CEG3155 Digital Systems II

UART Project Report

Submitted in partial fulfillment for the award of the credit of the course CEG3155 Computer Architecture II to ***Professor Rami Abielmona***

By

Group 26

Fareis Canoe (300299663)
Leon Mathews (300307926)

Faculty of Engineering
Department of Electrical and Computer Engineering
UNIVERSITY OF OTTAWA
September, 2024

Table of Contents

Introduction.....3

Objective.....3

Discussion of Problem.....4

Discussion of Algorithmic Solution.....25

Design Part and Simulation.....31

Discussion.....33

Report Overview.....34

Introduction

In this project, we designed and implemented a UART (Universal Asynchronous Receiver-Transmitter) in VHDL, which included creating the transmitter, receiver, baud rate generator, and UART registers for serial communication. We used the MAX232 chip to handle signal conversion between CMOS and RS-232 levels, making it possible to send data between devices. A key part of the project was integrating a traffic light controller that sends debug messages (like "Mg Sr" for green light) to a computer screen using the UART. We built and tested our circuits with Quartus II software and used the DE-2 Altera FPGA board to implement four FSMs for controlling the traffic lights, UART, and data transmission. This project helped us learn about UART design and how to apply it to real-world systems.

Objective

The objective of this project was to design and implement a UART system in VHDL to enable serial communication between devices and integrate it with a traffic light controller. This involved creating and testing components like the transmitter, receiver, and baud rate generator while using the UART to send debug messages to a computer screen. The project aimed to deepen our understanding of UART design and its practical applications in systems like traffic light controllers.

Discussion of Problem

The problem involves designing a UART that can operate in real-time to send and receive serial data asynchronously. The UART is integrated with a traffic light controller to produce debug messages corresponding to state transitions.

- **UART (UART.vhd)**

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

ENTITY UART IS
	PORT (
		RxD, Load, GReset, GClock : IN STD_LOGIC;
		State_Information : IN STD_LOGIC_VECTOR(5 downto 0);
		TxD, CanProceedState : OUT STD_LOGIC
	);
END UART;

ARCHITECTURE basic of UART IS

	COMPONENT UARTTransmitter
```

```
		PORT (
			loadTDR, GReset, GClock, BaudRate, shiftLoad : IN STD_LOGIC;
			D_In : IN STD_LOGIC_VECTOR(6 downto 0);
			TxD, TDRE, countEqualTenOut : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT counter_4bit
		PORT (
			CLK   : in STD_LOGIC;
      i_enable   : in STD_LOGIC;
			i_hold : IN STD_LOGIC;
			i_reset : IN STD_LOGIC;
      COUNT : out STD_LOGIC_VECTOR (3 downto 0)
		);
	END COMPONENT;

	COMPONENT equality_Comparator_4bit
		PORT (
			A, B : IN STD_LOGIC_VECTOR(3 downto 0);
			isEqual : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT UARTReciever
		PORT (
			RxD, GReset, BClkx8, GClock : IN STD_LOGIC;
			RecieveOut : OUT STD_LOGIC_VECTOR(7 downto 0);
			RDRF : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT dFF_8bit
		PORT (
			i_d	: IN  STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit input data
			i_en		: IN STD_LOGIC;
			i_reset		: IN STD_LOGIC;
			i_clock	: IN  STD_LOGIC;			-- Clock input
			o_q	: OUT STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit output
data
			o_qBar	: OUT STD_LOGIC_VECTOR(7 downto 0)
		);
	END COMPONENT;

	COMPONENT BaudRateGen
		PORT (
```

```
                GClock, GReset : IN STD_LOGIC;
                SEL : IN STD_LOGIC_VECTOR(2 downto 0);
                BClk, BClkx8 : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT InterruptGenerator
        PORT (
                TIE, RIE, RDRF, OE, TDRE: in std_logic;
                IRQ : out std_logic_vector(1 downto 0)
        );
    END COMPONENT;

    COMPONENT UARTaddressDecode
        PORT (
                Address : in STD_LOGIC_VECTOR(1 downto 0);
                RWbar   : in STD_LOGIC;
                DecodeOut   : out STD_LOGIC_VECTOR(5 downto 0)
        );
    END COMPONENT;

    COMPONENT d_FF
  PORT(
    i_d, i_en, i_reset      : IN  STD_LOGIC;
    i_clock   : IN  STD_LOGIC;
    o_q       : OUT STD_LOGIC;
    o_qBar    : OUT STD_LOGIC
  );
 END COMPONENT;

     COMPONENT mux_8to1_1bit
        PORT (
                d_in : IN STD_LOGIC_VECTOR(7 downto 0);  -- 8 inputs
                sel : IN STD_LOGIC_VECTOR(2 downto 0);   -- 3-bit selector
                d_out : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL TransitorySCSROutLatched, TransitorySCCRLatched, DataBus,
TransitoryRDROut, TransitorySCCROut, TransitorySCSROut, TransitoryRDRHeld :
STD_LOGIC_VECTOR(7 downto 0);
    SIGNAL TransitoryMSTLState, TransitorySSTLState : STD_LOGIC_VECTOR(1
downto 0);
    SIGNAL selectBits, ByteToSelect : STD_LOGIC_VECTOR(2 downto 0);
    SIGNAL BytesSent : STD_LOGIC_VECTOR(3 downto 0);
    SIGNAL TransitoryTxD, TransitoryCanProceedStateLatched,
```

```
transitoryCountEqualTen, incrementByteCounter, ResetByteCounter,
TransitoryCanProceedState, DoneSendingWord, BClk, BClkx8, transitoryRDRF,
transitoryTDRE, transitoryTDRELatched : STD_LOGIC;
        SIGNAL ByteToSend, transitoryTDRByte, ByteCR, ByteM, ByteY, ByteR,
ByteUnderscore, ByteS, ByteG : STD_LOGIC_VECTOR(6 downto 0);
        SIGNAL WordToSend, State1Word, State2Word, State3Word, State4Word :
STD_LOGIC_VECTOR(41 downto 0);
begin

        ByteM <= "1001101";
        ByteY <= "1011001";
        ByteR <= "1010010";
        ByteUnderscore <= "1011111";
        ByteS <= "1010011";
        ByteG <= "1000111";
        ByteCR <= "0001101";

        State1Word <= ByteM & ByteG & ByteUnderscore & ByteS & ByteR & ByteCR;
        State2Word <= ByteM & ByteY & ByteUnderscore & ByteS & ByteR & ByteCR;
        State3Word <= ByteM & ByteR & ByteUnderscore & ByteS & ByteG & ByteCR;
        State4Word <= ByteM & ByteR & ByteUnderscore & ByteS & ByteY & ByteCR;

        WordToSend <= State1Word when (State_Information = "100001") else
          State2Word when (State_Information = "010001") else
          State3Word when (State_Information = "001100") else
          State4Word when (State_Information = "001010") else
                                State1Word when (State_Information = "001001") else
          (others => '0');

        ByteToSend <= WordToSend(41 downto 35) when BytesSent = "0000" else
            WordToSend(34 downto 28) when BytesSent = "0001" else
            WordToSend(27 downto 21) when BytesSent = "0010" else
            WordToSend(20 downto 14) when BytesSent = "0011" else
            WordToSend(13 downto 7)  when BytesSent = "0100" else
            WordToSend(6 downto 0)   when BytesSent = "0101" else
            (others => '0');

        incrementByteCounter <= transitoryTDRELatched AND TransitoryCountEqualTen;
        d_FF_proceedStateLatched : d_FF
                PORT MAP (
                        i_d => TransitoryCanProceedState,
                        i_en => '1',
                        i_reset => GReset,
                        i_clock => GClock,
                        o_q => TransitoryCanProceedStateLatched,
                        o_qBar => open
```

```
        );
CanProceedState <= TransitoryCanProceedStateLatched;
ResetByteCounter <= GReset OR TransitoryCanProceedStateLatched;

counter_4bit_BytesSent : counter_4bit
        PORT MAP (
                CLK => incrementByteCounter,
                i_enable => '1',
                i_hold => '0',
                i_reset => ResetByteCounter,
                COUNT => BytesSent
        );

equality_Comparator_4bit_sent6Bytes : equality_Comparator_4bit
        PORT MAP (
                A => BytesSent,
                B => "0110",
                isEqual => TransitoryCanProceedState
        );

dFF_8bit_SCCR : dFF_8bit
        PORT MAP (
                i_d => TransitorySCCROut,
                i_en => '1',
                i_reset => GReset,
                i_clock => GClock,
                o_q => TransitorySCCRLatched,
                o_qBar => open
        );
TransitorySCSROut(6) <= transitoryRDRF;

dFF_8bit_SCSR : dFF_8bit
        PORT MAP (
                i_d => TransitorySCSROut,
                i_en => '1',
                i_reset => GReset,
                i_clock => GClock,
                o_q => TransitorySCSROutLatched,
                o_qBar => open
        );

selectBits <= "000";

BaudRateGen_inst : BaudRateGen
        PORT MAP (
                GClock => GClock,
```

```
                GReset => GReset,
                SEL => selectBits,
                BClk => BClk,
                BClkx8 => BClkx8
        );

UARTReciever_inst : UARTReciever
        PORT MAP (
                RxD => TransitoryTxD,
                GClock => GClock,
                BClkx8 => BClkx8,
                GReset => GReset,
                RecieveOut => TransitoryRDROut,
                RDRF => transitoryRDRF
        );


dFF_8bit_HoldRecieverOut : dFF_8bit
        PORT MAP (
                i_d => TransitoryRDROut,
                i_en => '1',
                i_reset => GReset,
                i_clock => transitoryRDRF,
                o_q => TransitoryRDRHeld,
                o_qBar => open
        );

UARTTransmitter_inst : UARTTransmitter
        PORT MAP (
                loadTDR => '1',
                GReset => GReset,
                GClock => GClock,
                BaudRate => BClk,
                shiftLoad => transitoryTDRELatched,
                D_In => ByteToSend,
                TxD => TransitoryTxD,
                TDRE => transitoryTDRE,
                countEqualTenOut => TransitoryCountEqualTen
        );

d_FF_latchTDRE : d_FF
        PORT MAP (
                i_d => transitoryTDRE,
                i_en => '1',
                i_reset => GReset,
                i_clock => GClock,
```

```
                o_q => transitoryTDRELatched,
                o_qBar => open
        );
    TxD <= TransitoryTxD;

end basic;
```

This VHDL code sets up a UART (Universal Asynchronous Receiver/Transmitter) system using several smaller components that work together. It includes a transmitter, receiver, baud rate generator, and other parts like counters and flip-flops to control how data is sent and received. The system is controlled by signals like a clock (GClock) and reset (GReset), and it works in different states to handle tasks like choosing which word to send or receiving data. The transmitter sends data one byte at a time, while the counters keep track of the process and make sure everything happens at the right time

- **Transmit Module (UARTTransmitter.vhd)**

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

ENTITY UARTTransmitter IS
    PORT (
        loadTDR, GReset, GClock, BaudRate, shiftLoad : IN STD_LOGIC;
        D_In : IN STD_LOGIC_VECTOR(6 downto 0);
        TxD, TDRE, countEqualTenOut : OUT STD_LOGIC
    );
END UARTTransmitter;

ARCHITECTURE basic of UARTTransmitter IS

    COMPONENT dFF_8bit
        PORT (
            i_d     : IN  STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit input data
            i_en        : IN STD_LOGIC;
            i_reset     : IN STD_LOGIC;
            i_clock     : IN  STD_LOGIC;                -- Clock input
            o_q     : OUT STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit output
data
            o_qBar      : OUT STD_LOGIC_VECTOR(7 downto 0)
        );
    END COMPONENT;

    COMPONENT ParallelInSerialOut_8Bit
        PORT (
            i_d         : IN    STD_LOGIC_VECTOR(9 downto 0);
```

```
                i_reset         : IN    STD_LOGIC;
                i_enable        : IN    STD_LOGIC;
                i_shift  : IN  STD_LOGIC;
                i_shift_Load            : IN    STD_LOGIC;
                i_clock                 : IN    STD_LOGIC;
                s_out  : OUT  STD_LOGIC
        );
  END COMPONENT;

  COMPONENT parityGen
        PORT (
                i_d     : IN    STD_LOGIC_VECTOR(6 downto 0);
                p_out  : OUT  STD_LOGIC
        );
  END COMPONENT;

  COMPONENT mux_2to1_1bit
        PORT (
                d0 : IN STD_LOGIC;
                d1 : IN STD_LOGIC;
                sel : IN STD_LOGIC;
                d_out : OUT STD_LOGIC
        );
  END COMPONENT;

  COMPONENT counter_4bit
        PORT (
                CLK   : in STD_LOGIC;
i_enable   : in STD_LOGIC;
                i_hold : IN STD_LOGIC;
                i_reset : IN STD_LOGIC;
COUNT : out STD_LOGIC_VECTOR (3 downto 0)
        );
  END COMPONENT;

  COMPONENT equality_Comparator_4bit
        PORT (
                A, B : IN STD_LOGIC_VECTOR(3 downto 0);
                isEqual : OUT STD_LOGIC
        );
  END COMPONENT;

  signal transitoryI_D : STD_LOGIC_VECTOR(9 downto 0);
  signal d_indff, d_indffLatched : STD_LOGIC_VECTOR(7 downto 0);
  signal transitorySerialOut, enableCount, countEqualTen : STD_LOGIC;
  signal countValue : STD_LOGIC_VECTOR(3 downto 0);
```

```
begin


	transitoryI_D(9) <= '1';
	transitoryI_D(0) <= '0';

	d_indff(6 downto 0) <= D_In;
	d_indff(7) <= '0';

	dFF_8bitID : dFF_8bit
		PORT MAP (
			i_d => d_indff,
			i_en => loadTDR,
			i_reset => GReset,
			i_clock => GClock,
			o_q => d_indffLatched,
			o_qBar => open
		);

	transitoryI_D(7 downto 1) <= d_indffLatched(6 downto 0);

	parityGen_check : parityGen
		PORT MAP (
			i_d => transitoryI_D(7 downto 1),
			p_out => transitoryI_D(8)
		);

	ParallelInSerialOut_8BitTXD : ParallelInSerialOut_8Bit
		PORT MAP (
			i_d => transitoryI_D,
			i_reset => GReset,
			i_enable => '1',
			i_shift => '1',
			i_shift_Load => shiftLoad,
			i_clock => BaudRate,
			s_out => transitorySerialOut
		);


	mux_2to1_1bit_selout : mux_2to1_1bit
		PORT MAP (
			d0 => transitorySerialOut,
			d1 => '1',
			sel => shiftLoad,
			d_out => TxD
		);
```

```
	enableCount <= not countEqualTen;

	counter_4bit_count_10 : counter_4bit
		PORT MAP (
			CLK => BaudRate,
			i_enable => enableCount,
			i_hold => '0',
			i_reset => GReset,
			COUNT => countValue
		);

	equality_Comparator_4bit_donesend : equality_Comparator_4bit
		PORT MAP (
			A => countValue,
			B => "1010",
			isEqual => countEqualTen
		);
	countEqualTenOut <= countEqualTen;
	TDRE <= countEqualTen OR
	(NOT (countValue(3) XOR '0') AND
		 NOT (countValue(2) XOR '0') AND
		 NOT (countValue(1) XOR '0') AND
		 NOT (countValue(0) XOR '0'));

end basic;
```

This VHDL code implements a UART Transmitter, which converts 7-bit parallel data (D_In) into a serial data stream (TxD) for transmission. It starts by latching the input data using an 8-bit flip-flop (dFF_8bit) and calculates a parity bit for error-checking (parityGen). The data frame, including a start bit ('0'), the 7 data bits, a parity bit, and a stop bit ('1'), is converted into a serial bitstream using a ParallelInSerialOut_8Bit module. A 4-bit counter tracks the bits being sent, and when all 10 bits (the full frame) are transmitted, a comparator signals the transmission is complete (countEqualTen). The mux_2to1_1bit determines whether to send the serial data or hold the stop bit, ensuring proper output on TxD. Additionally, the TDRE signal indicates when the system is ready for new data.

- **Receive Module (UARTReciever.vhd)**

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

ENTITY UARTReciever IS
	PORT (
		RxD, GReset, BClkx8, GClock : IN STD_LOGIC;
		RecieveOut : OUT STD_LOGIC_VECTOR(7 downto 0);
```

```
                RDRF : OUT STD_LOGIC
        );
END UARTReciever;

ARCHITECTURE basic of UARTReciever IS

        COMPONENT counter_4bit
                PORT (
                        CLK   : in STD_LOGIC;
      i_enable   : in STD_LOGIC;
                  i_hold : IN STD_LOGIC;
                  i_reset : IN STD_LOGIC;
      COUNT : out STD_LOGIC_VECTOR (3 downto 0)
                );
        END COMPONENT;

        COMPONENT equality_Comparator_4bit
                PORT (
                        A, B : IN STD_LOGIC_VECTOR(3 downto 0);
                        isEqual : OUT STD_LOGIC
                );
        END COMPONENT;

        COMPONENT SerialInParallelOut_8Bit
                PORT (
                        i_reset         : IN    STD_LOGIC;
                        i_enable        : IN    STD_LOGIC;
                        i_shift  : IN  STD_LOGIC;
                        i_clock         : IN    STD_LOGIC;
                        s_out  : OUT  STD_LOGIC_VECTOR(7 downto 0)
                );
        END COMPONENT;

        COMPONENT dFF_8bit
                PORT (
                        i_d        : IN  STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit input data
                        i_en               : IN STD_LOGIC;
                        i_reset          : IN STD_LOGIC;
                        i_clock     : IN  STD_LOGIC;                    -- Clock input
                        o_q        : OUT STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit output
data
                        o_qBar      : OUT STD_LOGIC_VECTOR(7 downto 0)
                );
        END COMPONENT;

        COMPONENT d_FF
```

```
    PORT(
        i_d, i_en, i_reset      : IN  STD_LOGIC;
        i_clock   : IN  STD_LOGIC;
        o_q       : OUT STD_LOGIC;
        o_qBar    : OUT STD_LOGIC
    );
  END COMPONENT;

        SIGNAL countFourVal, countEightVal, numBitsCollected : STD_LOGIC_VECTOR(3
downto 0);
        SIGNAL transitoryPRecieve : STD_LOGIC_VECTOR(7 downto 0);
        SIGNAL isInMiddle, isAtEight, startSample, isStartBit, Latch8, Latch4, captureBit,
finishedCapture, finishedCaptureLatch, resetbitCounter : STD_LOGIC;
begin
        counter_4bit_countmiddle : counter_4bit
                PORT MAP (
                        CLK => BClkx8,
                        i_enable => isInMiddle,
                        i_hold => '0',
                        i_reset => GReset,
                        COUNT => countFourVal
                );
        counter_4bit_counteight : counter_4bit
                PORT MAP (
                        CLK => BClkx8,
                        i_enable => Latch8,
                        i_hold => '0',
                        i_reset => GReset,
                        COUNT => countEightVal
                );

        isInMiddle <= NOT isStartBit AND NOT startSample AND NOT (NOT
(countFourVal(3) XOR '0')
        AND NOT (countFourVal(2) XOR '0') AND NOT (countFourVal(1) XOR '1') AND
NOT (countFourVal(0) XOR '1'));
        isAtEight <= (NOT isInMiddle AND isStartBit) OR (startSample AND NOT (NOT
(countEightVal(3) XOR '0')
        AND NOT (countEightVal(2) XOR '1') AND NOT (countEightVal(1) XOR '1') AND
NOT (countEightVal(0) XOR '0')));
        d_FF_latch8 : d_FF
                PORT MAP (
                        i_d => isAtEight,
                        i_en => '1',
                        i_reset => GReset,
                        i_clock => BClkx8,
                        o_q => Latch8,
```

```
                o_qBar => open
        );
    d_FF_latch4 : d_FF
        PORT MAP (
                i_d => isInMiddle,
                i_en => '1',
                i_reset => GReset,
                i_clock => BClkx8,
                o_q => Latch4,
                o_qBar => open
        );
    isStartBit <= NOT RxD AND NOT Latch4 AND NOT (numBitsCollected(3) XOR '0')
AND NOT (numBitsCollected(2) XOR '0') AND NOT (numBitsCollected(1) XOR '0') AND
NOT (numBitsCollected(0) XOR '0');
    equality_Comparator_4bit_startsmpl : equality_Comparator_4bit
        PORT MAP (
                A => numBitsCollected,
                B => "0001",
                isEqual => startSample
        );

    equality_Comparator_4bit_finishsmpl : equality_Comparator_4bit
        PORT MAP (
                A => numBitsCollected,
                B => "1001",
                isEqual => finishedCapture
        );
    d_FF_finishedCap : d_FF
        PORT MAP (
                i_d => finishedCapture,
                i_en => '1',
                i_reset => GReset,
                i_clock => BClkx8,
                o_q => finishedCaptureLatch,
                o_qBar => open
        );
    resetbitCounter <= finishedCaptureLatch OR GReset;
    captureBit <= (NOT Latch4 AND Latch8);
    counter_4bit_countRecieved : counter_4bit
        PORT MAP (
                CLK => captureBit,
                i_enable => '1',
                i_hold => '0',
                i_reset => resetbitCounter,
                COUNT => numBitsCollected
        );
```

```
	SerialInParallelOut_8Bitinst : SerialInParallelOut_8Bit
		PORT MAP (
			i_reset => GReset,
			i_enable => '1',
			i_shift => RxD,
			i_clock => Latch8,
			s_out => transitoryPRecieve
		);

	dFF_8bit_RDR : dFF_8bit
		PORT MAP (
			i_d => transitoryPRecieve,
			i_en => '1',
			i_reset => GReset,
			i_clock => BClkx8,
			o_q => RecieveOut,
			o_qBar => open
		);

	RDRF <= finishedCapture;


end basic;
```

This VHDL implementation of a UART Receiver converts serial data from RxD into parallel data (RecieveOut) while detecting the reception status with RDRF. A counter_4bit tracks the bit sampling progress, and an equality_Comparator_4bit ensures proper synchronization by identifying start and stop conditions. The serial data is shifted into an 8-bit register (SerialInParallelOut_8Bit) after sampling and is latched into RecieveOut using a flip-flop (dFF_8bit). Control signals like isInMiddle, isStartBit, and captureBit manage the reception flow, while latch signals (Latch4 and Latch8) and a bit counter ensure correct timing and byte alignment. Finally, RDRF indicates when data reception is complete, and the system resets counters as needed for continuous operation.

- **Baud Rate Generator (BaudRateGen.vhd)**

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

ENTITY BaudRateGen IS
	PORT (
		GClock, GReset : IN STD_LOGIC;
		SEL : IN STD_LOGIC_VECTOR(2 downto 0);
		BClk, BClkx8 : OUT STD_LOGIC
	);
```

```
END BaudRateGen;

ARCHITECTURE basic of BaudRateGen IS

	COMPONENT counter_6bit
		PORT (
			CLK   : in STD_LOGIC;
			i_enable   : in STD_LOGIC;
			i_hold : IN STD_LOGIC;
			i_reset : IN STD_LOGIC;
			COUNT : out STD_LOGIC_VECTOR (5 downto 0)
		);
	END COMPONENT;

	COMPONENT counter_8bit
		PORT (
			CLK   : in STD_LOGIC;
			i_enable   : in STD_LOGIC;
			i_hold : IN STD_LOGIC;
			i_reset : IN STD_LOGIC;
			COUNT : out STD_LOGIC_VECTOR (7 downto 0)
		);
	END COMPONENT;

	COMPONENT equality_Comparator_6bit
		PORT (
			A, B : IN STD_LOGIC_VECTOR(5 downto 0);
			isEqual : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT mux_2to1_1bit
		PORT (
			d0 : IN STD_LOGIC;
			d1 : IN STD_LOGIC;
			sel : IN STD_LOGIC;
			d_out : OUT STD_LOGIC
		);
	END COMPONENT;

	COMPONENT mux_8to1_1bit
		PORT (
			d_in : IN STD_LOGIC_VECTOR(7 downto 0);  -- 8 inputs
			sel : IN STD_LOGIC_VECTOR(2 downto 0);   -- 3-bit selector
			d_out : OUT STD_LOGIC                    -- Output
		);
	END COMPONENT;

	component d_FF
    Port (
```

```
                                i_d, i_en, i_reset                              : IN    STD_LOGIC;
                                i_clock                 : IN    STD_LOGIC;
                                o_q, o_qBar             : OUT   STD_LOGIC
    );
  end component;

        SIGNAL fourty, clockVal : STD_LOGIC_VECTOR(5 downto 0);
        SIGNAL newclk : STD_LOGIC_VECTOR(7 downto 0);
        SIGNAL STDDiv41Clock, div41IsEqual, div41IsEqualGated, resetCheck : STD_LOGIC;
        SIGNAL STDDiv41ClockValue : STD_LOGIC := '1';
begin
        fourty <= "101000";
        resetCheck <= GReset OR div41IsEqualGated;

        counter_6bit_div41 : counter_6bit
                PORT MAP (
                        CLK => GClock,
                        i_enable => '1',
                        i_hold => '0',
                        i_reset => resetCheck,
                        COUNT => clockVal
                );

        equality_Comparator_6bit_check41 : equality_Comparator_6bit
                PORT MAP (
                        A => clockVal,
                        B => fourty,
                        isEqual => div41IsEqual
                );

        d_FF_Gate_isEqual : d_FF
                PORT MAP (
                        i_d => div41IsEqual,
                        i_en => '1',
                        i_reset => GReset,
                        i_clock => GClock,
                        o_q => div41IsEqualGated,
                        o_qBar => open
                );

        STDDiv41ClockValue <= ((STDDiv41Clock XOR div41IsEqualGated) AND NOT GReset)
OR GReset;

        d_FF_Gate_clk : d_FF
                PORT MAP (
                        i_d => STDDiv41ClockValue,
                        i_en => '1',
                        i_reset => '0',
                        i_clock => GClock,
                        o_q => STDDiv41Clock,
```

```
                o_qBar => open
        );

    counter_8bit_inst : counter_8bit
        PORT MAP (
                CLK => STDDiv41Clock,
                i_enable => '1',
                i_hold => '0',
                i_reset => GReset,
                COUNT => newclk
        );

    mux_8to1_1bit_inst : mux_8to1_1bit
        PORT MAP (
                d_in => newclk,
                sel => SEL,
                d_out => BClk
        );

    BClkx8 <= newclk(0);

end basic;
```

The BaudRateGen module takes an input clock (GClock) and creates two baud rate clocks: BClk and BClkx8. It uses a 6-bit counter to divide the clock and checks when the counter reaches a specific value (40). Then, it uses an 8-bit counter to generate the final clock. A flip-flop stores the results, and a multiplexer selects the output baud clock based on the SEL input. This way, the module controls how fast the clock signals are generated for communication.

- **6-bit Counter (counter_6bit.vhd)**

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

entity counter_6bit is
   Port (
      CLK   : in STD_LOGIC;
      i_enable   : in STD_LOGIC;
                  i_hold : IN STD_LOGIC;
                  i_reset : IN STD_LOGIC;
      COUNT : out STD_LOGIC_VECTOR (5 downto 0)
   );
end counter_6bit;

architecture Structural of counter_6bit is
  component d_FF
     Port (
                              i_d                    : IN      STD_LOGIC;
```

```
                                i_en                    : IN STD_LOGIC;
                                i_reset         : IN STD_LOGIC;
                                i_clock                 : IN    STD_LOGIC;
                                o_q, o_qBar             : OUT   STD_LOGIC
    );
  end component;

        component mux_2to1_6bit
                PORT (
                        sel     : IN  STD_LOGIC;                    -- Select input
      d_in1   : IN  STD_LOGIC_VECTOR(5 downto 0);       -- 8-bit Data input 1
      d_in2   : IN  STD_LOGIC_VECTOR(5 downto 0);       -- 8-bit Data input 2                --
Reset input
      d_out   : OUT STD_LOGIC_VECTOR(5 downto 0)
                );
        END COMPONENT;

        signal t_en : std_LOGIC;
  signal Q, td, td_2, td_3 : STD_LOGIC_VECTOR(5 downto 0);
begin
  -- Flip-flop instances
        FF0: d_FF
                port map (
                        i_d => td(0),
                        i_en => i_enable,
                        i_reset => i_reset,
                        i_clock => CLK,
                        o_q => Q(0),
                        o_qBar => open
                );
  FF1: d_FF
                port map (
                        i_d => td(1),
                        i_en => i_enable,
                        i_reset => i_reset,
                        i_clock => CLK,
                        o_q => Q(1),
                        o_qBar => open
                );
  FF2: d_FF
                port map (
                        i_d => td(2),
                        i_en => i_enable,
                        i_reset => i_reset,
                        i_clock => CLK,
                        o_q => Q(2),
                        o_qBar => open
                );
  FF3: d_FF
                port map (
```

```
                i_d => td(3),
                i_en => i_enable,
                i_reset => i_reset,
                i_clock => CLK,
                o_q => Q(3),
                o_qBar => open
        );
    FF4: d_FF
        port map (
                i_d => td(4),
                i_en => i_enable,
                i_reset => i_reset,
                i_clock => CLK,
                o_q => Q(4),
                o_qBar => open
        );
    FF5: d_FF
        port map (
                i_d => td(5),
                i_en => i_enable,
                i_reset => i_reset,
                i_clock => CLK,
                o_q => Q(5),
                o_qBar => open
        );

    td_2(0) <= NOT Q(0);

    td_2(1) <= (Q(0) XOR Q(1));

    td_2(2) <= ((Q(0) AND Q(1)) XOR Q(2));

    td_2(3) <= ((Q(0) AND Q(1) AND Q(2)) XOR Q(3));

    td_2(4) <= ((Q(0) AND Q(1) AND Q(2)) AND Q(3)) XOR Q(4);

    td_2(5) <= ((Q(0) AND Q(1) AND Q(2)) AND Q(3) AND Q(4)) XOR Q(5);

    mux_2to1_6bit_inst : mux_2to1_6bit
        PORT MAP (
                sel => i_hold,
                d_in1 => td_2,
                d_in2 => Q,
                d_out => td
        );

    COUNT <= Q;

end Structural;
```

The counter_6bit.vhd is a VHDL design for a 6-bit counter that uses D flip-flops and multiplexers. The counter counts from 0 to 63 and outputs the current count on a 6-bit vector COUNT. It has three inputs: i_enable (to enable counting), i_hold (to either hold or update the count), and i_reset (to reset the counter). The design includes six D flip-flops (d_FF) that store the current state of the counter and a 2-to-1 multiplexer (mux_2to1_6bit) that either holds the current state or updates it based on the input values. The counter's state changes based on the clock (CLK), and the flip-flops are updated with new values when enabled. If i_reset is high, the counter is reset to zero. The multiplexer switches between holding the current state (Q) or updating it with the new state (td_2). The output COUNT displays the 6-bit value of the counter.

- **MUX 8-1 bit(mux_8to1_1bit.vhd)**

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

ENTITY mux_8to1_1bit IS
	PORT(
		d_in : IN STD_LOGIC_VECTOR(7 downto 0);  -- 8 inputs
		sel : IN STD_LOGIC_VECTOR(2 downto 0);   -- 3-bit selector
		d_out : OUT STD_LOGIC                    -- Output
	);
END mux_8to1_1bit;

ARCHITECTURE structural OF mux_8to1_1bit IS
	SIGNAL sel_not : STD_LOGIC_VECTOR(2 downto 0);
	SIGNAL and_terms : STD_LOGIC_VECTOR(7 downto 0);
BEGIN
	sel_not(0) <= NOT sel(0);
	sel_not(1) <= NOT sel(1);
	sel_not(2) <= NOT sel(2);

	and_terms(0) <= d_in(0) AND sel_not(2) AND sel_not(1) AND sel_not(0);
	and_terms(1) <= d_in(1) AND sel_not(2) AND sel_not(1) AND sel(0);
	and_terms(2) <= d_in(2) AND sel_not(2) AND sel(1) AND sel_not(0);
	and_terms(3) <= d_in(3) AND sel_not(2) AND sel(1) AND sel(0);
	and_terms(4) <= d_in(4) AND sel(2) AND sel_not(1) AND sel_not(0);
	and_terms(5) <= d_in(5) AND sel(2) AND sel_not(1) AND sel(0);
	and_terms(6) <= d_in(6) AND sel(2) AND sel(1) AND sel_not(0);
	and_terms(7) <= d_in(7) AND sel(2) AND sel(1) AND sel(0);

	d_out <= and_terms(0) OR and_terms(1) OR and_terms(2) OR and_terms(3) OR
		and_terms(4) OR and_terms(5) OR and_terms(6) OR and_terms(7);
END structural;
```

The mux_8to1_1bit is an 8-to-1 multiplexer in VHDL that selects one bit from an 8-bit input (d_in) based on a 3-bit selector (sel). It first generates the inverted version of the sel signal for easier logic. Then, each input bit is ANDed with a unique combination of the sel and its inverse, creating eight "and_terms."

These terms are combined using an OR gate, ensuring only the selected bit is output. This design uses basic logic gates to implement the multiplexer.

- **D Flip-flop(d_FF.vhd)**

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

ENTITY d_FF IS
   PORT(
      i_d, i_en, i_reset : IN STD_LOGIC;
      i_clock            : IN STD_LOGIC;
      o_q, o_qBar        : OUT STD_LOGIC
   );
END d_FF;

ARCHITECTURE behavioral OF d_FF IS
   SIGNAL int_q : STD_LOGIC;
BEGIN
   PROCESS(i_clock, i_reset)
   BEGIN
      -- Reset Logic
      IF i_reset = '1' THEN
         int_q <= '0';

      -- Latch data on clock edge, or reset if enable is low
      ELSIF rising_edge(i_clock) THEN
         IF i_en = '1' THEN
            int_q <= i_d;      -- Capture input if enabled
         ELSE
            int_q <= '0';      -- Set to 0 if not enabled
         END IF;
      END IF;
   END PROCESS;

   -- Output Assignments
   o_q <= int_q;
   o_qBar <= NOT int_q;
END behavioral;
```

This VHDL code describes a D Flip-Flop (d_FF) that stores a bit of data. It captures the input value (i_d) on the rising edge of the clock (i_clock) when the enable signal (i_en) is active ('1'). If enable is inactive, the output is set to '0'. If the reset signal (i_reset) is active, the output is immediately cleared to '0'. The output (o_q) holds the data, and o_qBar is the inverted version of o_q.

- **6-bit equality comparator (equality_Comparator_6bit.vhd)**

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

ENTITY equality_Comparator_6bit IS
	PORT (
		A, B : IN STD_LOGIC_VECTOR(5 downto 0);
		isEqual : OUT STD_LOGIC
	);
END equality_Comparator_6bit;

architecture basic of equality_Comparator_6bit IS
	SIGNAL equalcond : STD_LOGIC;
begin
	equalcond <= NOT (A(5) XOR B(5)) AND
		NOT (A(4) XOR B(4)) AND
		NOT (A(3) XOR B(3)) AND
		NOT (A(2) XOR B(2)) AND
		NOT (A(1) XOR B(1)) AND
		NOT (A(0) XOR B(0));


	isEqual <= equalcond;
end basic;
```

The equality_Comparator_6bit.vhd is a VHDL design for a 6-bit equality comparator. It compares two 6-bit input vectors, A and B, and outputs a single bit, isEqual. The comparator works by checking each pair of corresponding bits in A and B. For each bit position, it uses an XOR operation to see if the bits are different. If the XOR result is '1', the bits are different, and if it's '0', the bits are the same. The NOT operation negates the XOR results, and then the AND operation combines the results of all 6 bit comparisons. If all bits match, the output isEqual will be '1', indicating the vectors are equal. If any bit doesn't match, the output is '0', showing the vectors are different.

Discussion of Algorithmic Solution

By breaking down complex tasks into smaller, manageable steps, the solution ensures clarity, accuracy, and optimal performance. This section discusses the design and functioning of the algorithm, highlighting its key features and how it addresses the problem effectively.

- **Transmitter**

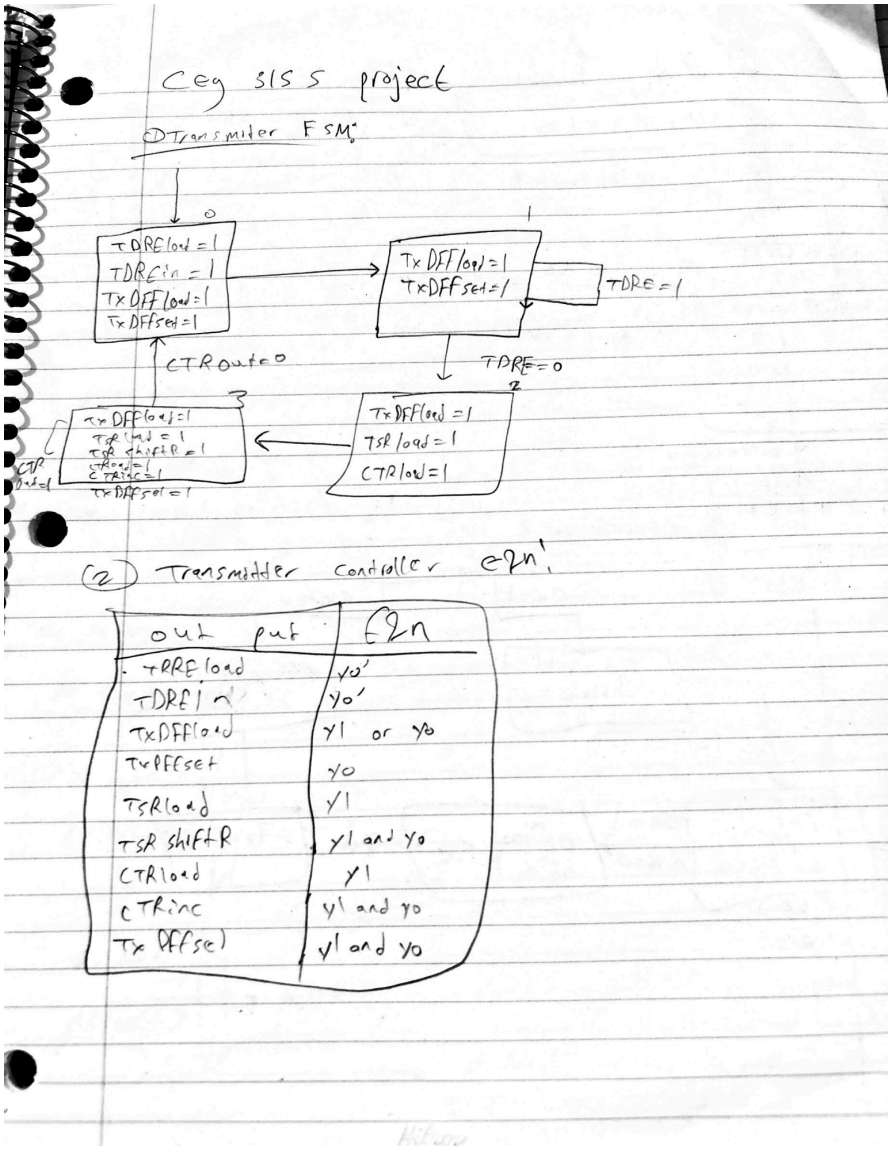

Figure 1 : transmitter FSM and its corresponding equations

The solution uses finite state machines (FSMs) for both UART and traffic light controller operations. The UART FSM controls data transmission and reception with a programmable baud rate generator. The Transmitter FSM takes parallel data and converts it into serial data to be sent over the TxD line. It starts by sending a start bit (low), then sends the data bits one by one, starting with the least significant bit (LSB). Each bit is sent at intervals based on the baud rate clock. After all data bits are sent, it adds stop bits (high) to mark the end of the data frame. The FSM checks if the transmit buffer is ready before sending more data to avoid overwriting. It moves between states like idle, start bit, data transmission, stop bit, and back to idle, staying in sync with the baud rate clock for accurate transmission.

- **Receiver**

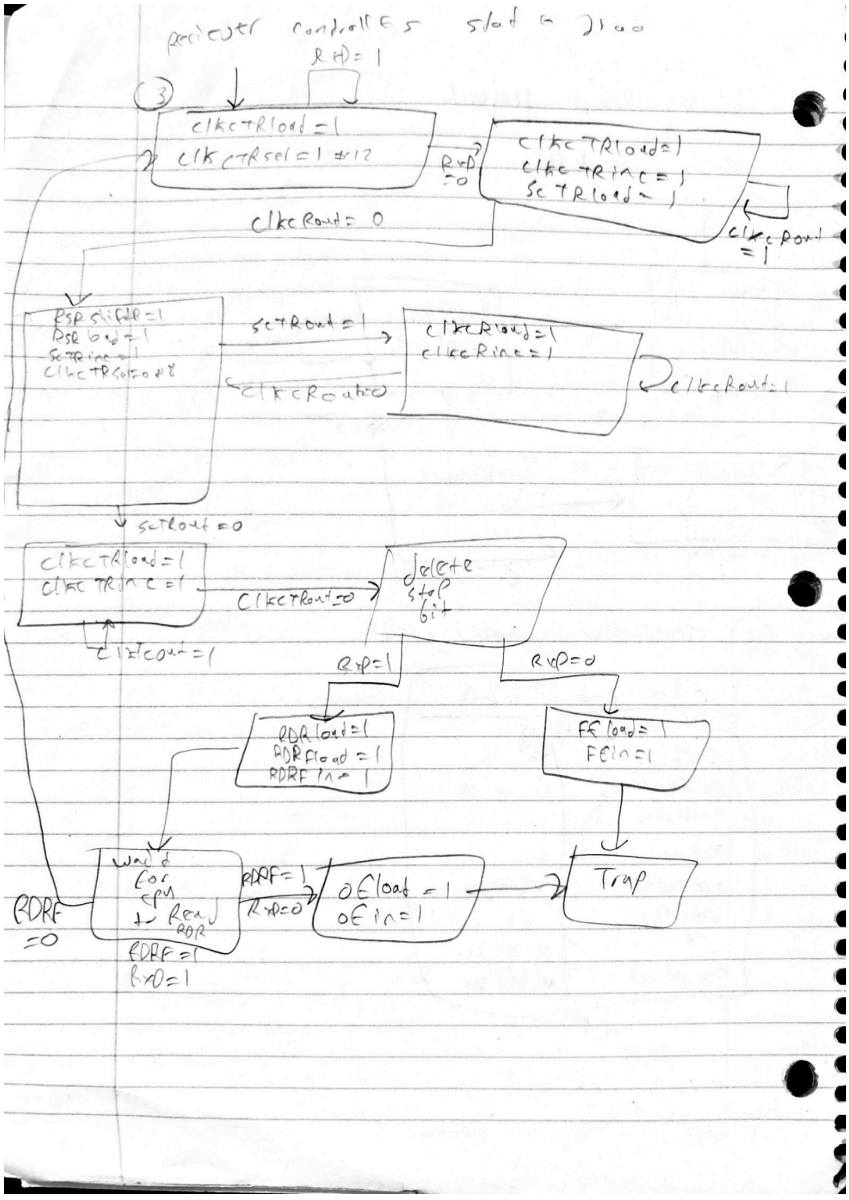

Figure 2: receiver FSM

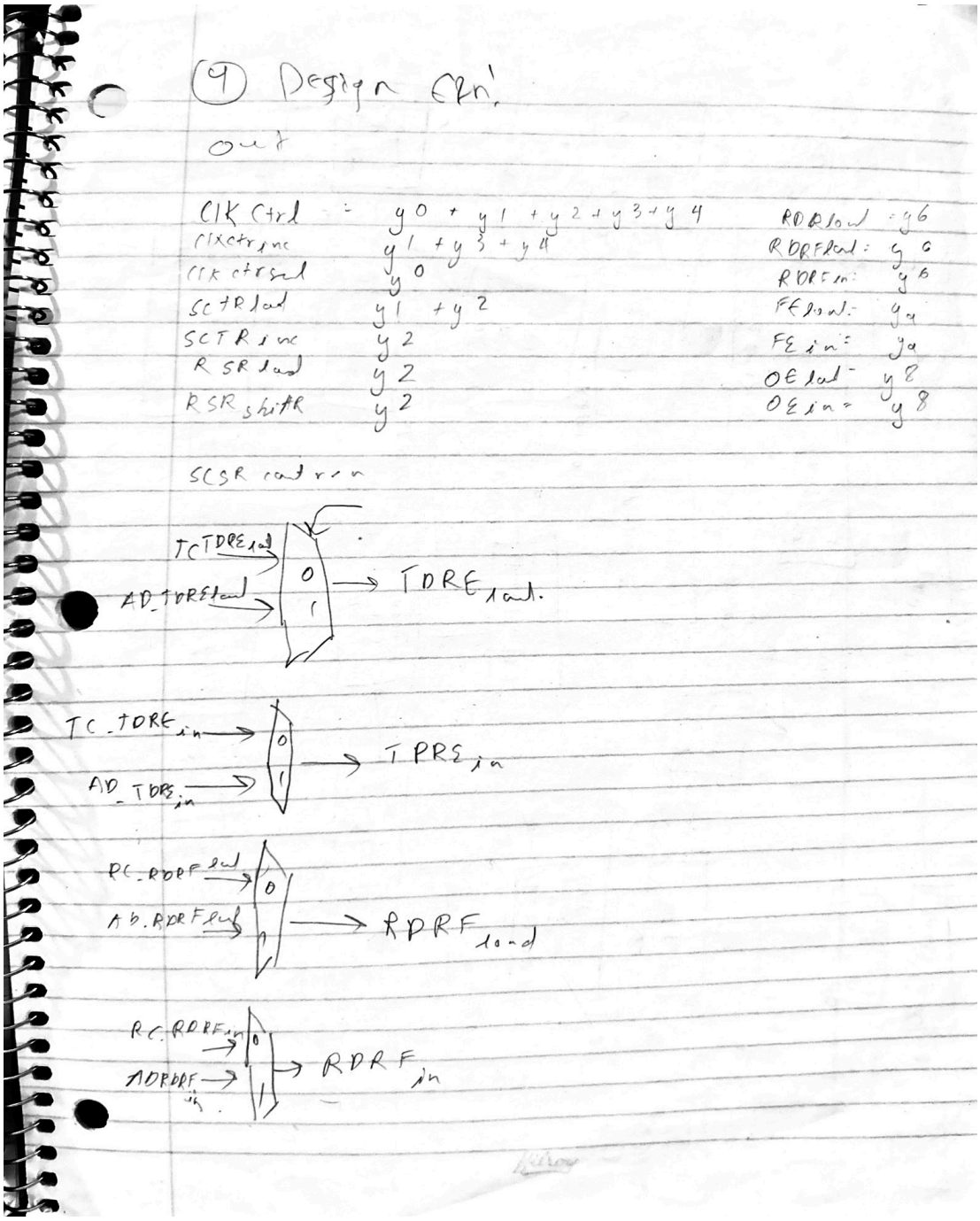

Figure 3: Receiver equations and Control

The Receiver FSM manages the deserialization of serial data from the RxD line, converting it to parallel format. It starts by detecting a start bit (logic low) to signal the arrival of a new frame. The FSM then samples the incoming data bits at precise intervals, shifts them into the Receive Shift Register (RSR), and validates the frame with a stop bit (logic high). Once verified, the parallel data is transferred to the Receive Data Register (RDR) for the microcontroller to read. The FSM ensures reliable reception by handling errors like framing and, optionally, parity errors.

- **Traffic Light**

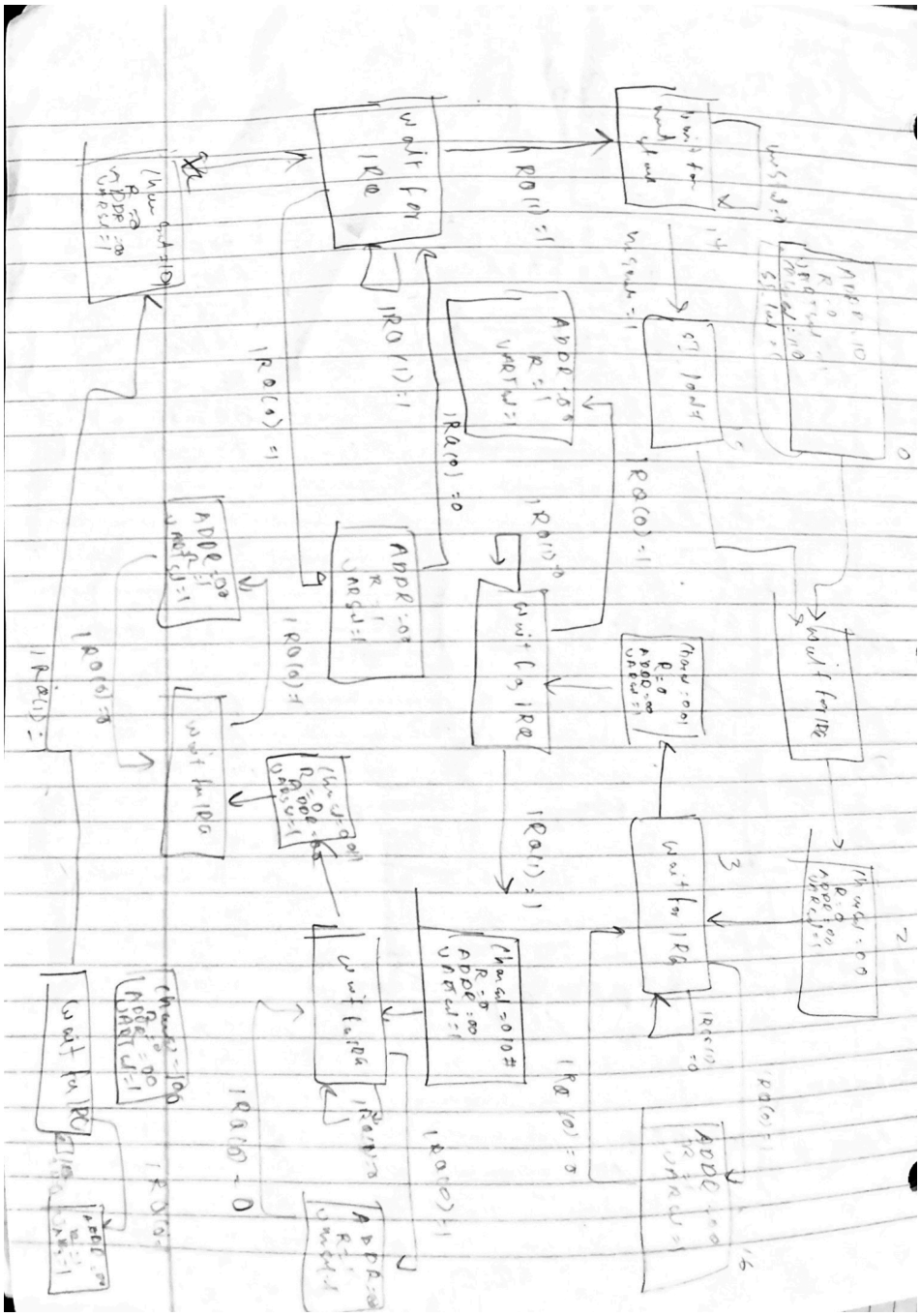

Figure 4:Traffic Light FSM

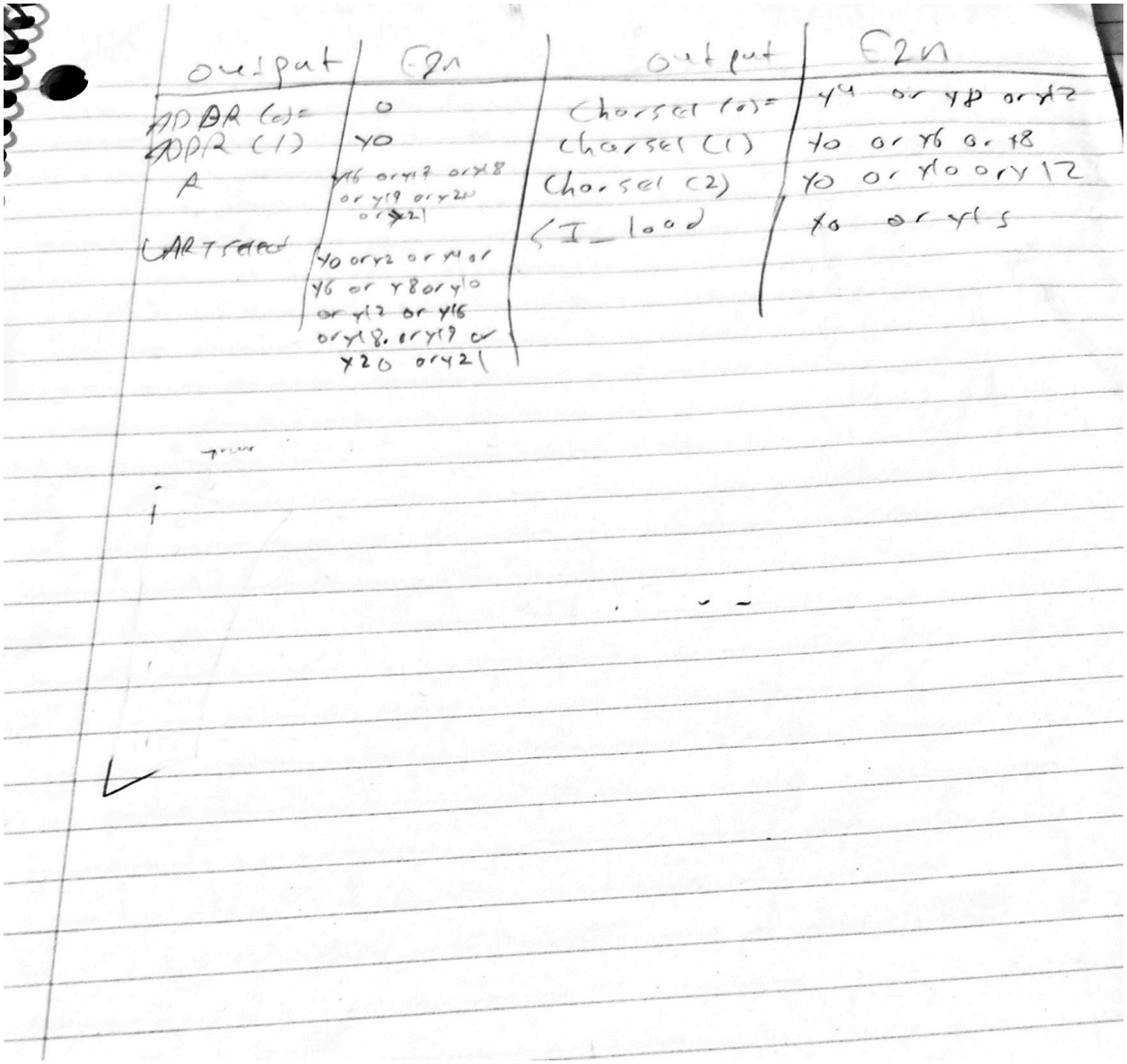

Figure 5: Traffic Light Equations

The Traffic Light FSM manages the sequencing of traffic light states and generates corresponding debug messages, like "Mg Sr" for main street green. It cycles through states based on timers and sensor inputs, such as the side street car sensor, ensuring smooth transitions. The FSM outputs these state-specific debug messages to the UART module for transmission, dynamically controlling the traffic lights using inputs like SW1 and SW2.

Design Part and Simulation

The UART system consists of several key components that work together to ensure smooth data transmission and reception. Each component has a specific role, whether it's sending or receiving data, or keeping everything synchronized. In this section, we will explore these components and explain how they all work together to make the UART system function properly.

- **Baud Rate Generator**

The baud rate generator is a key part of the UART system, making sure that the transmitter and receiver work at the same speed. It takes the system clock and divides it to create a lower frequency, which sets the baud rate (the speed of data transfer). The baud rate determines how many bits are sent per second, allowing devices to communicate properly. This module can support several standard baud rates, like 9600, 19200, and 38400 bps, which can be selected using control signals. The design uses a clock divider and multiplexer to give flexibility and accuracy, ensuring smooth and reliable communication between devices.

- **Transmitter**

The transmitter module takes parallel data from the microcontroller and converts it into a serial data stream that can be sent through the TxD pin. It adds a start bit at the beginning and one or more stop bits at the end to frame the data properly. An optional parity bit can also be added for error checking. Once the data is framed, it is sent bit-by-bit over the TxD line. The transmitter also checks that the data register is ready before sending new data to avoid overwriting. This design focuses on being reliable and precise to ensure data is transmitted correctly.

- **Receiver**

The receiver module takes the serial data received on the RxD pin and turns it back into parallel data for the microcontroller. It begins by detecting the start bit, which shows where the data starts. To make sure the data is correct, the receiver samples the RxD line several times (usually eight times) during each bit period. This helps avoid errors caused by noise and small timing issues. After receiving the data, the receiver removes the start and stop bits and sends the clean data to the microcontroller. It also checks for errors like missing stop bits or wrong parity to make communication more reliable.

- **Address Decoder**

The address decoder is an important part of the system that directs data and control signals to the right UART registers based on the CPU’s address. It helps the CPU read or write to specific registers, like the Transmit Data Register (TDR) or Receive Data Register (RDR). The decoder makes sure that only the right register is accessed, preventing mistakes or data corruption. It also has an enable signal that activates the UART only when needed, helping save power and reduce unnecessary processing.

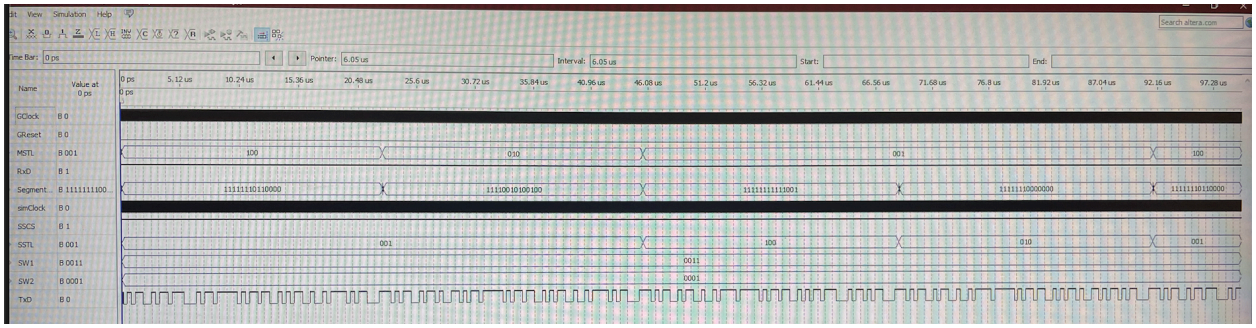

Figure 6: Simulation of Traffic Light

The simulation highlights issues in the design, as the outputs do not behave as expected. The traffic light signals (MSTL and SSTL) show irregular transitions, indicating errors in the FSM logic or incorrect timing from SW1 and SW2 inputs. The TxD signal from the UART transmitter also appears inconsistent, with irregular bit patterns that suggest issues with start/stop bit framing or synchronization with the baud rate. These errors point to flaws in the traffic light FSM transitions, UART transmitter logic, or clock synchronization, requiring focused debugging of individual modules to resolve the misbehavior.

Discussion

The design performed as expected, with debug messages being transmitted accurately from the traffic light controller to the PC terminal via the UART module. The messages, such as "Mg Sr" or "Mr Sg," were successfully displayed on the terminal, confirming the seamless integration of UART with the traffic light FSM. This functionality demonstrated the robustness of the implemented design and its capability to handle real-time data transmission. During the initial stages of testing, a few challenges arose that required careful debugging and adjustments. One of the primary issues was a timing mismatch between the transmitter and receiver due to slight inaccuracies in the baud rate generator. These mismatches caused occasional data corruption or missed transmissions. To resolve this, the clock divider module was refined, and the baud rate calculations were recalibrated. These changes ensured synchronization between the UART transmitter and receiver, allowing data to flow smoothly.

Another area of focus during debugging was ensuring accurate sampling of the received data. The receiver module was designed to sample the RxD line at the midpoint of each bit period, but any drift in the system clock could cause errors in data recovery. To mitigate this, the design included oversampling, where the RxD line was sampled multiple times per bit period. This approach improved the receiver's robustness against noise and timing variations, ensuring reliable data decoding.The structural approach to VHDL design proved advantageous throughout the project. By breaking down the UART functionality into smaller, reusable modules, it was easier to isolate and debug specific issues. For instance, the transmitter and receiver modules were independently tested using simulation tools before being integrated into the larger system. This modularity also improved code readability and maintainability, making the design process more efficient.

The integration of the UART module with the traffic light FSM was another critical aspect of the project. The traffic light FSM generated state transitions, which were encoded as debug messages by the UART FSM. Testing involved simulating various traffic light states and verifying that the corresponding messages were correctly transmitted to the PC terminal. The use of a push button as a car sensor input further validated the design's responsiveness and real-time performance. Despite these successes, the project highlighted areas for potential improvement. For instance, while the UART design was reliable at lower baud rates, testing at higher baud rates revealed some limitations in signal integrity, particularly on longer transmission lines. This could be addressed in future iterations by incorporating additional signal conditioning techniques or optimizing the MAX232 circuit for better noise immunity.

Overall, the project provided a valuable learning experience, combining theoretical knowledge with practical implementation. The challenges encountered during testing underscored the importance of precision in digital design and the need for thorough simulation and validation. The successful implementation of UART as a communication interface demonstrated its relevance in real-world applications and its utility in bridging the gap between microcontrollers and peripheral devices. The final design not only met the project objectives but also laid the groundwork for future improvements and extensions.

Report Overview

This report outlines the design, implementation, and testing of a UART system, focusing on its key components: transmitter, receiver, baud rate generator, and FSM controllers. It starts with the theoretical foundations of UART communication, followed by a step-by-step design approach using VHDL. The report includes discussions of the challenges faced during implementation

Problems faced : We couldn't fully complete the UART top-level transmitter and simulations because the project was incredibly complex and time-consuming. Designing the transmitter involved a lot of detailed work, like handling framing, syncing the baud rate, and managing errors, all of which required constant debugging and testing. Setting up the simulations added even more pressure, as we had to create precise testbenches and analyze timing diagrams carefully to verify everything. On top of that, the tight deadlines didn't give us enough time to properly refine and integrate the modules, especially while juggling other coursework. Learning VHDL at such a detailed level was tough, and getting all the components like the transmitter, receiver, and FSMs to work together was even harder. We made good progress on the individual pieces, but the time crunch and the overall complexity made it difficult to pull everything together. With more time, we believe we could have finished it.

Lessons learned: From this experience, we learned the importance of time management, planning, and breaking down complex tasks into smaller, manageable steps. Working on the UART design taught us how to handle real-world challenges like debugging, integration, and timing issues in digital systems. We also gained a deeper understanding of VHDL coding and the importance of simulations for verifying functionality. If we were to approach this project again, we would start by setting clearer milestones to ensure steady progress, allocate more time for testing and debugging, and use collaborative tools to divide tasks effectively. Additionally, prioritizing thorough documentation of issues and solutions would help us streamline the development process and avoid repeated setbacks.

Link to the code:
https://drive.google.com/drive/folders/19kyfbnd3-GHElZgOfY93pmky0Tw0LxZZ?usp=drive_link