter (Source of 8 frequencies)
    U_Counter : counter_8_bit
        port map (
            clk   => clk_41_out,
            reset => reset,
            q     => count_lines
        );

    -- 3. The Multiplexer (Selects the baud rate)
    U_Mux : mux_8to1
        port map (
            sel  => sel,
            data => count_lines,
            y    => mux_out
        );


     -- 4. The Final Divider (BClkx8 -> BClk)
     U_Div8 : clock_div_8
         port map (
             clk_in  => mux_out,
             reset   => reset,
             clk_out => BClk
         );

     BClkx8 <= mux_out;


 end Structural;
```

Address Decider

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY address_decoder IS
    PORT(
        i_ADDR         : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
        i_RW           : IN  STD_LOGIC; -- 1=Read, 0=Write

        -- Inputs for READ MUX
        i_RDR          : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_SCSR         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_CBR          : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        i_SCCR         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);

        -- Outputs
        o_databus_read : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_TDR_write    : OUT STD_LOGIC; -- Write to TDR

        -- *** NEW OUTPUTS ***
        o_clear_RDRF   : OUT STD_LOGIC; -- Reset RDRF status
        o_SCCR_write   : OUT STD_LOGIC  -- Write enable for SCCR
    );
END address_decoder;

ARCHITECTURE structural OF address_decoder IS
    COMPONENT mux4to1_8bit
        PORT(
            i_sel   : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            i_d0    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d1    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d2    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            i_d3    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            o_q     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
    END COMPONENT;

BEGIN
    -- READ MUX
    read_mux: mux4to1_8bit
        PORT MAP(
            i_sel => i_ADDR,
            i_d0  => i_RDR,   -- 00
            i_d1  => i_SCSR,  -- 01
            i_d2  => i_CBR,   -- 10
            i_d3  => i_SCCR,  -- 11
            o_q   => o_databus_read
        );

    -- WRITE ENABLE for TDR (Addr 00, Write)
    o_TDR_write <= '1' WHEN (i_ADDR = "00" AND i_RW = '0') ELSE '0';

    -- CLEAR ENABLE for RDRF (Addr 00, Read)
    -- This tells SCSR to set RDRF=0 because CPU read the data
    o_clear_RDRF <= '1' WHEN (i_ADDR = "00" AND i_RW = '1') ELSE '0';

    -- WRITE ENABLE for SCCR (Addr 11, Write)
    -- This tells SCCR to load the new config
    o_SCCR_write <= '1' WHEN (i_ADDR = "11" AND i_RW = '0') ELSE '0';

END structural;
```

SCCR

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY SCCR IS
    PORT(
        i_clock : IN  STD_LOGIC;
        i_reset : IN  STD_LOGIC;
        i_in    : IN  STD_LOGIC; -- Acts as Load / Write Enable
        i_SEL   : IN  STD_LOGIC; -- Acts as ACTIVE (1 DOWNTO 0)
        o_SCCR  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_SEL   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END SCCR;

ARCHITECTURE structural OF SCCR IS
    COMPONENT dFF_reset0
        PORT(
            D     : IN  STD_LOGIC;
            clk   : IN  STD_LOGIC;
            reset : IN  STD_LOGIC;
            Q     : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL s_SCCR : STD_LOGIC_VECTOR(7 DOWNTO 0);

    -- Intermediate signals for mux logic
    SIGNAL s_d2, s_d1, s_d0 : STD_LOGIC;

    -- NOT of control signals
    SIGNAL i_in_n  : STD_LOGIC;
    SIGNAL i_SEL_n : STD_LOGIC;

    -- Select signals for each mux path
    SIGNAL load_and_sel    : STD_LOGIC;  -- i_in AND i_SEL
    SIGNAL load_and_sel_n  : STD_LOGIC;  -- i_in AND (NOT i_SEL)
    SIGNAL hold            : STD_LOGIC;  -- NOT i_in

BEGIN

    -- Generate NOT gates for control signals
    i_in_n  <= not i_in;
    i_SEL_n <= not i_SEL;

    -- Generate select signals for multiplexers
    load_and_sel   <= i_in and i_SEL;       -- Load from s_SCCR
    load_and_sel_n <= i_in and i_SEL_n;     -- Load '1'
    hold           <= i_in_n;                -- Hold current value

    -- BIT 7-3: Unused (tied to 0)
    -- These bits always output 0
    s_SCCR(7) <= '0';
    s_SCCR(6) <= '0';
    s_SCCR(5) <= '0';
    s_SCCR(4) <= '0';
    s_SCCR(3) <= '0';

    -----------------------------------------------------------------
    -- BIT 2 Logic
    -- If load(i_in) is 1, take Input(i_SEL). Else, keep Current(s_SCCR).
    -----------------------------------------------------------------
    -- s_d2 = (load_and_sel AND s_SCCR(2)) OR (load_and_sel_n AND '1') OR (hold AND
 s_SCCR(2))
    -- Simplified: s_d2 = (load_and_sel AND s_SCCR(2)) OR load_and_sel_n OR (hold AND
 s_SCCR(2))
    -- Further: s_d2 = ((load_and_sel OR hold) AND s_SCCR(2)) OR load_and_sel_n

    s_d2 <= (load_and_sel and s_SCCR(2)) or load_and_sel_n or (hold and s_SCCR(2));

    dff_bit2: dFF_reset0
        PORT MAP(
            D     => s_d2,
            clk   => i_clock,
            reset => i_reset,
            Q     => s_SCCR(2)
        );

    -----------------------------------------------------------------
    -- BIT 1 Logic
    -----------------------------------------------------------------
    s_d1 <= (load_and_sel and s_SCCR(1)) or load_and_sel_n or (hold and s_SCCR(1));

    dff_bit1: dFF_reset0
        PORT MAP(
            D     => s_d1,
            clk   => i_clock,
            reset => i_reset,
            Q     => s_SCCR(1)
        );

    -----------------------------------------------------------------
    -- BIT 0 Logic
    -----------------------------------------------------------------
    s_d0 <= (load_and_sel and s_SCCR(0)) or load_and_sel_n or (hold and s_SCCR(0));

    dff_bit0: dFF_reset0
        PORT MAP(
            D     => s_d0,
            clk   => i_clock,
            reset => i_reset,
            Q     => s_SCCR(0)
        );

    -----------------------------------------------------------------
    -- Output Assignments
    -----------------------------------------------------------------
    o_SCCR <= s_SCCR;
    o_SEL  <= s_SCCR(7 DOWNTO 0);

END structural;
```

SCSR

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY SCSR IS
    PORT(
        i_clock      : IN  STD_LOGIC;
        i_reset      : IN  STD_LOGIC;
        i_setTDRE    : IN  STD_LOGIC; -- Comes from Slow Transmitter
        i_clearTDRE  : IN  STD_LOGIC; -- Comes from Fast CPU
        i_setRDRF    : IN  STD_LOGIC; -- Comes from Slow Receiver
        i_clearRDRF  : IN  STD_LOGIC; -- Comes from Fast CPU
        i_OE         : IN  STD_LOGIC;
        i_FE         : IN  STD_LOGIC;
        o_SCSR       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END SCSR;

ARCHITECTURE structural OF SCSR IS
    COMPONENT dFF_reset0
        PORT(
            D     : IN  STD_LOGIC;
            clk   : IN  STD_LOGIC;
            reset : IN  STD_LOGIC;
            Q     : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT dFF_set1
        PORT(
            D     : IN  STD_LOGIC;
            clk   : IN  STD_LOGIC;
            reset : IN  STD_LOGIC;
            Q     : OUT STD_LOGIC
        );
    END COMPONENT;
    -- Inputs to the Status Bit Flip-Flops
    SIGNAL s_TDRE_input : STD_LOGIC;
    SIGNAL s_RDRF_input : STD_LOGIC;

    -- Current Status Bit Values
    SIGNAL s_TDRE       : STD_LOGIC;
    SIGNAL s_RDRF       : STD_LOGIC;

    -- Edge Detection Signals
    SIGNAL s_TDRE_delayed : STD_LOGIC;
    SIGNAL s_setTDRE_edge : STD_LOGIC; -- The "Clean" Pulse

    SIGNAL s_RDRF_delayed : STD_LOGIC;
    SIGNAL s_setRDRF_edge : STD_LOGIC; -- The "Clean" Pulse

    SIGNAL s_SCSR       : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
    ---------------------------------------------------------------------
    -- EDGE DETECTION LOGIC (The Fix)
    ---------------------------------------------------------------------
    -- 1. Delay the input by 1 clock cycle using a Flip-Flop
    dff_edge_tdre: dFF_reset0
        PORT MAP(D => i_setTDRE, clk => i_clock, reset => i_reset, Q => s_TDRE_delayed);

    -- 2. Create a pulse ONLY when Input is 1 AND Delayed Input is 0 (Rising Edge)
    s_setTDRE_edge <= i_setTDRE and (not s_TDRE_delayed);



    -- Repeat for Receiver (because it is also slower than SysClk)
    dff_edge_rdrf: dFF_reset0
        PORT MAP(D => i_setRDRF, clk => i_clock, reset => i_reset, Q => s_RDRF_delayed);

    s_setRDRF_edge <= i_setRDRF and (not s_RDRF_delayed);

    ---------------------------------------------------------------------
    -- STATUS BIT LOGIC
    ---------------------------------------------------------------------

    -- TDRE Logic (Bit 7)
    -- Use the EDGE signal (s_setTDRE_edge) instead of the raw input
    s_TDRE_input <= '0' WHEN i_clearTDRE = '1' ELSE
                    '1' WHEN s_setTDRE_edge = '1' ELSE  -- <--- FIXED HERE
                    s_TDRE;

    -- RDRF Logic (Bit 6)
    -- Use the EDGE signal (s_setRDRF_edge) instead of the raw input
    s_RDRF_input <= '0' WHEN i_clearRDRF = '1' ELSE
                    '1' WHEN s_setRDRF_edge = '1' ELSE  -- <--- FIXED HERE
                    s_RDRF;

    -- Bit 7 Flip-Flop
    dff_bit7: dFF_set1
        PORT MAP(D => s_TDRE_input, clk => i_clock, reset => i_reset, Q => s_TDRE);
    s_SCSR(7) <= s_TDRE;

    -- Bit 6 Flip-Flop
    dff_bit6: dFF_reset0
        PORT MAP(D => s_RDRF_input, clk => i_clock, reset => i_reset, Q => s_RDRF);
    s_SCSR(6) <= s_RDRF;

    -- Bits 5-2: Unused
    s_SCSR(5) <= '0';
    s_SCSR(4) <= '0';
    s_SCSR(3) <= '0';
    s_SCSR(2) <= '0';

    -- Bit 1: OE
    dff_bit1: dFF_reset0
        PORT MAP(D => i_OE, clk => i_clock, reset => i_reset, Q => s_SCSR(1));

    -- Bit 0: FE
    dff_bit0: dFF_reset0
        PORT MAP(D => i_FE, clk => i_clock, reset => i_reset, Q => s_SCSR(0));

    o_SCSR <= s_SCSR;

END structural;
```

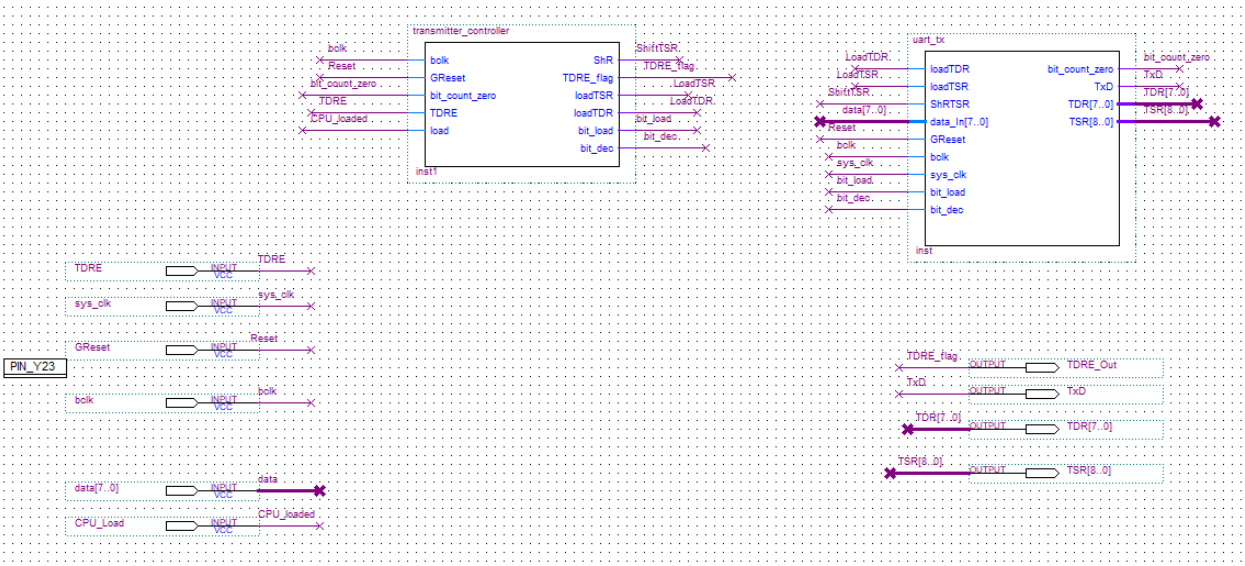
Uart Reciever

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity uart_receiver is
    Port (
        clk    : in  STD_LOGIC;
        reset : in  STD_LOGIC;
        Rx     : in  STD_LOGIC;
        RDR    : out STD_LOGIC_VECTOR(7 downto 0);
        RDRF   : out STD_LOGIC; -- Now a Pulse, not a latch

        -- Debug outputs
        RSR_debug : out STD_LOGIC_VECTOR(7 downto 0);
        S0_debug  : out STD_LOGIC;
        S1_debug  : out STD_LOGIC;
        S2_debug  : out STD_LOGIC;
        S3_debug  : out STD_LOGIC;
        S4_debug  : out STD_LOGIC;
        S5_debug  : out STD_LOGIC
    );
end uart_receiver;

architecture Structural of uart_receiver is

    component receiver_control_path
        Port (
            clk         : in  STD_LOGIC;
            reset       : in  STD_LOGIC;
            Rx          : in  STD_LOGIC;
            sample_zero : in  STD_LOGIC;
            bit_zero    : in  STD_LOGIC;
            new_zero    : in  STD_LOGIC;
            sample_load : out STD_LOGIC;
            sample_dec  : out STD_LOGIC;
            bit_load    : out STD_LOGIC;
            bit_dec     : out STD_LOGIC;
            new_load    : out STD_LOGIC;
            new_dec     : out STD_LOGIC;
            RSR_load    : out STD_LOGIC;
            shift_R     : out STD_LOGIC;
            load_RDR    : out STD_LOGIC;
            set_RDRF    : out STD_LOGIC;

            -- Debug: State outputs
            S0_out      : out STD_LOGIC;
            S1_out      : out STD_LOGIC;
            S2_out      : out STD_LOGIC;
            S3_out      : out STD_LOGIC;
            S4_out      : out STD_LOGIC;
            S5_out      : out STD_LOGIC
        );
    end component;

    component down_counter
        generic (
            WIDTH      : integer;
            LOAD_VALUE : integer
        );
        Port (
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            load  : in  STD_LOGIC;
            dec   : in  STD_LOGIC;
            count : out STD_LOGIC_VECTOR(WIDTH-1 downto 0);
            zero  : out STD_LOGIC
        );
    end component;

    component RSR_8bit
        Port (
            clk      : in  STD_LOGIC;
            reset    : in  STD_LOGIC;
            RxD      : in  STD_LOGIC;
            RSR_load : in  STD_LOGIC;
            shift_R  : in  STD_LOGIC;
            RSR_out  : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    component RDR_8bit
        Port (
            clk      : in  STD_LOGIC;
            reset    : in  STD_LOGIC;
            load_RDR : in  STD_LOGIC;
            RSR_in   : in  STD_LOGIC_VECTOR(7 downto 0);
            RDR_out  : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    -- Internal Signals
    signal sample_zero : STD_LOGIC;
    signal bit_zero    : STD_LOGIC;
    signal new_zero    : STD_LOGIC;

    signal sample_load : STD_LOGIC;
    signal sample_dec  : STD_LOGIC;
    signal bit_load    : STD_LOGIC;
    signal bit_dec     : STD_LOGIC;
    signal new_load    : STD_LOGIC;
    signal new_dec     : STD_LOGIC;
    signal RSR_load    : STD_LOGIC;
    signal shift_R     : STD_LOGIC;
    signal load_RDR    : STD_LOGIC;
    signal set_RDRF    : STD_LOGIC;

    signal RSR_to_RDR  : STD_LOGIC_VECTOR(7 downto 0);
    signal sample_count : STD_LOGIC_VECTOR(3 downto 0);
    signal bit_count    : STD_LOGIC_VECTOR(3 downto 0);
    signal new_count    : STD_LOGIC_VECTOR(2 downto 0);

begin

    control: receiver_control_path port map (
        clk         => clk,
        reset       => reset,
        Rx          => Rx,
        sample_zero => sample_zero,
        bit_zero    => bit_zero,
        new_zero    => new_zero,
        sample_load => sample_load,
        sample_dec  => sample_dec,
        bit_load    => bit_load,
        bit_dec     => bit_dec,
        new_load    => new_load,
        new_dec     => new_dec,
        RSR_load    => RSR_load,
        shift_R     => shift_R,
        load_RDR    => load_RDR,
        set_RDRF    => set_RDRF,

        S0_out      => S0_debug,
        S1_out      => S1_debug,
        S2_out      => S2_debug,
        S3_out      => S3_debug,
        S4_out      => S4_debug,
        S5_out      => S5_debug
    );

    sample_counter: down_counter
        generic map (WIDTH => 4, LOAD_VALUE => 3)
        port map (
            clk => clk, reset => reset,
            load => sample_load, dec => sample_dec,
            count => sample_count, zero => sample_zero
        );

    bit_counter: down_counter
        generic map (WIDTH => 4, LOAD_VALUE => 8)
        port map (
            clk => clk, reset => reset,
            load => bit_load, dec => bit_dec,
            count => bit_count, zero => bit_zero
        );

    new_counter: down_counter
        generic map (WIDTH => 3, LOAD_VALUE => 6)
        port map (
            clk => clk, reset => reset,
            load => new_load, dec => new_dec,
            count => new_count, zero => new_zero
        );

    shift_register: RSR_8bit port map (
        clk      => clk,
        reset    => reset,
        RxD      => Rx,
        RSR_load => RSR_load,
        shift_R  => shift_R,
        RSR_out  => RSR_to_RDR
    );

    receive_data_reg: RDR_8bit port map (
        clk => clk, reset => reset,
        load_RDR => load_RDR,
        RSR_in => RSR_to_RDR, RDR_out => RDR
    );

    -- *** FIX: Removed internal Flip-Flop ***
    -- RDRF is now a direct pass-through pulse from the control path.
    -- This pulse goes to the SCSR to 'Set' the external status bit.
    RDRF <= set_RDRF;

    RSR_debug <= RSR_to_RDR;

end Structural;
```

Reciever Control Path

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity receiver_control_path is
    Port (
        clk          : in  STD_LOGIC;
        reset        : in  STD_LOGIC;

        -- External inputs
        Rx           : in  STD_LOGIC;

        -- Status inputs from datapath
        sample_zero : in  STD_LOGIC;
        bit_zero     : in  STD_LOGIC;
        new_zero     : in  STD_LOGIC;

        -- Control outputs to datapath
        sample_load : out STD_LOGIC;
        sample_dec  : out STD_LOGIC;
        bit_load     : out STD_LOGIC;
        bit_dec      : out STD_LOGIC;
        new_load     : out STD_LOGIC;
        new_dec      : out STD_LOGIC;
        RSR_load     : out STD_LOGIC;
        shift_R      : out STD_LOGIC;
        load_RDR     : out STD_LOGIC;
        set_RDRF     : out STD_LOGIC;

        -- Debug: State outputs
        S0_out       : out STD_LOGIC;
        S1_out       : out STD_LOGIC;
        S2_out       : out STD_LOGIC;
        S3_out       : out STD_LOGIC;
        S4_out       : out STD_LOGIC;
        S5_out       : out STD_LOGIC
    );

end receiver_control_path;

architecture Structural of receiver_control_path is
    -- D flip-flop with reset to 0
    component dFF_reset0
        Port (
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

    -- D flip-flop with set to 1 (for S0 initial state)
    component dFF_set1
        Port (
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

    -- State signals (one-hot encoding)
    signal S0, S1, S2, S3, S4, S5 : STD_LOGIC;  -- Current state

    signal D0, D1, D2, D3, D4, D5 : STD_LOGIC;  -- Next state

    -- Helper signals
    signal Rx_n, sample_zero_n, bit_zero_n, new_zero_n : STD_LOGIC;

begin
    ---------------------------------------------------------------
    -- STATE REGISTER: 6 flip-flops (one-hot)
    ---------------------------------------------------------------

    -- S0 uses set-to-1 flip-flop (initial state)
    FF0: dFF_set1 port map (D => D0, clk => clk, reset => reset, Q => S0);

    -- All other states use reset-to-0 flip-flops
    FF1: dFF_reset0 port map (D => D1, clk => clk, reset => reset, Q => S1);
    FF2: dFF_reset0 port map (D => D2, clk => clk, reset => reset, Q => S2);
    FF3: dFF_reset0 port map (D => D3, clk => clk, reset => reset, Q => S3);
    FF4: dFF_reset0 port map (D => D4, clk => clk, reset => reset, Q => S4);
    FF5: dFF_reset0 port map (D => D5, clk => clk, reset => reset, Q => S5);

    ---------------------------------------------------------------
    -- INVERTED SIGNALS
    ---------------------------------------------------------------

    Rx_n          <= not Rx;
    sample_zero_n <= not sample_zero;
    bit_zero_n    <= not bit_zero;
    new_zero_n    <= not new_zero;

    ---------------------------------------------------------------
    -- NEXT STATE LOGIC (exactly your equations!)
    ---------------------------------------------------------------

   D0 <= (S0 and Rx) or S5;

   D1 <= (S0 and Rx_n) or (S1 and sample_zero_n);

   D2 <= (S1 and sample_zero) or (S4 and bit_zero_n and new_zero);

   D3 <= S2;

   D4 <= S3 or (S4 and new_zero_n);  -- ← FIXED: Only stay if counting

   D5 <= S4 and bit_zero and new_zero;

    ---------------------------------------------------------------
    -- OUTPUT LOGIC (Control Signals)
    ---------------------------------------------------------------

    -- Counter loads
    sample_load <= S0;
    bit_load    <= S0;
    new_load    <= S2;

    -- Counter decrements
    sample_dec <= S1;
    bit_dec    <= S3;
    new_dec    <= S3 or S4;

    -- Register controls
    RSR_load <= S2;
    shift_R  <= S3;
    load_RDR <= S5;
    set_RDRF <= S5;

    -- Output states for debugging
    S0_out <= S0;
    S1_out <= S1;
    S2_out <= S2;
    S3_out <= S3;
    S4_out <= S4;
    S5_out <= S5;

end Structural;
```

Transmitter BDF



Transmitter Controller

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity transmitter_controller is
    port(
        bclk : in std_logic;
        GReset : in std_logic;
        bit_count_zero : in std_logic;
        TDRE : in std_logic;
        load : in std_logic;

        ShR : out std_logic;
        TDRE_flag : out std_logic;
        loadTSR : out std_logic;
        loadTDR : out std_logic;
        bit_load : out std_logic;
        bit_dec : out std_logic
    );
end entity;

architecture structural of transmitter_controller is
    signal q_s0, q_s1, q_s2, q_s3: std_logic;
    signal tdre_internal, tdre_next_logic : std_logic;

    component dFF_reset0 is
        port(
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

    component dFF_set1 is
        port(
            D     : in  STD_LOGIC;
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            Q     : out STD_LOGIC
        );
    end component;

begin

    U_D0 : dFF_set1
        port map(
            D => (q_s0 and TDRE) or (q_s3 and TDRE),
            clk => bclk,
            reset => GReset,
            Q => q_s0
        );

    U_D1 : dFF_reset0
        port map(
            D => (q_s0 and not TDRE) or (q_s3 and not TDRE),
            clk => bclk,
            reset => GReset,
            Q => q_s1
        );

    U_D2 : dFF_reset0
        port map(


            D => q_s1 or (q_s2 and (not bit_count_zero)),
            clk => bclk,
            reset => GReset,
            Q => q_s2
        );

    U_D3 : dFF_reset0
       port map(
            D => q_s2 and bit_count_zero,
            clk => bclk,
            reset => GReset,
            Q => q_s3
        );



    loadTSR <= q_s1;
    loadTDR <= load;
    bit_dec <= q_s2;
    bit_load <= q_s1;
    ShR <= q_s2;
    TDRE_flag <= q_s1;

end architecture structural;
```

Transmitter Controlpath

```vhdl
-- transmitter_controlpath.vhd
-- Top-level control path connecting state register, next state logic, and output logic
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY transmitter_controlpath IS
    PORT(
        -- Clock and reset
        i_clock           : IN  STD_LOGIC;
        i_reset           : IN  STD_LOGIC;

        -- Condition inputs
        i_RDRF            : IN  STD_LOGIC;  -- From SCSR bit 6
        i_TDRE            : IN  STD_LOGIC;  -- From SCSR bit 7
        i_state_changed   : IN  STD_LOGIC;  -- From state_change_detector
        i_byte_count_done : IN  STD_LOGIC;  -- From CBRC

        -- Control outputs
        o_ADDR            : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        o_load_TDR        : OUT STD_LOGIC;
        o_clear_RDRF      : OUT STD_LOGIC;
        o_byte_count_load : OUT STD_LOGIC;
        o_byte_count_inc  : OUT STD_LOGIC;

        -- Debug state outputs
        o_debug_S0        : OUT STD_LOGIC;
        o_debug_S1        : OUT STD_LOGIC;
        o_debug_S2        : OUT STD_LOGIC;
        o_debug_S3        : OUT STD_LOGIC;
        o_debug_S4        : OUT STD_LOGIC;
        o_debug_S5        : OUT STD_LOGIC
    );
END transmitter_controlpath;

ARCHITECTURE structural OF transmitter_controlpath IS
    -- Component declarations
    COMPONENT state_register
        PORT(
            i_clock   : IN  STD_LOGIC;
            i_reset   : IN  STD_LOGIC;
            i_S0_next : IN  STD_LOGIC;
            i_S1_next : IN  STD_LOGIC;
            i_S2_next : IN  STD_LOGIC;
            i_S3_next : IN  STD_LOGIC;
            i_S4_next : IN  STD_LOGIC;
            i_S5_next : IN  STD_LOGIC;
            o_S0      : OUT STD_LOGIC;
            o_S1      : OUT STD_LOGIC;
            o_S2      : OUT STD_LOGIC;
            o_S3      : OUT STD_LOGIC;
            o_S4      : OUT STD_LOGIC;
            o_S5      : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT next_state_logic
        PORT(
            i_S0, i_S1, i_S2, i_S3, i_S4, i_S5 : IN  STD_LOGIC;
            i_RDRF            : IN  STD_LOGIC;
            i_TDRE            : IN  STD_LOGIC;
            i_state_changed   : IN  STD_LOGIC;
            i_byte_count_done : IN  STD_LOGIC;

            o_S4_next         => s_S4_next,
            o_S5_next         => s_S5_next
        );

    -- Output logic
    out_logic: output_logic
        PORT MAP(
            i_S0              => s_S0,
            i_S1              => s_S1,
            i_S2              => s_S2,
            i_S3              => s_S3,
            i_S4              => s_S4,
            i_S5              => s_S5,
            o_ADDR            => o_ADDR,
            o_load_TDR        => o_load_TDR,
            o_clear_RDRF      => o_clear_RDRF,
            o_byte_count_load => o_byte_count_load,
            o_byte_count_inc  => o_byte_count_inc
        );

    -- Debug outputs: expose FSM states
    o_debug_S0 <= s_S0;
    o_debug_S1 <= s_S1;
    o_debug_S2 <= s_S2;
    o_debug_S3 <= s_S3;
    o_debug_S4 <= s_S4;
    o_debug_S5 <= s_S5;

END structural;
```

```vhdl
            o_S0_next         : OUT STD_LOGIC;
            o_S1_next         : OUT STD_LOGIC;
            o_S2_next         : OUT STD_LOGIC;
            o_S3_next         : OUT STD_LOGIC;
            o_S4_next         : OUT STD_LOGIC;
            o_S5_next         : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT output_logic
        PORT(
            i_S0, i_S1, i_S2, i_S3, i_S4, i_S5 : IN  STD_LOGIC;
            o_ADDR            : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
            o_load_TDR        : OUT STD_LOGIC;
            o_clear_RDRF      : OUT STD_LOGIC;
            o_byte_count_load : OUT STD_LOGIC;
            o_byte_count_inc  : OUT STD_LOGIC
        );
    END COMPONENT;

    -- Internal signals
    SIGNAL s_S0, s_S1, s_S2, s_S3, s_S4, s_S5 : STD_LOGIC;
    SIGNAL s_S0_next, s_S1_next, s_S2_next : STD_LOGIC;
    SIGNAL s_S3_next, s_S4_next, s_S5_next : STD_LOGIC;

BEGIN
    -- State register
    state_reg: state_register
        PORT MAP(
            i_clock   => i_clock,
            i_reset   => i_reset,
            i_S0_next => s_S0_next,
            i_S1_next => s_S1_next,
            i_S2_next => s_S2_next,
            i_S3_next => s_S3_next,
            i_S4_next => s_S4_next,
            i_S5_next => s_S5_next,
            o_S0      => s_S0,
            o_S1      => s_S1,
            o_S2      => s_S2,
            o_S3      => s_S3,
            o_S4      => s_S4,
            o_S5      => s_S5
        );

    -- Next state logic
    nsl: next_state_logic
        PORT MAP(
            i_S0              => s_S0,
            i_S1              => s_S1,
            i_S2              => s_S2,
            i_S3              => s_S3,
            i_S4              => s_S4,
            i_S5              => s_S5,
            i_RDRF            => i_RDRF,
            i_TDRE            => i_TDRE,
            i_state_changed   => i_state_changed,
            i_byte_count_done => i_byte_count_done,
            o_S0_next         => s_S0_next,
            o_S1_next         => s_S1_next,
            o_S2_next         => s_S2_next,
            o_S3_next         => s_S3_next,
```

UART Tx

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity uart_tx is
   port(
      loadTDR : in std_logic;
      loadTSR : in std_logic;
      ShRTSR : in std_logic;
      data_In : in std_logic_vector(7 downto 0);
      GReset : in std_logic;
      bclk : in std_logic;
      sys_clk : in std_logic;  -- *** ADDED: System clock for TDR to catch fast load pulses
***
      bit_load : in std_logic;
      bit_dec : in std_logic;
      bit_count_zero : out std_logic;
      TxD : out std_logic;
      TDR : out std_logic_vector(7 downto 0);
      TSR : out std_logic_vector(8 downto 0)

   );
end uart_tx;

architecture structural of uart_tx is
   signal TDR_Data : std_logic_vector(7 downto 0); --8 bits from TDR_Data
   signal cnt_9 : std_logic_vector(8 downto 0);
   signal TxD_out : std_logic;
   signal TSR_Data : std_LOGIC_VECTOR (8 downto 0);

   component Register_8_Bit is
      port(
         GReset   : in  STD_LOGIC;
         clk      : in  STD_LOGIC;
         load     : in  STD_LOGIC;
         data_in  : in  STD_LOGIC_VECTOR(7 downto 0);
         data_out : out STD_LOGIC_VECTOR(7 downto 0)
      );
   end component;

   component SHR_Register_9_Bit is
      port(
         ShR : in std_logic;
         load : in std_logic;
         GReset : in std_logic;
         clk : in std_logic;
         data : in std_logic_vector(7 downto 0);
         data_out : out std_logic;
         TSR_debug : out std_logic_vector(8 downto 0)
      );
   end component;

   component down_counter is
      generic (
         WIDTH : integer := 4;           -- Bit width of counter
         LOAD_VALUE : integer := 9      -- Value to load when load signal is active
      );
      Port (
         clk      : in  STD_LOGIC;
         reset    : in  STD_LOGIC;
         load     : in  STD_LOGIC;       -- Load control signal
         dec      : in  STD_LOGIC;       -- Decrement control signal
         count    : out STD_LOGIC_VECTOR(WIDTH-1 downto 0);  -- Current count value


         zero     : out STD_LOGIC        -- '1' when count = 0
      );
   end component;

begin
   U_TSR : SHR_Register_9_Bit
      port map(
         ShR => ShRTSR,
         data => TDR_Data,
         GReset => GReset,
         clk => bclk,
         data_out => TxD_out,
         load => loadTSR,
         TSR_debug => TSR_Data
      );
   U_TDR : Register_8_Bit
      port map(
         GReset => GReset,
         clk => sys_clk,  -- *** CHANGED: Use system clock to catch fast load pulses ***
         load => loadTDR,
         data_in => data_In,
         data_out => TDR_Data
      );
   U_downCount : down_counter
      port map(
         clk => bclk,
         reset => GReset,
         load => bit_load,
         dec => bit_dec,
         count => open,
         zero => bit_count_zero
      );
   TxD <= TxD_out;
   TDR <= TDR_Data;
   TSR <= TSR_Data;
end architecture;
```

Design Part | Discussion of Solution

3.1 Discussion of Used Components

Baud Rate Generator (BaudRateGenerator.vhd)

To adhere to the strict structural modeling requirements, the Baud Rate Generator was designed without behavioral arithmetic operators. Instead, it utilizes a cascaded division strategy to derive the specific RS-232 timing signals from the 50 MHz system clock. The design begins with a clock_div_41 component that reduces the system clock to a manageable base frequency. This is followed by an 8-bit counter (counter_8_bit) which functions as a frequency divider chain, providing eight distinct clock taps. A multiplexer (mux_8to1), controlled by the 3-bit SEL input, selects the appropriate base frequency. Finally, a clock_div_8 component generates the final BClk (baud rate clock), while the input to this divider serves as the BClkx8 signal required for the receiver's oversampling logic.

UART Receiver (uart_receiver.vhd)

The receiver module is responsible for capturing asynchronous data frames and is divided into a distinct control path and datapath. The control logic (receiver_control_path.vhd) implements a Finite State Machine (FSM) using One-Hot Encoding (States S0 to S5) to sequence the detection of the Start Bit, the data sampling loop, and the Stop Bit validation. The datapath employs three structural down-counters (sample_counter, bit_counter, and new_counter) to precisely track the sampling intervals and bit positions. An 8x oversampling scheme is used to detect the start bit and sample subsequent data bits at the center of their periods. The serial data is shifted into the RSR_8bit (Receive Shift Register) and subsequently latched into the RDR_8bit (Receive Data Register) upon successful frame reception.

UART Transmitter (UART_Transmitter.vhd)

The transmitter module serializes parallel data from the system onto the TxD line and features a critical Dual-Clock Architecture to handle timing. The control logic is managed by the transmitter_controller FSM, which coordinates the loading and shifting signals. The datapath consists of an 8-bit Transmit Data Register (Register_8_Bit) and a 9-bit Transmit Shift Register (SHR_Register_9_Bit). A key design detail is that the Data Register is clocked by the fast 50 MHz sys_clk to reliably capture the CPU's short write pulse, whereas the Shift Register and bit counters are clocked by the slow bclk to ensure the data is shifted out at the correct baud rate.

UART FSM Controller (uart_fsm_top.vhd)

This component acts as the embedded microcontroller for the system, bridging the Traffic Light logic and the UART. It incorporates a Character Byte Register (CBR), which is a purely combinational lookup table that selects ASCII characters—such as 'M', 'g', or 'r'—based on the current traffic light state ($S_2S_1S_0$) and the byte count. The module also includes a state_change_detector that uses a 3-bit register and comparator to trigger transmission sequences upon detecting a state transition. The FSM manages the load_TDR signal and continuously polls the TDRE status bit to ensure reliable handshaking during transmission.

3.2 Discussion of Actual Solution

The actual solution integrates these atomic components into a cohesive system governed by a top-level Block Diagram File (.bdf). The logic flows uni-directionally, beginning with the traffic_light_controller, which operates on a 1 Hz clock to control the intersection lights and exports its internal state bits (S_2, S_1, S_0) to the top level. These state bits are captured by the state_change_detector, which asserts a pulse whenever a transition occurs.

Upon receiving this pulse, the UART_FSM_Controller resets its internal Byte Counter and initiates a fetch-and-send loop. It uses the current State and Byte Count to address the CBR, retrieving the correct ASCII character, and then asserts o_load_TDR to load the character into the Transmitter. The FSM then monitors the TDRE flag from the Status Register, waiting for the Transmitter to complete serialization before incrementing the Byte Counter and sending the next character. This ensures that the high-speed control logic remains synchronized with the lower-speed serial transmission. The connections in the top-level BDF ensure that the global reset clears all registers and that the Baud Rate Generator feeds the correct clocks to both the Transmitter and Receiver.

3.3 Discussion of Tool

The design and verification were performed using the Altera Quartus II Web Edition software, targeting the Altera DE2 Development Board (Cyclone IV EP4CE115F29C FPGA). This environment provided the necessary tools for VHDL compilation, RTL synthesis, and functional simulation via Vector Waveform Files. The board's 50 MHz oscillator served as the master clock source, while physical switches and LEDs were utilized for input configuration and state visualization. The tool's RTL Viewer was particularly valuable for verifying that the structural VHDL code was correctly synthesized into the intended logic gates and registers.

Challenging Problems (Bonus)

7.1 Clock Domain Crossing (The Fast-Write / Slow-Read Problem)

One of the most significant technical challenges encountered was synchronizing the high-speed system logic with the low-speed serial interface. The UART_FSM operates on the 50 MHz system clock and asserts the Load signal for a single clock cycle (20 ns). However, the UART_Transmitter shifts data based on the Baud Clock (approx. 9600 Hz), meaning the shift logic would likely miss the fast load pulse entirely. To resolve this, a Dual-Clock domain strategy was implemented within the transmitter. The Transmit Data Register (TDR) is clocked by the fast sys_clk to immediately capture the load pulse, while the Transmit Shift Register (TSR) is clocked by bclk. Because the TDR holds the data stable until the next write, the TSR can safely load this data at the slower baud rate edge, eliminating the need for complex asynchronous FIFO buffers.

7.2 Strict Structural Modeling Constraints

The requirement to use only structural VHDL significantly increased design complexity, as behavioral process blocks with if-else statements were prohibited. Simple operations, such as resetting a counter or defining state transitions, required custom atomic components. For example, the FSMs were decomposed into Next_State_Logic modules composed of pure combinational gates and State_Registers made of instantiated D-Flip-Flops. Similarly, timers were built using a custom down_counter_4bit entity constructed from flip-flops and manual XOR gate logic. Status flags also required a structural edge detector (SCSR.vhd) using delay flip-flops and inverters to generate clean pulses, avoiding behavioral rising-edge detection.

7.3 VHDL to BDF Migration

Transitioning from a text-based VHDL top-level to a graphical Block Diagram File (BDF) introduced integration risks, particularly regarding signal types. VHDL STD_LOGIC_VECTOR ports do not always map cleanly to BDF pins if bus widths are not explicitly defined, often resulting in compilation errors where signals are stuck at VCC or Ground. This was resolved by strictly generating Block Symbol Files (.bsf) for every verified VHDL module and using explicit bus naming conventions (e.g., Data[7..0]) during wiring. This ensured that the Quartus netlist compiler correctly associated the 8-bit buses between the CBR, Data_Bus, and UART_Transmitter

Discussion

The final implementation of the Debuggable Traffic Light Controller successfully met the core project requirements, resulting in a fully functional system that integrates synchronous traffic logic with asynchronous serial communication. The primary design success was the robust synchronization between the FPGA's 50 MHz clock domain and the 9600 baud asynchronous domain. Despite the significant speed difference, the structural edge detectors and dual-clock architecture in the transmitter ensured that no data packets were dropped during the rapid state transitions of the traffic light. This was verified during the live demonstration, where the "Mg-Sr" and "My-Sr" strings appeared on the PuTTY terminal instantaneously and without corruption, confirming that the handshaking protocol using the TDRE (Transmit Data Register Empty) flag functioned as intended to prevent buffer overruns.

However, a notable discrepancy existed between our initial theoretical design and the final actual implementation regarding the top-level entity. The original design intent was to implement the entire hierarchy, including the top-level wrapper, using pure VHDL to maintain code portability. During the implementation phase, we encountered connectivity issues where 8-bit bus signals between the UART FSM and the Transmitter were not mapping correctly, leading to "stuck-at-zero" faults during simulation. To resolve this, we shifted to a Block Diagram File (.bdf) implementation for the top-level entity. This visual approach allowed us to identify that specific bus widths were mismatched in the port maps. While this deviated from a code-only approach, the BDF implementation provided a clearer high-level view of the signal flow and ultimately facilitated the successful integration of the Baud Rate Generator and Address Decoder, which had previously been isolated components.

While the core UART system was successful, we encountered a specific error regarding the optional "BCD Decoder" bonus objective. We attempted to implement a decimal display to show the remaining time for the traffic light states on the 7-segment displays. However, in the actual hardware implementation, the displays exhibited unstable behavior, showing shifting or random segment patterns rather than clean decimal numbers. Waveform analysis suggested that this error was caused by glitching in the combinational decoder logic as the counter values transitioned. Because the counter output was directly driving the decoder without an intermediate register to stabilize the value, the 7-segment display captured transient states. Due to time constraints, we were unable to implement the necessary registering to debounce these values, and consequently, the BCD feature was omitted from the final stable build to ensure the reliability of the main traffic light and UART functions.

Conclusion

This project successfully integrated a custom UART interface with a Traffic Light Controller, transforming a closed-loop FSM into a debuggable real-time system on the Altera DE-2 board. The strict requirement for structural VHDL and the migration to a graphical top-level (BDF) challenged us to deeply understand low-level logic instantiation and precise signal routing between clock domains. Ultimately, the successful transmission of correct ASCII debug messages to the PC terminal validated our design, demonstrating the critical role of serial communication protocols and modular architecture in modern digital systems engineering.